Université de Montréal

Intégration de la visualisation à multiples vues pour le développement du logiciel

par
Guillaume Langelier

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des arts et des sciences
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en informatique

décembre, 2010

© Guillaume Langelier, 2010.

Université de Montréal
Faculté des arts et des sciences

Cette thèse intitulée:

Intégration de la visualisation à multiples vues pour le développement du logiciel

présentée par:

Guillaume Langelier

a été évaluée par un jury composé des personnes suivantes:

Neil Stewart,	président-rapporteur
Houari Sahraoui,	directeur de recherche
Pierre Poulin,	codirecteur
Bruno Dufour,	membre du jury
Oscar Nierstrasz,	examinateur externe
Michel Delfour,	représentant du doyen de la FES

Thèse acceptée le: 22 mars 2011

RÉSUMÉ

Le développement du logiciel actuel doit faire face de plus en plus à la complexité de programmes gigantesques, élaborés et maintenus par de grandes équipes réparties dans divers lieux. Dans ses tâches régulières, chaque intervenant peut avoir à répondre à des questions variées en tirant des informations de sources diverses. Pour améliorer le rendement global du développement, nous proposons d'intégrer dans un IDE populaire (*Eclipse*) notre nouvel outil de visualisation (*VERSO*) qui calcule, organise, affiche et permet de naviguer dans les informations de façon cohérente, efficace et intuitive, afin de bénéficier du système visuel humain dans l'exploration de données variées. Nous proposons une structuration des informations selon trois axes : (1) le contexte (qualité, contrôle de version, bogues, etc.) détermine le type des informations ; (2) le niveau de granularité (ligne de code, méthode, classe, paquetage) dérive les informations au niveau de détails adéquat ; et (3) l'évolution extrait les informations de la version du logiciel désirée. Chaque vue du logiciel correspond à une coordonnée discrète selon ces trois axes, et nous portons une attention toute particulière à la cohérence en naviguant entre des vues adjacentes seulement, et ce, afin de diminuer la charge cognitive de recherches pour répondre aux questions des utilisateurs. Deux expériences valident l'intérêt de notre approche intégrée dans des tâches représentatives. Elles permettent de croire qu'un accès à diverses informations présentées de façon graphique et cohérente devrait grandement aider le développement du logiciel contemporain.

Mots clés: Visualisation, développement de logiciel, environnement de développement, intégration, évolution du logiciel, animation.

ABSTRACT

Nowadays, software development has to deal more and more with huge complex programs, constructed and maintained by large teams working in different locations. During their daily tasks, each developer may have to answer varied questions using information coming from different sources. In order to improve global performance during software development, we propose to integrate into a popular integrated development environment (*Eclipse*) our new visualization tool (*VERSO*), which computes, organizes, displays and allows navigation through information in a coherent, effective, and intuitive way in order to benefit from the human visual system when exploring complex data. We propose to structure information along three axes: (1) context (quality, version control, etc.) determines the type of information; (2) granularity level (code line, method, class, and package) determines the appropriate level of detail; and (3) evolution extracts information from the desired software version. Each software view corresponds to a discrete coordinate according to these three axes. Coherence is maintained by navigating only between adjacent views, which reduces cognitive effort as users search information to answer their questions. Two experiments involving representative tasks have validated the utility of our integrated approach. The results lead us to believe that an access to varied information represented graphically and coherently should be highly beneficial to the development of modern software.

Keywords: Visualization, software development, development environment, integration, software evolution, animation.

TABLE DES MATIÈRES

RÉSUMÉ . . . . **iii**

ABSTRACT . . . . **iv**

TABLE DES MATIÈRES . . . . **v**

LISTE DES TABLEAUX . . . . **ix**

LISTE DES FIGURES . . . . **x**

LISTE DES ANNEXES . . . . **xii**

LISTE DES SIGLES . . . . **xiii**

DÉDICACE . . . . **xiv**

REMERCIEMENTS . . . . **xv**

CHAPITRE 1 : INTRODUCTION **1**

1.1 Contexte 1

1.2 Problématique 4

1.3 Proposition 7

1.4 Structure de cette thèse 10

CHAPITRE 2 : ÉTAT DE L'ART **11**

2.1 Psychologie cognitive et perception 11

2.2 Visualisation scientifique et visualisation d'informations 14

2.3 Visualisation du logiciel 16

2.4 Animation et visualisation 19

2.5 Visualisation de l'évolution du logiciel 22

2.6 Langages visuels 26

2.7 Approches intégrées 27

CHAPITRE 3 : APPROCHE 31

3.1 Métriques pour la représentation des logiciels 34

3.2 Calcul des données en direct 35

3.3 Visualisation 37

3.3.1 Axes et multiples vues 37

3.4 Principe de cohérence 40

3.4.1 La cohérence liée à un outil intégré 41

3.4.2 La cohérence visuelle 42

CHAPITRE 4 : VISUALISATION 45

4.1 Granularité des représentations 45

4.1.1 Représentation des classes 47

4.1.2 Représentation des méthodes 52

4.1.3 Représentation des paquetages 55

4.1.4 Représentation des relations 58

4.1.5 Représentation des lignes de code 59

4.1.6 Navigation et déplacement entre les niveaux de granularité . . . 61

4.1.7 Précisions sur la cohérence 63

4.2 Représentation des différents contextes 64

4.2.1 Précisions sur la cohérence 65

4.2.2 Évolution du logiciel 66

CHAPITRE 5 : INTÉGRATION DANS L'ENVIRONNEMENT DE DÉVELOPPEMENT 70

5.1 Calcul des métriques dans *Eclipse* 70

5.2 Description des métriques calculées 72

5.2.1 Couplage 73

5.2.2 Complexité 75

5.2.3 Cohésion 75

5.2.4 Héritage 75
5.2.5 Auteur principal 76
5.2.6 Programmeur principal en charge des bogues 76
5.2.7 Nombre de bogues 77
5.2.8 Nombre de *commits* 77
5.2.9 Date 77
5.2.10 Métriques de lignes de code 77
5.3 Incorporation de la visualisation 78
5.4 Interface graphique et ramifications avec *Eclipse* 79
5.5 Ajouts et suppressions d'éléments 81

CHAPITRE 6 : ÉVALUATION 83
6.1 Contexte 83
6.2 Revue de quelques évaluations en visualisation 85
6.2.1 Théorie 85
6.2.2 Visualisation en dehors du logiciel 87
6.2.3 Outils de développement 89
6.2.4 Visualisation du logiciel 90
6.3 Expérience 1 : Étude de cas *in vitro* 91
6.3.1 Méthode 91
6.3.2 Les résultats 95
6.3.3 Discussion 97
6.3.4 Menaces à la validité 100
6.4 Expérience 2 : Étude de cas *in vivo* 101
6.4.1 Méthode 102
6.4.2 Résultats 106
6.4.3 Discussion 110
6.4.4 Menaces à la validité 114
6.5 Évaluation globale et conclusions 115

CHAPITRE 7 : CONCLUSION **118**
7.1 Réalisations 118
7.2 Contributions 120
7.3 Travaux futurs 123
7.4 Quelques réflexions sur la visualisation du logiciel dans l'avenir 125

BIBLIOGRAPHIE **127**

LISTE DES TABLEAUX

5.I Associations pour la qualité 73

5.II Associations pour *SVN* 74

5.III Associations pour les bogues 74

6.I Tableau des réussites des activités 96

6.II Tableau de l'utilité perçue de *VERSO* 97

6.III Proportion d'utilisation de *VERSO* 98

6.IV Cote d'appréciation de l'utilité pour *VERSO* 98

6.V Fonctionnalités les plus utiles 99

6.VI Expérience préalable au projet pour la 2e étude de cas 103

6.VII Tableau de l'utilisation des vues dans *VERSO* 111

6.VIII Tableau sur la qualité et la performance dans *VERSO* 112

6.IX Tableau sur l'évaluation subjective de *VERSO* 113

LISTE DES FIGURES

1.1 Une vue de l'environnement de développement intégré *Eclipse* . . 3

1.2 Représentation des métriques de qualité pour le logiciel *PCGEN* . 8

1.3 *VERSO* intégré à l'IDE *Eclipse* 9

2.1 Un extrait des travaux de Healey 12

2.2 Exemple de visualisation en médecine montrant les cavités sinusales 15

2.3 Exemple de visualisation d'informations pour représenter les tendances du vote 16

2.4 Outil *SEESOFT* 18

2.5 Représentation de *ArgoUML* avec *Code City* 19

2.6 Une vue d'*EXTRAVIS* 21

2.7 Visualisation de l'évolution du logiciel à l'aide d'une matrice . . . 23

2.8 Représentation à l'aide d'un diagramme de Kiviat 24

2.9 Exemple de l'*Evolution Radar* 25

2.10 Exemple de *code swarm* 26

2.11 Vue de fragments d'informations (Fritz *et al.*) 29

3.1 Matrice des différentes vues atteignables selon les trois dimensions 33

3.2 Présentation de la cohérence 39

4.1 Matrice des différentes vues accessibles 46

4.2 Exemples des niveaux de granularité 46

4.3 Exemple des trois caractéristiques graphiques liées aux classes . . 48

4.4 Illusion de Müller-Lyer 49

4.5 Représentation des classes 50

4.6 Explication de l'algorithme modifié du *Treemap* 51

4.7 Exemple de l'algorithme modifié du *Treemap* 52

4.8 Principe de placement des méthodes 54

4.9 Représentation du niveau de granularité des méthodes 55

4.10 Représentation du niveau de granularité des paquetages 57
4.11 Représentation du système de filtre pour les relations dans *VERSO* 59
4.12 Représentation du niveau de granularité des lignes 61
4.13 Exemple du zoom sémantique 63
4.14 Exemples des contextes 66

5.1 Exemple de *VERSO* intégré dans *Eclipse* 79

6.1 Questions expérience 1, partie 1 93
6.2 Questions expérience 1, partie 2 94
6.3 Questions expérience 1, partie 3 95
6.4 Expérience préalable 102
6.5 Questions expérience 2, partie 1 105
6.6 Questions expérience 2, partie 2 106
6.7 Questions expérience 2, partie 3 107
6.8 Questions expérience 2, partie 4 108
6.9 Visualisation des deux projets liés à l'expérience 2 110

LISTE DES ANNEXES

Annexe I : **Questionnaire de la première expérience** . . . . **xvi**

Annexe II : **Questionnaire de l'expérience préalable au projet** . . . . . **xxi**

Annexe III : **Projet (première partie)** . . . . **xxii**

Annexe IV : **Projet (deuxième partie)** . . . . **xxvi**

Annexe V : **Questionnaire de la deuxième expérience** . . . . **xxviii**

LISTE DES SIGLES

CBO	*Coupling Between Objects*
CRSNG	Conseil de Recherches en Sciences Naturelles et en Génie du Canada
DIT	*Depth in Inheritance Tree*
IDE	*Integrated Development Environment*
LCOM5	*Lack of COhesion in Method Version 5*
RGB	*Red Green Blue*
UML	*Unified Modeling Language*
VERSO	Visualisation pour l’Évaluation et la Rétro-ingénierie du *SOftware*
WMC	*Weighted Methods per Class*

à Marie-Claude.

REMERCIEMENTS

Je tiens à remercier toutes les personnes qui ont participé à la réussite du projet *VERSO*. Des remerciements particuliers vont à Karim Dhambri, Simon Bouvier et Stéphane Vaucher pour leur apport au projet et pour le temps qu'ils y ont investi. J'aimerais aussi remercier tous les autres étudiants du laboratoire pour les idées et les débats qu'ils ont suscités lors de mes recherches.

Je tiens à remercier mes directeur et co-directeur Houari Sahraoui et Pierre Poulin pour m'avoir donné la chance de faire de la recherche sous leur direction. J'aimerais les remercier pour leur aide précieuse tout au long du projet ainsi que pour le temps qu'ils ont consacré à la correction des versions préliminaires de cette thèse et des articles.

Je remercie les organismes subventionnaires *CRSNG* et *FQRNT* pour m'avoir épaulé dans mes études supérieures à différents moments. Les bourses fournies par l'Université de Montréal et le département d'informatique et de recherche opérationnelle (DIRO) ont aussi été d'un grand soutien durant mes études.

Les sujets ayant participé aux expériences méritent des remerciements particuliers pour avoir pris de leur temps bénévolement pour faire avancer ce projet.

Finalement, j'aimerais remercier mes parents pour leur soutien de tous les instants tant au niveau académique que moral. Leurs encouragements, leur soutien financier ainsi que leurs judicieux conseils ont été des facteurs indispensables à la réussite de ce doctorat. Des remerciements particuliers vont à ma compagne, Marie-Claude Parenteau, pour sa patience et son soutien aussi altruiste qu'infatigable. Je veux aussi remercier mes amis pour leurs encouragements soutenus durant mon doctorat.

CHAPITRE 1

INTRODUCTION

Cette thèse de doctorat repose sur la construction d'un environnement intégré de développement du logiciel avec support visuel. Cet environnement organise et affiche plusieurs aspects du logiciel à travers différents niveaux de granularité. Il sert tant pour la phase de conception du logiciel durant laquelle les utilisateurs se questionnent pour progresser dans leurs tâches, que pour son évaluation à travers diverses informations. Ces informations sont extraites du code source, des bogues répertoriés, des programmeurs responsables de modifications, la liste des *commits*, etc., et prennent la forme de métriques[1]. Des informations sur la structure comme les relations entre les éléments, la liste des auteurs des éléments, etc., viennent compléter les informations fournies par les métriques. Les utilisateurs de cet environnement sont alors en mesure de mieux répondre à des questions sur les différents aspects du développement du logiciel et d'en évaluer la qualité au fur et à mesure de sa construction. Ils peuvent aussi suivre l'évolution du logiciel à travers les multiples versions selon les informations énumérées plus haut. Dans le but d'intégrer la visualisation de meilleure façon, les utilisateurs peuvent interagir directement avec la vue graphique pour construire et modifier le code. Cet environnement intégré offre une nouvelle vision du développement du logiciel.

1.1 Contexte

Le développement du logiciel est une activité complexe qui demande la connaissance et l'interprétation de plusieurs aspects du logiciel à différents niveaux. Même dans les cas où le logiciel est très modulaire, il est important d'en garder une vue d'ensemble. La taille du logiciel est un problème crucial à la compréhension, que même la modularité

[1]Dans cette thèse, l'appellation *métrique* désignera toute information reliée au logiciel présentée sous forme de valeur singleton attribuée à un élément. Pour être considérée métrique, l'information n'a pas besoin d'être le résultat d'un calcul élaboré ou d'être reliée directement à la qualité de maintenance du logiciel, comme c'est le cas dans d'autres publications en génie logiciel. Le nom d'un programmeur est considéré une métrique au même titre que la cohésion d'une classe.

ne peut pas toujours résoudre. Cette connaissance est encore plus difficile à acquérir quand plusieurs programmeurs séparés sur des sites différents doivent collaborer pour faire évoluer un même code. Même si la contribution de chacun est limitée à une partie précise du logiciel, une connaissance des autres parties est nécessaire pour éviter les problèmes de compatibilité dans les phases ultérieures du projet ou de reproduire dans sa partie isolée, du code existant déjà ailleurs. De plus, la vue d'ensemble est importante pour conserver une bonne qualité du produit logiciel, non pas seulement à l'intérieur de ses parties individuelles, mais aussi dans son ensemble.

Les environnements de développement intégrés (appelés IDE pour *Integrated Development Environment*) ont su évoluer durant les dernières années pour s'adapter aux besoins des utilisateurs. Des efforts ont été réalisés au point de vue de l'interface graphique, notamment par l'apparition de l'arbre expansible et rétractable (appelé *TreeView* par *Microsoft*) souvent présenté à la gauche de la page de code (voir la figure 1.1 à gauche). Par contre, les améliorations se sont essentiellement concentrées dans l'aide à la rédaction du code et non pas dans les tâches connexes au développement du logiciel. Des exemples intéressants d'aide directe à la rédaction du code comprennent la complétion automatique et la compilation en direct du code. De plus, il existe déjà la possibilité de se diriger vers les définitions des différentes méthodes et vers la définition des types (classes et interfaces) utilisés dans le code. Ces accès à la manière d'hyperliens sont déjà pertinents pour la compréhension du logiciel de manière textuelle et ils gagneraient à être utilisés à l'intérieur d'un outil de visualisation. La figure 1.1 montre une capture d'écran d'*Eclipse* [25], un IDE moderne.

Pour les outils de développement actuels, peu de place est laissée à l'analyse[2] des informations de plus haut niveau. Tantôt, on recherche la rapidité, tantôt on recherche la convivialité dans les outils d'aide directe à la programmation, mais on se soucie peu du fait qu'on doive maintenir le logiciel dans le futur. On accorde plutôt l'importance à l'efficacité et à la production d'un plus grand nombre de lignes de code, alors que cela ne se traduit pas toujours par des coûts moindres sur l'ensemble du projet.

[2]Dans cette thèse, analyse sera utilisée au sens de compréhension et d'interprétation d'informations. Il ne s'agit donc pas d'analyse des besoins comme entendu dans le cycle de vie d'un logiciel, et il n'agit pas non plus d'analyse strictement réservée à la qualité du produit logiciel.

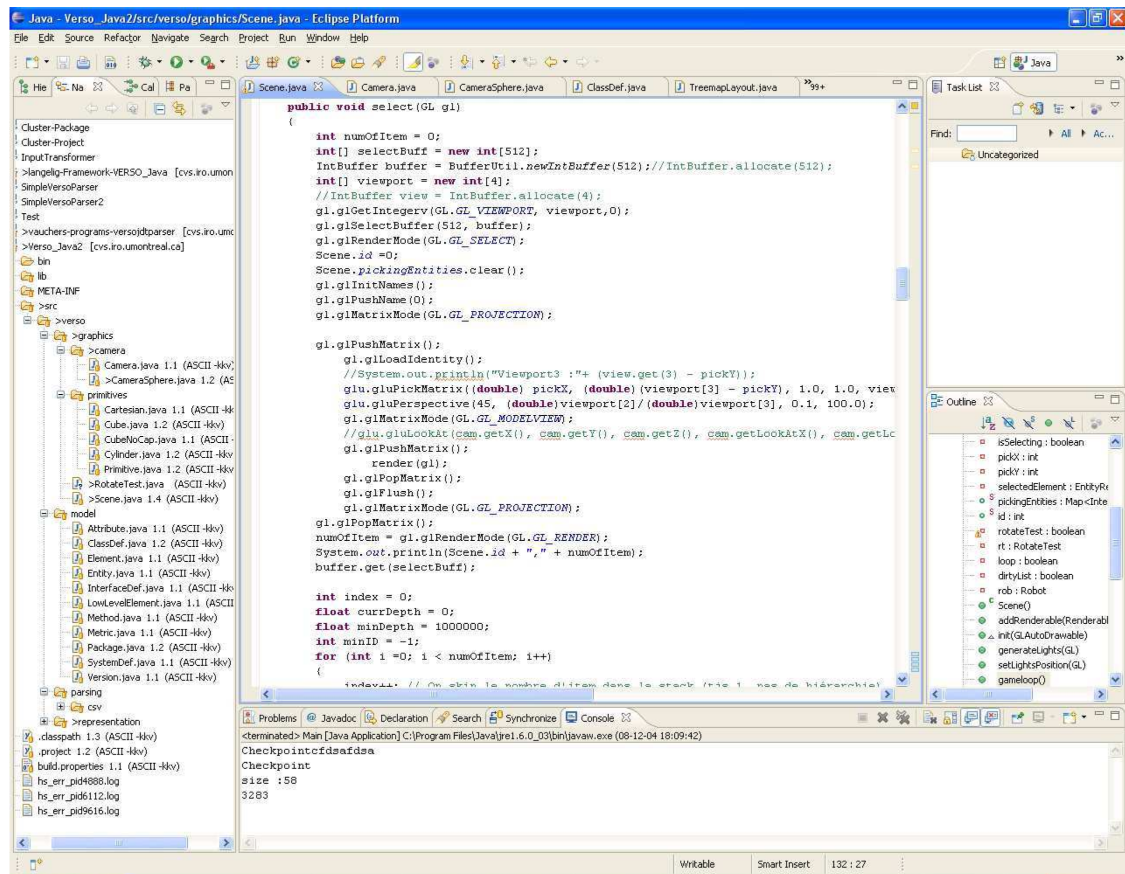

Figure 1.1 – Une vue de l’environnement de développement intégré *Eclipse*. On y voit entre autres à gauche l’arbre expansible et rétractable, qui est une forme simple et populaire de visualisation.

D’un autre côté, en génie logiciel et plus précisément en rétro-ingénierie servant à l’évaluation de la qualité du logiciel, les experts s’accordent pour dire que des aspects plus complexes entrent en jeu dans les coûts globaux d’un système [3]. Ces coûts incluent l’effort de maintenance déployé pour les versions subséquentes d’un projet, la complexité des échanges entre les intervenants dans le développement d’un logiciel, les tests, les bogues créés lors de la conception du logiciel et leur gestion, la complexité de l’architecture du code et de la mise en commun des différentes parties du logiciel, etc. Tous ces facteurs sont déterminants et peuvent faire la différence entre l’échec ou la réussite d’un projet.

La connaissance de l'évolution du logiciel joue aussi un rôle important dans sa construction et son analyse. Il devient plus intuitif de construire un logiciel quand on connaît les étapes qui ont mené à son état actuel. Il est d'autant plus pertinent d'avoir ces informations puisqu'on peut détecter à quel moment un problème est survenu et tenter de l'associer à d'autres événements comme l'apparition d'un nouveau module ou l'arrivée d'un programmeur. L'évolution permet aussi de revenir en arrière pour continuer l'édition du code à un autre moment dans l'histoire du logiciel si les changements récents empêchent d'obtenir les résultats escomptés.

Nous pensons qu'en plus de fournir des outils de rédaction efficaces aux développeurs, il est essentiel de leur fournir des outils pour mieux comprendre et interpréter divers aspects du logiciel et ainsi contextualiser leur travail dans une vision plus globale.

En ce moment, les outils permettant de comprendre des aspects du logiciel autres que le code lui-même sont souvent appelés à l'extérieur de l'outil principal, ce qui s'intègre mal dans un processus de développement au sens plus large. Ces outils se concentrent généralement sur des niches pour résoudre des problèmes plus particuliers et ils s'adressent donc seulement à quelques analystes. La vue globale ou encore l'accès à des informations provenant de plusieurs aspects du logiciel sont souvent écartés de la réalisation des outils d'analyse. Bien que les outils spécialisés peuvent être efficaces pour certaines tâches précises, ils ne permettent généralement pas de répondre à un large éventail de questions des utilisateurs comme c'est le cas de notre proposition.

1.2 Problématique

Le développement du logiciel est une activité complexe qui peut demander l'interprétation de plusieurs facteurs. On compte parmi ceux-ci l'aspect du code lui-même et de ses différentes constructions, l'aspect de la qualité du logiciel et de sa maintenance, l'aspect de la gestion de projet et de l'interaction entre les programmeurs, l'aspect des bogues et de leur correction, etc. Chacun de ces aspects demande une vue différente. Il est donc pertinent d'offrir plusieurs vues sur le logiciel adaptées aux différents types d'analyse. Par contre, pour permettre des analyses plus structurées, ces vues devraient

être intégrées dans un outil et les transitions entre celles-ci devraient être harmonieuses et pertinentes. L'intégration et l'interpolation de ces différentes vues doivent se faire de façon cohérente pour éviter aux utilisateurs des efforts cognitifs importants imposés par la réappropriation des contextes dans plusieurs vues sans liens apparents.

Le développement du logiciel exige l'analyse de différents niveaux de granularité des composants logiciels. En effet, ce développement nécessite d'avoir des informations très précises sur une méthode ou même sur une boucle dans le logiciel, pour les modifier ou pour corriger un bogue les concernant. D'un autre côté, l'ajout d'une fonctionnalité complexe demande d'avoir une vue plus large sur la classe, le paquetage ou même le système. Les questionnements relatifs aux auteurs d'un élément pour leur part demandent d'ajuster le niveau de granularité à l'élément d'intérêt (méthode, classe ou paquetage). De plus, l'analyse de la qualité demande souvent de naviguer entre différents niveaux de granularité pour s'assurer qu'une modification est correcte aussi bien dans la méthode que dans tout le système. Encore une fois, il faut des transitions harmonieuses entre les différents niveaux de granularité pour réduire la complexité des analyses où l'accès à plusieurs niveaux est nécessaire.

Le développement du logiciel demande aussi d'être en mesure d'évaluer et d'améliorer sa qualité pour réduire les coûts importants reliés à la maintenance [35]. Nous pensons que cette évaluation gagne à être faite le plus tôt possible, et même de façon régulière et continue dans le processus de développement. D'ailleurs, les experts s'entendent pour dire que plus un défaut de conception est détecté rapidement dans le processus de développement d'un logiciel, moins ce défaut sera coûteux à corriger [3]. Malgré l'évidence de cet énoncé, il semble qu'on s'intéresse très peu à la qualité du logiciel dans les premières phases de développement. Une façon plus naturelle de faire cette évaluation le plus tôt possible et régulièrement est de l'effectuer en parallèle avec la construction du code, ce qui peut être accompli efficacement en intégrant ce processus à même les IDE.

L'analyse d'un logiciel à travers tous les aspects mentionnés plus haut demande l'assimilation d'une grande quantité de données. Dans des travaux antérieurs de notre équipe [50, 52], nous avons développé une approche pour analyser le logiciel du point

de vue de la qualité. Nous avons constaté que les approches manuelles où les informations sont présentées de façon brute aux utilisateurs sont difficilement utilisables à cause de l'énorme quantité de données. Nous avons aussi constaté que l'analyse automatique où des algorithmes traitent les données, sont difficilement utilisables à cause de la complexité et du flou qui peuvent exister dans l'interprétation des données. L'analyse est aussi largement dépendante du type d'application, de sa taille et de la gravité potentielle reliée à une défaillance, tous des facteurs difficiles à évaluer pour un algorithme automatique.

Notre solution de l'époque a utilisé avec succès la visualisation en tant qu'approche semi-automatique. La même stratégie sera utilisée pour améliorer un IDE existant en ajoutant une composante graphique importante à son interface. À l'aide de la visualisation, on réduit la complexité des informations en les représentant graphiquement et en déléguant une partie de la détection au système visuel humain qui est en mesure de percevoir certains phénomènes graphiques instantanément [40]. On facilite aussi la compréhension dans différentes vues en utilisant des représentations plus naturelles pour l'humain s'apparentant au milieu dans lequel il vit [74]. Ces représentations sont souvent beaucoup plus efficaces que des colonnes de chiffres représentant des informations brutes extraites sur les logiciels ou encore que le texte du programme lui-même [59, 93].

La visualisation élimine plusieurs problèmes liés à la complexité et à l'analyse des informations. Par contre, pour être en mesure d'apporter de l'aide aux utilisateurs, la représentation des informations doit être réussie. De plus, le logiciel est une entité abstraite et il n'est pas évident de lui trouver une représentation qui en fournit une image mentale adéquate [46]. Le désir de fournir de meilleurs outils pour la compréhension du logiciel amène de nouveaux défis. La visualisation doit être intelligente pour apporter des informations pertinentes et interprétables par la majorité. Il faut entre autres s'assurer de ne pas perdre les utilisateurs en les noyant dans trop d'informations ou en leur présentant celles-ci de façon contre-intuitive. Les grands défis se trouvent dans la sélection de la quantité d'informations à montrer dans une même vue et dans la navigation entre les différentes vues pour montrer le plus d'informations possibles. On doit aussi naviguer entre les différents niveaux de détails pour s'ajuster aux besoins de l'analyse. Durant les

transitions, on doit conserver une cohérence graphique et aussi une cohérence dans les informations présentées pour aider les utilisateurs à bien les interpréter.

La représentation de l'évolution du logiciel amène aussi ses défis. Dans un premier temps, la quantité d'informations à présenter sera encore plus considérable puisqu'elle est multipliée par le nombre de versions présentes. Les problèmes de gestion de l'espace à l'écran et de saturation du système visuel sont donc encore plus critiques. La cohérence lors des passages d'une version à l'autre sera encore une fois primordiale à une bonne assimilation des informations nécessaires aux analyses.

1.3 Proposition

Dans nos travaux antérieurs, nous avons étudié la qualité du logiciel en visualisant différentes métriques au niveau de granularité de la classe. Un exemple de ces résultats est présenté à la figure 1.2. Il était possible de suivre les valeurs des métriques à travers les différentes versions du logiciel. On agissait donc uniquement du côté de l'évaluation de la qualité des logiciels et de leurs classes (rétro-ingénierie). Les ajouts que nous proposons dans cette thèse s'étendront à de nouvelles informations sur le logiciel, comme les classes comportant des bogues, des informations statistiques sur le code lui-même et des informations sur le processus de développement incluant les développeurs les plus actifs ou encore les éléments les plus modifiés, des niveaux de granularité supplémentaires et la visualisation de l'évolution pour toutes ces vues. De plus, l'édition du logiciel pourra se faire à même la représentation du modèle graphique.

Après l'implantation des nouveaux concepts, on pourra, par exemple, construire une classe à un plus haut niveau et écrire une fonction en entrant dans le code textuel à l'aide d'un zoom. Ensuite, il sera possible de sortir de la classe pour observer les éléments qui lui sont rattachés et leurs aspects de qualité. Si par exemple, une de ces classes semble être liée à plusieurs autres dans le logiciel, qu'elle est très complexe et est souvent pointée du doigt quand un bogue survient dans le logiciel, on pourra plonger à l'intérieur de cette classe pour parcourir les méthodes causant principalement des problèmes, ou s'éloigner en mode paquetage pour voir quels programmeurs ont participé à ce module

pour ensuite les interroger sur le design en présence.

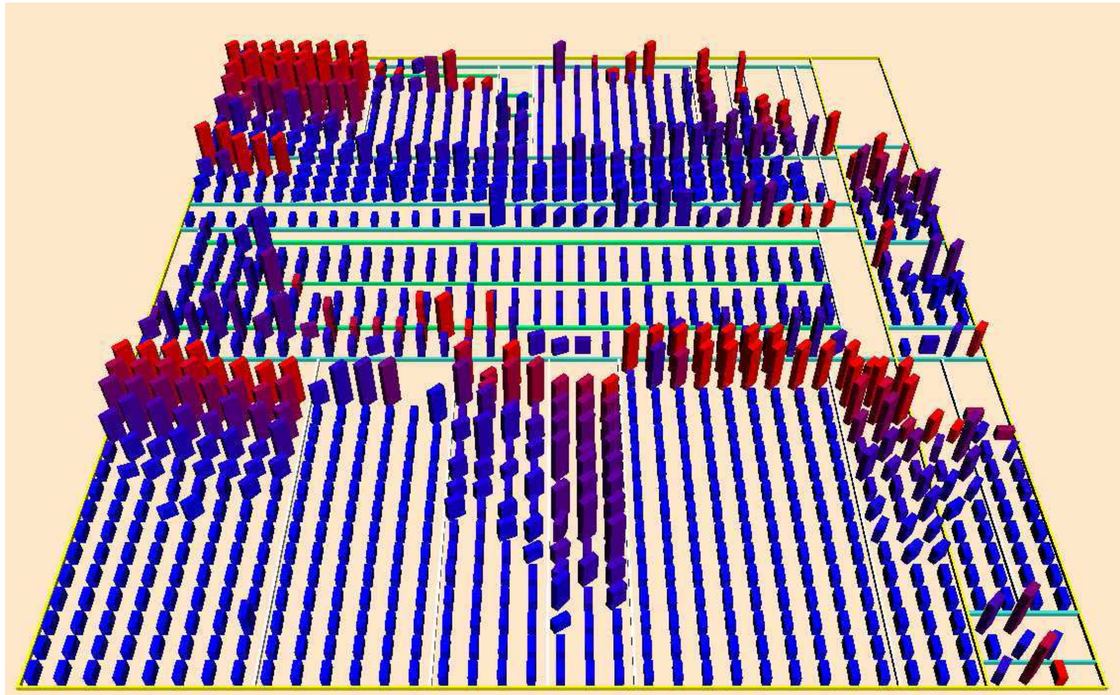

Figure 1.2 – Représentation des métriques de qualité pour le logiciel *PCGEN*. On y voit les informations sur pratiquement 1000 classes avec le couplage, la complexité et la cohésion associés respectivement à la couleur, la hauteur et la rotation des boîtes représentant les classes et les bordures représentant la hiérarchie des paquetages.

Le principe derrière notre approche est de fournir des informations aux utilisateurs en temps réel pour qu'ils soient en mesure de répondre à leurs interrogations lors du développement. Il peut s'agir de trouver la personne qui a travaillé sur une section du logiciel, de bogues affectant un élément, de l'endroit approprié pour faire une modification ou encore de l'évaluation de la qualité d'une partie du logiciel. L'analyse est supportée par des principes de visualisation et l'utilisation de vues multiples, mais cohérentes.

Les mécanismes de navigation se trouvent au coeur de la solution proposée. Il s'agit de donner aux développeurs une nouvelle façon d'interpréter et de concevoir le logiciel, et surtout une nouvelle façon de l'analyser et de faire sa maintenance tout en le conservant dans un état de bonne qualité. On peut aussi observer l'évolution du logiciel à travers toutes ses versions selon toutes les vues énumérées précédemment. Pour

que les utilisateurs puissent développer leur logiciel dans un environnement familier et aient toujours accès aux fonctionnalités déjà existantes, les principes de visualisation sont ajoutés dans un IDE connu et populaire (*Eclipse*). L'outil de visualisation (nommé *VERSO*) sera donc intégré dans *Eclipse*. Les deux interfaces communiquent entre elles pour permettre aux utilisateurs d'avoir une expérience transparente face à ce jumelage. La figure 1.3 montre le prototype de *VERSO* à l'oeuvre à l'intérieur d'*Eclipse*.

Auparavant, on séparait les activités de développement et d'analyse du logiciel. Nous proposons plutôt aux développeurs de suivre les aspects du logiciel eux-mêmes en leur montrant des informations facilement accessibles et interprétables. Ceci accélère le processus de développement en évitant les blocages causés par des questions non résolues, et évite de conserver des problèmes liés à la qualité devenant coûteux quand leur détection est tardive. Le fait d'être constamment témoin de la qualité d'un logiciel devrait rendre son traitement intimement lié au développement.

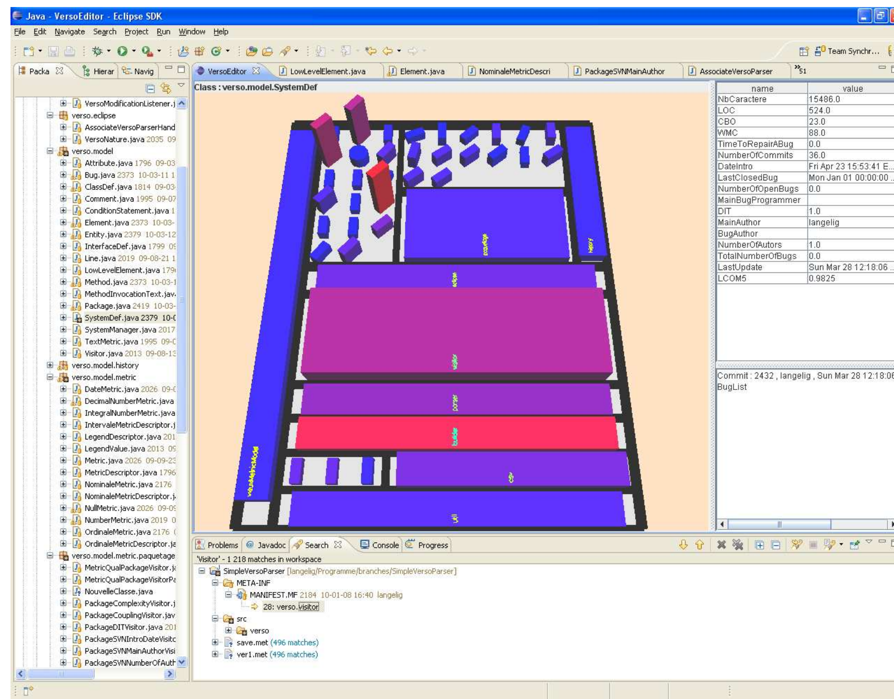

Figure 1.3 – *VERSO* intégré à l'IDE *Eclipse*. Les deux interfaces échangent des informations pour les présenter efficacement aux utilisateurs. Un seul outil permet donc de faire l'édition de code et la visualisation en parallèle.

1.4 Structure de cette thèse

La suite de cette thèse amènera des précisions sur les points abordés dans l'introduction et sera divisée en six chapitres. Nous allons, dans un premier temps, discuter de travaux connexes dans l'état de l'art au chapitre 2. Ensuite l'approche proposée sera l'objet du chapitre 3 et décrira en détail les choix faits lors des recherches et les motivations derrière ces choix. Le chapitre 4 portera sur la visualisation reliée à *VERSO* et le chapitre 5 discutera de l'intégration du prototype à l'intérieur de l'IDE, tant au point de vue technique que conceptuel. Le chapitre 6 discutera de l'évaluation de l'approche principalement à travers deux expériences menées durant les recherches. Finalement, le chapitre 7 conclura cette thèse en révisant nos réalisations et en présentant des pistes de recherche pour le futur.

CHAPITRE 2

ÉTAT DE L'ART

Le travail réalisé dans le cadre de cette thèse de doctorat couvre des domaines diversifiés, et la revue de littérature que nous présentons ici s'étend donc sur plusieurs domaines. Dans ce chapitre, nous discutons ces aspects et présentons différents outils les implantant. Ce chapitre sur l'état de l'art est divisé selon les sections suivantes : psychologie cognitive et visualisation, visualisation scientifique et visualisation d'informations, visualisation du logiciel, animation dans la visualisation, visualisation de l'évolution du logiciel, langages visuels et approches intégrées. Il n'existe pas de travaux à notre connaissance qui soient directement comparables au nôtre touchant à tous ces aspects.

2.1 Psychologie cognitive et perception

Toute forme de visualisation, y compris la visualisation du logiciel se doit d'être efficace pour donner accès à des informations qui n'étaient pas (ou difficilement) accessibles auparavant. Pour être efficaces, il faut que les utilisateurs soient en mesure de regarder rapidement les informations présentées et de les interpréter de façon précise. Les recherches en perception nous permettent de connaître les procédés qui génèrent les meilleures réponses de la part des utilisateurs. L'application de principes déjà reconnus et efficaces est aussi une façon de s'assurer de la pertinence des visualisations développées.

Dans ce domaine, plusieurs travaux de Healey et ses collègues [36–40] ont permis de faire progresser la recherche principalement dans le secteur de la visualisation d'informations. Ces chercheurs ont entre autres fait des expériences au niveau de la couleur, de la perspective dans un environnement en trois dimensions simulé, des formes et de la densité des éléments graphiques. Ces études faites avec plusieurs sujets ont démontré empiriquement l'importance de certains facteurs perceptuels dans la visualisation. On y apprend que le nombre de couleurs utilisables est limité lors de la représentation d'infor-

mations nominales [36] et que la perspective en trois dimensions ne diminue pas du tout ou moins qu'on ne le penserait la perception de différentes caractéristiques graphiques. En particulier, la hauteur réelle des éléments peut être adéquatement estimée même si le nombre de pixels varie avec l'éloignement de la caméra. Une image d'une partie des recherches de Healey est présentée à la figure 2.1. Elle montre une visualisation basée sur une carte géographique utilisant des bandes de différentes hauteurs et différentes couleurs pour représenter des variables reliées à une position sur la carte.

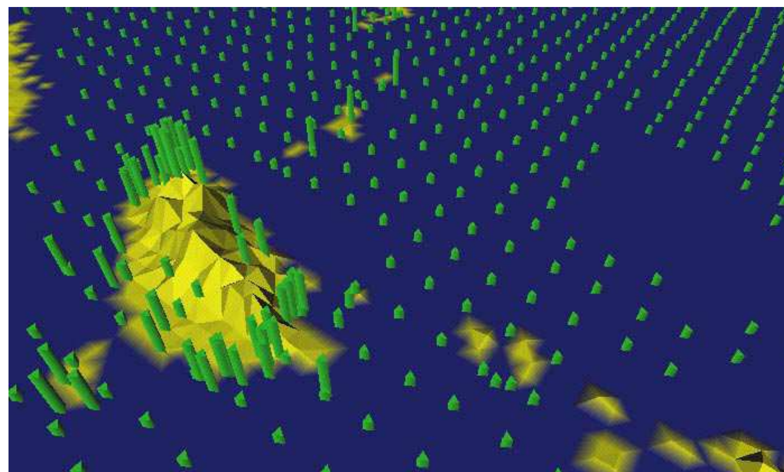

Figure 2.1 – Un extrait des travaux de Healey [40]. Il s'agit d'une visualisation d'une carte géographique avec des bandes représentant par une hauteur et une couleur des variables reliées à la position.

Dans une étude portant plus spécifiquement sur la couleur, Chuang et Weiskopf [13] ont développé une meilleure façon d'interpoler la couleur dans un dégradé ou encore dans une composition de couleurs résultant de la transparence des objets. Leur technique implique la préservation de la teinte de la couleur dominante pour empêcher qu'il y ait de nouvelles teintes qui apparaissent lors des compositions, comme c'est le cas avec une interpolation linéaire plus traditionnelle. D'un autre côté, Ware [94] souligne les difficultés d'utiliser la couleur pour représenter plus d'une dimension. Effectivement, la capacité des utilisateurs à évaluer avec précision des valeurs représentées autant par le principe additif que le principe soustractif des couleurs est mauvaise. Pour les visualisations appliquées sur des cartes et présentant deux variables supplémentaires, il propose

plutôt d'ajouter des textures simples (*Q-tons*) laissant passer l'information de la couleur sous-jacente. Dans le cas de l'utilisation de couleurs pour représenter une échelle de valeurs, comme c'est le cas dans la présente recherche, il serait probablement préférable de conserver la saturation et la brillance fixes en ne faisant varier que la teinte. Ceci représente une meilleure façon de réduire le biais quand cette technique est combinée à un arrière-plan de couleur fixe.

Rensink *et al.* [78–80] discutent beaucoup de l'importance de l'attention dans la perception des différences d'une image à l'autre. Ces recherches peuvent être utilisées pour améliorer les transitions dans les visualisations comportant de multiples vues et dans l'évolution où, plusieurs images sont montrées successivement aux utilisateurs et où les modifications doivent être analysées autant que l'image elle-même. On y apprend que l'ajout d'images intermédiaires ou d'éléments qui distraient l'observateur entre deux images empêche de percevoir des modifications, bien qu'elles soient évidentes sans ces perturbations. Dans notre cas, ces informations seront utiles lors de l'implantation des transitions entre différentes vues, différents niveaux de granularité et différentes versions du logiciel.

Shanmugasundaram *et al.* [89] donnent des résultats spécialisés à l'animation des transitions dans les graphes, mais qui sont tout aussi intéressants dans le contexte de la visualisation du logiciel. Des expériences avec des sujets ont démontré que les animations aident à l'interprétation des changements lors de la visualisation. De plus, ces travaux ont montré que les animations de courte durée sont aussi utiles que celles de plus longue durée. Elles peuvent donc être utilisées sans ralentir les analyses des utilisateurs. Ceci vient contrecarrer la critique classique de l'animation, où on prétend que le temps nécessaire à son déroulement la rend inefficace.

Dans un autre ordre d'idées, les travaux de Ramachandran et Hirstein [77] donnent des perspectives sur les réactions cognitives et émotives des gens devant les représentations graphiques. Tout en démontrant l'influence de certains phénomènes graphiques sur le cerveau humain, ces recherches nous permettent de cibler les points importants que devrait avoir une visualisation pour attirer l'attention des utilisateurs. Il est ainsi possible de savoir comment mettre l'emphase sur un phénomène ou encore comment éviter

de fausses impressions par le choix des caractéristiques graphiques.

2.2 Visualisation scientifique et visualisation d'informations

Le domaine de la visualisation est extrêmement vaste et les articles discutés dans cette section ne forment qu'un petit échantillon. Tous les articles ont été tirés des conférences *VIS* et *InfoVis*. Il existe des visualisations sur toutes sortes de domaines. Bien que les articles présentés dans cette section ne soient pas tous directement reliés à nos recherches, les avancements au niveau de la représentation graphique peuvent être utiles pour créer des visualisations spécifiques au logiciel.

La visualisation scientifique concerne les éléments ayant une représentation physique qu'il est possible de reproduire en image. Le but de ces représentations est d'être fidèle à l'élément original, bien que des couleurs artificielles soient parfois utilisées pour accentuer certains aspects. On pense entre autres à la visualisation médicale ou encore à la visualisation géologique.

D'un autre côté, la visualisation d'informations représente des données qui n'ont pas une apparence physique ; il n'est donc pas pertinent de les reproduire fidèlement. On peut penser à différentes variables sociologiques en fonction du salaire annuel ou encore aux fichiers sur un disque dur. La visualisation d'informations est d'ailleurs utilisée tous les jours dans différentes sphères de la société. La présentation d'un histogramme ou d'un diagramme circulaire dans le domaine de la finance sont des exemples classiques de visualisation d'informations. La visualisation du logiciel se retrouve dans cette catégorie de la visualisation d'informations puisque le code n'a pas de formes concrètes.

Du côté de la visualisation scientifique, il existe plusieurs travaux impliquant la médecine, dont voici quelques exemples [47, 62, 90]. Ces travaux ont comme point commun d'utiliser une représentation fidèle de certains tissus pour assister les professionnels de la santé. Ces recherches travaillent à partir d'images en provenance de digitalisateurs volumiques et améliorent les images obtenues en ajoutant des couleurs pertinentes et en simulant la troisième dimension. Les utilisateurs peuvent alors manipuler et observer les objets plus naturellement. Un exemple de ce type de visualisation est montré à

la figure 2.2. Le même principe est appliqué sur des images sismologiques [75]. Les phénomènes physiques comme la météo et les vents sont aussi souvent représentés en visualisation scientifique.

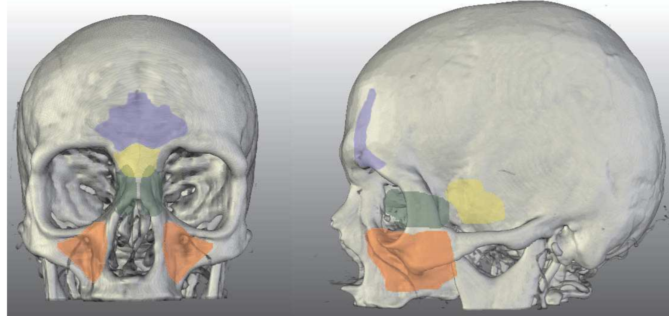

Figure 2.2 – Exemple de visualisation en médecine montrant les cavités sinusales [47]. La modélisation en trois dimensions et la couleur ajoutée permettent de mieux évaluer l'anatomie dans la région des sinus.

La visualisation d'informations couvre un éventail plus large de domaines que la visualisation scientifique puisque n'importe quelles informations abstraites peuvent être représentées. Étant donné qu'il n'existe pas de représentation inhérente à ces données, on se fie aux attributs quantifiables des éléments, et ce sont ces derniers qui sont représentés. C'est pourquoi plusieurs procédés de visualisation dans ce domaine sont directement calqués sur des types de visualisation mathématiques tels des graphes, histogrammes, diagrammes de Venn et diagrammes circulaires.

Par exemple, on présente souvent les intentions de vote à l'aide de cartes géographiques afin d'indiquer la répartition du vote par région. Par contre, les travaux de Draper et Riesenfeld [24] utilisent une technique circulaire pour décortiquer le vote selon d'autres variables en fonction des requêtes des utilisateurs (figure 2.3). Dörk *et al.* [23] donnent pour leur part une nouvelle façon de faire des recherches sur le web tout en multipliant les variables prises en compte à l'aide de la visualisation. On y montre des cartes géographiques en plus du texte lui-même dans une visualisation à multiples vues. Finalement, Jia *et al.* [43] discutent de différentes techniques pour la représentation de graphes. Ils emploient l'exemple de la représentation de réseaux sociaux pour montrer

l'importance de filtrer les graphes très volumineux.

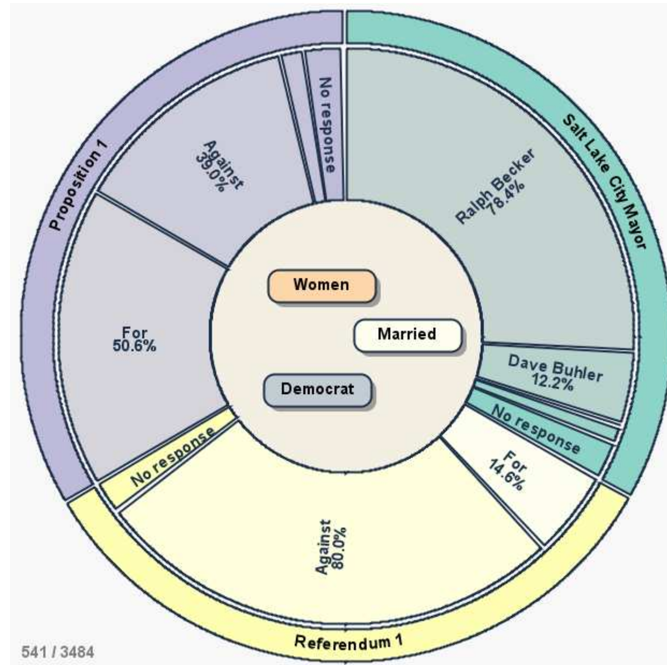

Figure 2.3 – Exemple de visualisation d'informations pour représenter les tendances du vote [24]. Il s'agit d'une représentation utilisant un système de requêtes pour répondre à des questions précises des utilisateurs sur un vote.

2.3 Visualisation du logiciel

La visualisation du logiciel est étroitement en lien avec les travaux de cette thèse. Notre objectif est de représenter le logiciel pour que des utilisateurs puissent efficacement accéder à des informations pour les aider à progresser rapidement dans leurs tâches ou à créer de meilleurs logiciels. Par contre, les travaux énumérés dans cette section se concentrent souvent sur des aspects précis du logiciel et ils ne sont pas intégrés dans le processus de développement, mais plutôt dans une phase d'analyse subséquente.

La visualisation du logiciel fait partie de la visualisation d'informations, puisque les logiciels sont des entités abstraites et la représentation binaire ou textuelle des programmes n'apporte pas de nouvelles informations intéressantes au processus de développement et d'analyse des logiciels. C'est pourquoi il faut créer des représentations

de toutes pièces pour afficher des informations pertinentes. Knight et Monroe [46] font d'ailleurs état de ces faits et ils discutent des apports de la réalité virtuelle sur la compréhension du logiciel. Plusieurs auteurs ont développé des outils de visualisation pour aider les programmeurs et les analystes à mieux comprendre le logiciel ou à interpréter les analyses faites sur celui-ci. On visualise tantôt les lignes de code elles-mêmes, tantôt des métriques sur des composants plus grands comme les méthodes ou les classes. L'analyse peut se faire de façon statique (code) ou dynamique (trace d'exécution). Bien que la visualisation dynamique soit plus difficilement généralisable et que la visualisation statique ne représente pas toujours l'utilisation réelle d'un programme, il reste que les deux façons de procéder amènent des informations intéressantes et parfois complémentaires.

Avec l'outil *SEESOFT* [2] (voir figure 2.4), Ball et Eick ont été parmi les premiers à représenter le logiciel, et ce, principalement au niveau de la ligne de code, en utilisant des lignes de couleurs. Ces lignes sont arrangées dans des rectangles représentant les fichiers. Ils sont en mesure d'afficher jusqu'à 50 000 lignes en même temps. Marcus *et al.* [60], pour leur part, ont transformé ces lignes de couleurs en bandes de couleurs présentées en trois dimensions. Cette fois-ci, les bandes représentent plutôt des méthodes où la hauteur et la couleur sont associées à des valeurs calculées sur chaque méthode. Ils montrent aussi une vue plus traditionnelle en deux dimensions, qui correspond à la vue aérienne de leur représentation en trois dimensions. Lanza et Ducasse [54] utilisent un procédé qu'ils nomment *Polymetric Views*, où les éléments sont représentés par des rectangles. Jusqu'à cinq valeurs peuvent être associées à des caractéristiques graphiques d'un rectangle soient la couleur, la hauteur, la largeur et la position dans les deux dimensions. Selon la tâche d'analyse, les éléments peuvent être triés ou encore on peut voir la structure hiérarchique à l'aide de liens sous forme de lignes entre les rectangles. Cette recherche est basée sur le métamodèle *MOOSE* [69] qui a d'ailleurs engendré toute une panoplie d'outils de visualisation appliqués à la rétro-ingénierie et à la présentation des métriques. Certaines approches se sont aussi intéressées à l'évolution du logiciel et aux informations tirées des logiciels de contrôle de versions. Lienhard *et al.* [56] proposent un outil d'aide à la rédaction des tests unitaires. Une fenêtre de sélection, où les méthodes sont arrangées sous forme d'arbre montrant le fil d'exécution, permet de choisir

une méthode pertinente. Une fois la méthode sélectionnée, un graphe des types utilisés dans les appels et les sous-appels de cette méthode est créé. Ce graphe est appelé *Test Blueprint* et, avec l'aide d'informations fournies sur demande, il assiste les utilisateurs dans la création de tests.

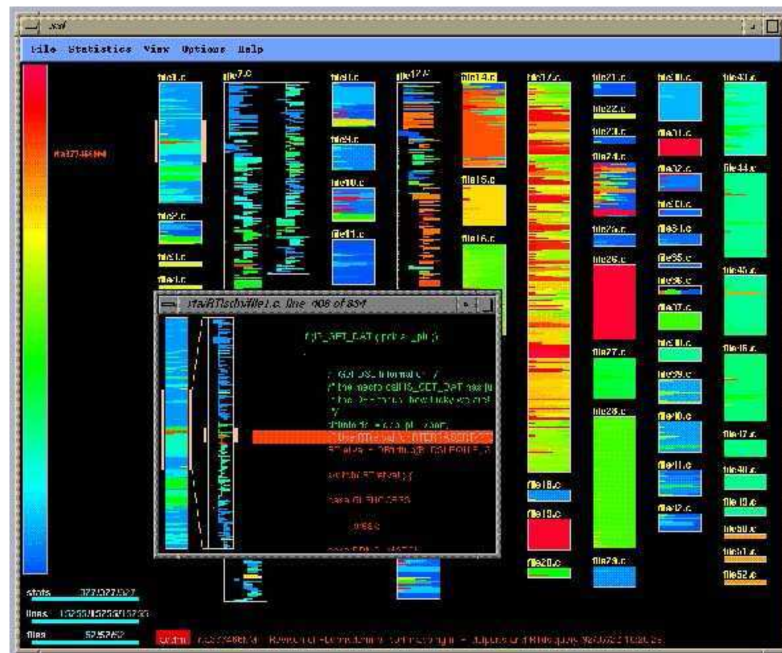

Figure 2.4 – Visualisation des lignes de code à l'aide de bandes de couleurs avec l'outil *SEESOFT* [2].

Termeer *et al.* [91] sont partis d'une visualisation traditionnelle du logiciel, c'est-à-dire le diagramme de classes UML, et l'ont améliorée grâce à la visualisation en trois dimensions et grâce à l'utilisation des métriques de classes. En fait, ils présentent le diagramme de classes UML sur un plan et ajoutent des cubes émergeant de ce plan. Chaque entité représente une métrique et la transparence permet de lire le diagramme UML sous-jacent. Rilling et Mudur [83] ont utilisé le *program slicing* [1] et une représentation de sphères en trois dimensions reliées entre elles pour visualiser le logiciel. La couleur permet de représenter une information supplémentaire. Le *program slicing* permet de réduire la quantité des informations présentées et ainsi de réduire les possibles

[1]Ensemble des instructions affectant une valeur dans un programme à un point donné.

occultations dues à la troisième dimension.

Ensuite, Wettel et Lanza [95] ont utilisé la métaphore de la ville pour représenter le logiciel, et plus particulièrement la facilité de maintenance du logiciel. Dans leur représentation, les classes sont des édifices et les lotissements sont des paquetages. Ils visualisent les métriques à l'aide de la largeur, de la hauteur et de la couleur des édifices(voir la figure 2.5). Les auteurs ont par la suite étendu leur approche à l'évolution des logiciels [96]. Panas *et al.* [74] avaient énoncé une idée semblable sur la métaphore de la ville, en présentant des travaux où l'importance du réalisme immerge mieux les utilisateurs lors de ses analyses. Le principe des métaphores permet aux utilisateurs de transférer des connaissances sur un sujet qu'ils maîtrisent bien (par exemple la ville) vers un sujet qui leur est nouveau (par exemple la qualité du logiciel). L'influence de la métaphore sur l'apprentissage et l'assimilation d'informations est étudiée par Mason [61].

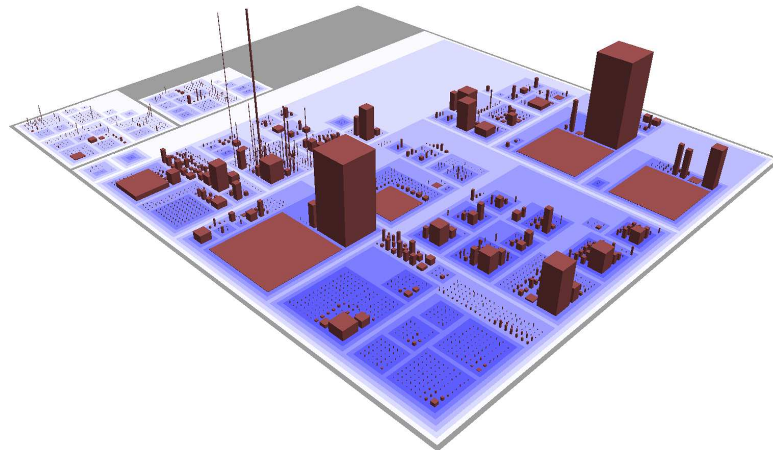

Figure 2.5 – Représentation de ArgoUML avec *Code City* [95]. Dans cette visualisation, la hauteur et la taille de la base des éléments représentent des métriques. Ces éléments sont placés selon la hiérarchie de paquetages. La métaphore de la ville est utilisée reliant les éléments représentés à des édifices.

2.4 Animation et visualisation

Pour une surface d'affichage équivalente, l'animation permet d'afficher plus d'informations qu'une visualisation statique. Différentes images successives permettent de

suivre les modifications du modèle sous-jacent. De plus, l'animation est souvent utilisée pour introduire une quatrième dimension, c'est-à-dire le temps. Non seulement l'animation permet de montrer plus d'informations, mais souvent les transitions entre les images clés apportent des informations pertinentes en elles-mêmes, ou encore aident à interpréter la prochaine image clé sans saturer le système visuel humain par des réarrangements instantanés [89]. Les principes développés dans les différents domaines, même autres que la visualisation du logiciel, permettent de comparer les approches et de choisir celle qui est la mieux adaptée au logiciel. L'animation est utilisée dans *VERSO*.

Dans un premier temps, Shanmugasundaram *et al.* [89] mentionnent la très grande efficacité de l'animation dans la perception des changements. Leurs essais surtout sur des transformations de graphes et d'arbres comparent leurs animations avec des images saccadées (changements instantanés). Leurs résultats montrent que les utilisateurs font jusqu'à deux fois moins d'erreurs avec l'animation. Robertson *et al.* [84] prétendent au contraire que l'animation dans l'observation de tendances est moins bonne lorsque comparée à d'autres techniques. Cette différence d'opinion s'explique par le fait que dans le premier cas, on parle d'images intermédiaires alors que dans le second, on parle plutôt d'images indépendantes montrées sans traitement spécifique les unes après les autres.

Nguyen et Huang [68] ont utilisé l'animation pour représenter la navigation dans des arbres. Dans cette approche, les utilisateurs sélectionnent un noeud qui devient progressivement le centre d'intérêt à l'aide d'images intermédiaires. Pour ce faire, on descend dans la hiérarchie vers le noeud sélectionné en représentant le sous-arbre de chaque noeud sur le chemin et en éliminant les autres branches de la représenation. Blah *et al.* [5] représentent un *Treemap* [44] avec l'aide d'une troisième dimension pour donner du relief à la visualisation. Ils font un zoom graduel sur une partie sélectionnée par les utilisateurs pour leur permettre de mieux observer les détails d'une région du *Treemap*. Fekete et Plaisant [27] travaillent aussi avec le *Treemap* pour représenter de grands ensembles de données. Ils utilisent l'animation pour voir fluctuer les valeurs des éléments dans l'arbre. Il est possible que l'espace occupé par les items soit modifié, mais ils ne traitent pas l'ajout ni le retrait d'éléments dans leur approche. Les déplacements

et les modifications des caractéristiques comme la couleur ou la grosseur des noeuds s'opèrent en deux étapes pour éviter une confusion découlant de l'influence croisée des modifications. North [70] a développé une technique pour représenter les arbres avec des additions et des suppressions de noeuds durant les transitions. Cette recherche tente de minimiser le plus possible le déplacement des noeuds entre chaque transition. Finalement, Cornelissen *et al.* [16] appliquent directement leur approche basée sur l'animation à la représentation de l'exécution d'un programme. À partir d'un diagramme circulaire et de liens représentés au centre du cercle à l'aide de *splines* réduisant l'occultation entre les liens, il est possible de suivre les appels de méthodes pour de longues traces d'exécution (voir exemple sur la figure 2.6).

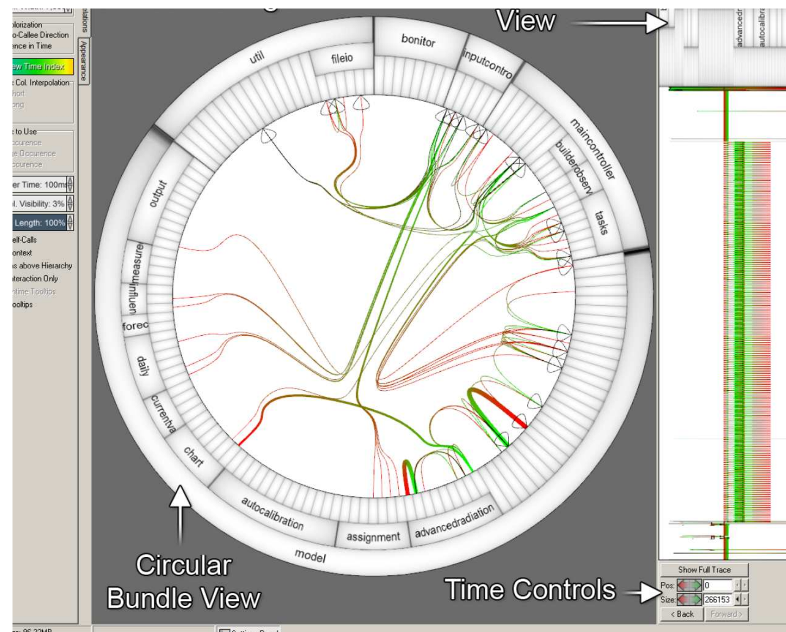

Figure 2.6 – Une vue d'*EXTRAVIS* [16]. *EXTRAVIS* est un outil pour la visualisation de grande trace d'exécution. Les fichiers sont affichés autour du cercle alors que les appels sont affichés dans son centre en utilisant des *splines* et des regroupements hiérarchiques pour diminuer l'occultation.

2.5 Visualisation de l'évolution du logiciel

L'évolution du logiciel est une partie de l'analyse qui se prête particulièrement bien à la visualisation puisqu'il faut rapidement trouver des informations pertinentes parmi la grande quantité d'informations. *VERSO* traite d'ailleurs de l'évolution du logiciel tant par la visualisation de plusieurs versions que dans un contexte affichant des métriques sur l'historique des versions.

Les approches automatiques d'analyse du logiciel sont en mesure de considérer de grandes quantités de données rapidement. C'est pourquoi ce genre d'approche s'est intéressé à l'évolution du logiciel, où il est possible d'extraire des informations sur le processus de développement d'un logiciel et sur sa maintenance. Les approches visuelles n'ont pas tardé à emboîter le pas pour pallier les manques des analyses entièrement manuelles ou entièrement automatiques. Curieusement, peu d'entre elles ont su exploiter l'animation discutée dans la sous-section précédente.

Une façon classique de représenter les informations des logiciels et de leur évolution est d'utiliser une matrice où l'axe des Y représente les entités et l'axe des X le temps ou encore les différentes versions. Aux intersections ligne-colonne, on place un point de couleur ou un élément graphique plus complexe possédant plusieurs caractéristiques. Un bon exemple de la première technique est présenté par Wu *et al.* [97] alors qu'un exemple de la deuxième peut être retrouvé à l'intérieur des travaux de Lanza et Ducasse [55]. Une comparaison entre les deux approches est présentée à la figure 2.7. Plusieurs travaux utilisent une ligne du temps [19, 57, 64] avec des configurations semblables aux matrices.

D'autres approches sont utilisées pour représenter l'évolution du logiciel. Dans ce cas, on agrège plutôt les informations dans une image unique et on présente les informations sur l'évolution elle-même plutôt que de la présenter à travers les transitions. Par exemple, Fischer et Gall [28] étudient les changements simultanés à l'aide de graphes suivant le principe d'attraction et de répulsion. Plus les éléments sont près, plus ils ont tendance à être modifiés ensemble. Pinzger *et al.* [76] utilisent des diagrammes de Kiviat en représentant les versions par de nouvelles couches sur le diagramme (voir figure 2.8). D'Ambros *et al.* [20] utilisent des figures fractales pour représenter la contribution des

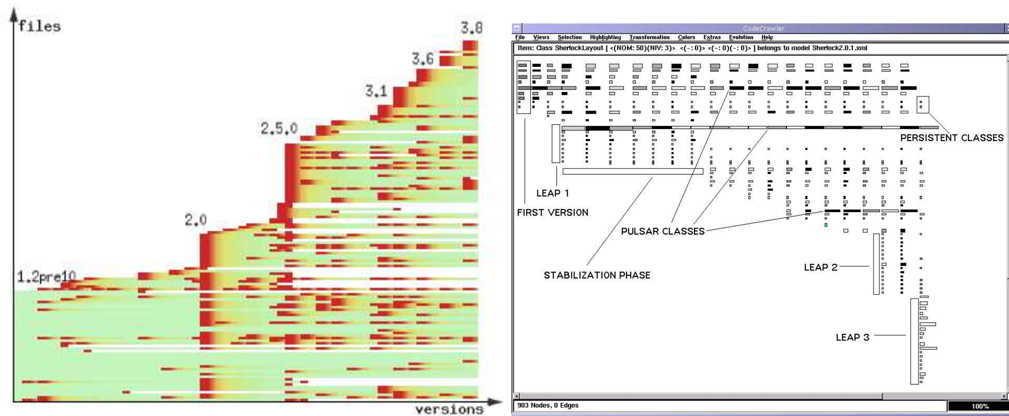

Figure 2.7 – Visualisation de l'évolution du logiciel à l'aide d'une matrice. Les objets peuvent être aussi simples que des pixels [97] (gauche) ou encore être des symboles plus complexes montrant plusieurs métriques à la fois [55] (droite).

auteurs pour les différents fichiers. Plus un auteur participe à un fichier, plus sa bande de couleur sera importante.

Lungu *et al.* [58] présentent une approche basée sur une application web et composée de plusieurs visualisations simples incluant des tableaux, des lignes de temps et des graphes. Les informations observées sont en relation avec des répertoires reliés à un système de contrôle de versions. Plutôt que d'observer un projet unique, ils choisissent d'observer tout l'*écosystème* d'un projet regroupant plusieurs répertoires. Greevy *et al.* [34] présentent une visualisation simple pour étudier l'évolution du logiciel, mais du point de vue d'entités plus abstraites que la classe (*features*). Ils utilisent des boîtes englobantes pour représenter les *features* et ajoutent les classes comme des petits carrés de couleur selon une caractérisation des entités. Plus spécifiquement pour l'évolution, des vues historiques et des vues d'additions montrent les classes retirées ou ajoutées sur une période de temps. De plus, un graphe d'évolution montre les tendances des propriétés pour une entité calculées en fonction des classes présentes dans l'entité. Ils évaluent leur approche à l'aide d'études de cas impliquant l'évolution de différentes branches pour le projet *SmallWiki*.

Kuhn *et al.* [48] proposent pour leur part une méthode de placement des éléments pour que les utilisateurs se créent une carte mentale du programme. Ils utilisent la si-

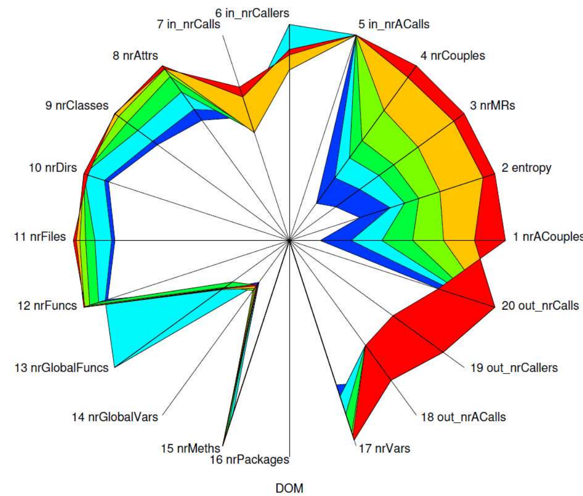

Figure 2.8 – Représentation de l'évolution de métriques à l'aide d'un diagramme de Kiviat [76]. Les métriques sont distribuées de façon équidistante autour du cercle et chaque couche de couleur représente une version.

milarité lexicale pour regrouper les classes entre elles et des techniques de réduction de dimensions pour arriver à les positionner en deux dimensions. L'utilisation de la proximité lexicale a pour but de réduire les écarts entre les placements des différentes versions. Selon les auteurs, cette proximité lexicale donne de meilleurs résultats pour la stabilité que la structure des programmes. Pour ajouter à cette stabilité, les résultats des placements aux étapes précédentes sont pris en compte lors du calcul des positions, résultant en des déplacements lents et progressifs.

Par ailleurs, quelques approches utilisent l'animation comme nous entendons le faire dans la partie traitant de l'évolution de cette thèse. Dans un premier temps, D'Ambros et Lanza [18] présentent une approche appelée *Evolution Radar* qui montre aussi les changements simultanés. Un fichier est placé au centre de la visualisation et pour une période donnée, on peut voir quels fichiers ont changé en même temps selon leur distance du centre du cercle (voir figure 2.9). Ces images peuvent être générées et montrées en séquence. Ensuite, Collberg *et al.* [14] ont développé une manière de visualiser les graphes de flux de contrôle et les graphes d'appels. Pour conserver une cohérence entre les différentes représentations, les versions de chaque noeud sont reliées par des liens invisibles,

ce qui réduit leur mouvement par effet de bord de l'algorithme de placement basé sur des ressorts. Beyer et Hassan [4] utilisent le principe du *story-board* pour présenter chaque étape de la visualisation. Ils utilisent aussi le principe de l'attraction et de la répulsion pour placer les éléments dans un univers en deux dimensions. Cette approche est utilisée dans le cadre de la représentation des informations extraites des systèmes de contrôle de versions.

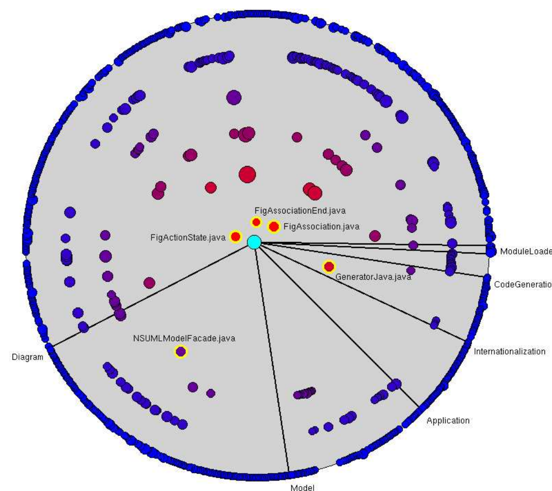

Figure 2.9 – Exemple de l'*Evolution Radar* [18]. Avec cette visualisation, un fichier est placé au centre du cercle et les autres fichiers du programme se rapprochent de ce dernier s'ils font l'objet d'un même *commit*.

Enfin, Ogawa et Ma [71] utilisent aussi l'animation pour représenter l'évolution du logiciel, plus particulièrement en s'intéressant aux systèmes de contrôle de versions. Leur approche met le programmeur (de logiciels à source ouverte) au centre de la visualisation avec les fichiers modifiés qui gravitent autour de lui. La métaphore utilisée est celle de l'astronomie où les fichiers sont plus gros quand ils sont modifiés plus souvent et dont la brillance à l'écran représente le temps passé depuis la dernière modification. Les auteurs insistent sur l'importance de l'esthétique et sur une durée optimale des petits films représentant l'évolution du logiciel. Ils veulent être en mesure de montrer ces films à un public qui n'est pas nécessairement analyste en informatique et discutent même de l'esthétique d'une visualisation organique pour des utilisateurs qui ne connaissent pas

les informations sous-jacentes. La génération des films est très longue et ne peut donc pas être utilisée pour une analyse immédiate et continue d'un projet, mais plutôt pour résumer les points culminants de l'évolution d'un logiciel. Une image illustrant leur approche est présentée à la figure 2.10.

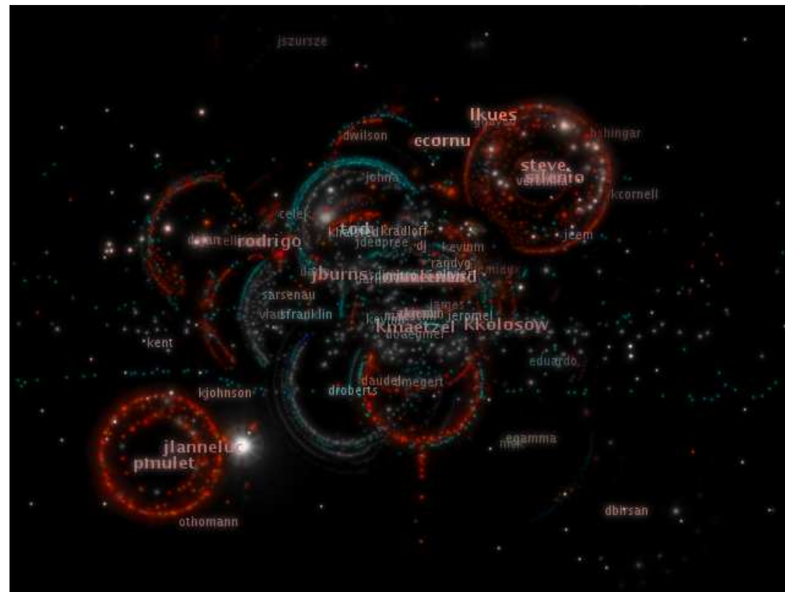

Figure 2.10 – Une image de l'évolution d'*Eclipse* telle que représentée dans le logiciel *code swarm* [71] où les fichiers gravitent autour de leurs auteurs.

2.6 Langages visuels

Les langages visuels utilisent beaucoup les indices graphiques pour montrer des programmes et aussi pour créer et assembler les instructions des programmes. Puisque l'approche présentée dans cette thèse a été conçue pour montrer les informations sur le logiciel pour l'analyser en même temps que les programmes sont créés, et pour permettre aux utilisateurs de commencer à construire le programme directement à partir des vues graphiques, il est pertinent de s'intéresser à cette forme de programmation qui fait usage de réponses directes aux utilisateurs.

Ces langages sont une forme différente de visualisation qui laisse les utilisateurs construire leur programme à l'aide de composantes graphiques. Ce principe ressemble

un peu à un casse-tête où l'on crée les logiciels à l'aide du glisser-déposer de la souris. Ces langages visuels utilisent aussi beaucoup d'images animées pour présenter les résultats de la programmation. Par contre, ils ne sont souvent pas de la visualisation à proprement parler et visent une clientèle jeune ou adolescente dans le but avoué d'augmenter leur intérêt envers l'informatique et de faciliter leur apprentissage [15]. Ces langages sont en général beaucoup moins puissants que les langages de programmation classiques et sont limités à la construction de petits jeux ou à des simulations simples.

Alice [15] est un environnement de développement qui utilise les images en trois dimensions pour attirer les jeunes vers la programmation et leur apprendre les principes de base. La scène principale et les différents objets graphiques sont déjà disponibles, et les utilisateurs appliquent des principes de programmation pour les animer. Cette programmation comprend des principes de base comme les affectations, les appels de fonction et les boucles, mais ils sont représentés par des boîtes englobantes pour mieux interpréter la portée des instructions. De plus, la création de programmes est aidée par plusieurs menus déroulants. *Agent Sheets* [81] est un programme semblable qui se base sur les agents. En fait, les agents sont des objets avec des propriétés, des méthodes, une représentation graphique et un fil d'exécution bien à eux. Les graphiques ne sont pas l'emphase principale dans ce cas, mais l'éditeur semble laisser plus de latitude aux programmeurs pour produire des programmes plus variés. Les auteurs ont plus récemment proposé une approche pour aider les jeunes programmeurs à passer du 2D au 3D [42]. *Scratch* [82] est un autre langage visuel destiné aux enfants et aux adolescents. Encore une fois, les utilisateurs sont invités à faire des dessins et à leur donner vie à l'aide de la programmation, où l'on assemble des instructions avec la souris. Ce ne sont que trois exemples d'une panoplie d'outils qui peuvent être utilisés par les débutants (souvent jeunes) dans l'apprentissage de la programmation.

2.7 Approches intégrées

Les approches intégrées, c'est-à-dire celles qui permettent à la fois de créer des programmes, de les analyser pour les comprendre et d'évaluer leur qualité, comme celle

présentée dans cette thèse, sont plutôt rares puisque l'analyse et la conception du logiciel sont encore aujourd'hui des activités considérées comme distinctes et faites par des intervenants différents. Par contre, il existe des interfaces usager évoluées qui se détachent de plus en plus de l'éditeur de texte traditionnel. Avec les architectures complètement découplées comme celle d'*Eclipse*, les développeurs sont encouragés à modifier les IDE à l'aide de plugiciels et peuvent maintenant le faire avec une flexibilité accrue. Ces plugiciels ont par contre tendance à se concentrer sur l'optimisation des activités d'écriture de code et ne sont que peu présents dans le secteur de l'analyse. Un bon exemple de ce genre d'innovation au niveau de l'interface graphique se trouve dans les travaux de Bragdon *et al.* [7]. Les auteurs présentent une interface réinventée basée sur des bulles de texte contenant des méthodes (de programmes *Java*) plutôt que des classes ou des fichiers, comme c'est le cas avec les éditeurs traditionnels. L'application présente un grand espace de travail dans lequel on navigue plutôt que d'être contraint dans le modèle de la page déroulante. *Code Bubbles* peut dériver automatiquement une série d'appels entre deux méthodes et afficher les méthodes intermédiaires selon les requêtes des utilisateurs.

Robillard et Murphy [85] présentent un outil d'aide intégré lui aussi à *Eclipse*. Ils présentent des arbres de points d'intérêt (*concerns*) qui permettent aux utilisateurs de naviguer à travers les éléments reliés à ces points d'intérêt. Les représentations utilisées consistent en des *Treeviews* ou des listes d'éléments, et on peut accéder au code des éléments à l'aide d'hyperliens. Cette approche permet aux utilisateurs de mieux comprendre les points d'intérêt qui peuvent être éparpillés dans le logiciel et ainsi leur permettre de progresser dans leurs tâches. Fritz et Murphy [29] utilisent pour leur part des fragments d'informations pour aider les programmeurs à répondre à des questions typiques durant le développement de logiciel. La composition des fragments d'informations est présentée dans des vues intégrées à *Eclipse*, composées encore une fois d'arbres et de listes. Le tri des fragments d'informations peut se faire selon les besoins des utilisateurs. La figure 2.11 présente un exemple de cet outil avec une vue centrée sur le code. Par la suite, Dagenais *et al.* [17] discutent des difficultés des nouveaux arrivants dans un projet, mais ne proposent pas d'outil pour les aider à mieux comprendre le code. Ils mentionnent par contre dans leur conclusion que différentes vues sur le code pourraient aider les nou-

veaux programmeurs à répondre à leurs questions sur des logiciels inconnus.

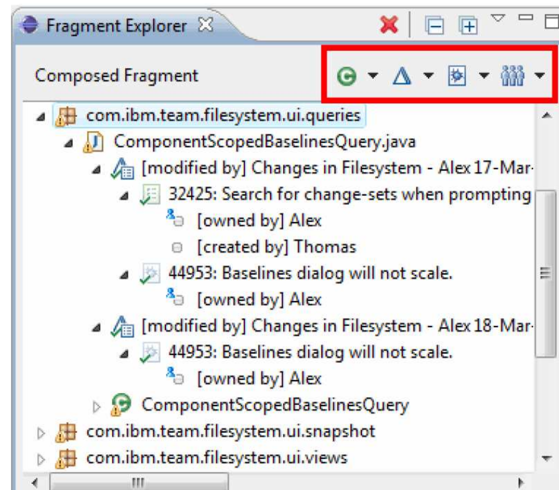

Figure 2.11 – Vue de fragments d'informations (Fritz *et al.*) Cette vue [29] présentée à l'intérieur d'un plugiciel d'*Eclipse* nous donne des informations centrées sur le code.

Kuhn *et al.* [49] ont aussi ajouté leur approche de carte du logiciel (introduite dans la section 2.5 de ce chapitre) dans l'environnement de développement *Eclipse*. Les liens avec l'IDE permettent de présenter les informations dans leur outil graphique. Une expérience menée auprès de sujets montre que l'outil est particulièrement utile pour montrer les résultats de recherches dans *Eclipse* et que la réponse immédiate de la visualisation à même l'IDE tout au long de la programmation est intéressante pour les utilisateurs. Röthlisberger *et al.* [87] montrent des informations sur les traces d'exécution des programmes à l'intérieur d'*Eclipse*. Des fenêtres apparaissent pour montrer des informations sur les types réels lors des appels de méthodes ou pour montrer des informations statiques par exemple, le nombre d'appels ou la quantité de mémoire utilisée. Les règles verticales, avec de petites bandes de couleurs, résument les informations. Des symboles sur l'explorateur de paquetages présentent les informations de plus haut niveau sur les traces d'exécution. De plus, tout comme avec *VERSO*, les métriques sont calculées par un plugiciel d'*Eclipse* et présentées dans l'éditeur dès que des résultats partiels sont disponibles.

À notre connaissance, il n'existe pas d'approches intégrant à la fois des informa-

tions pour l'analyse et la prise de décision en programmation, un contrôle de la qualité avec des vues multiples et la création du logiciel à partir de visualisation. Pacione *et al.* [73] présentent une approche intégrée regroupant comme nous plusieurs aspects du logiciel en fonction de différents niveaux de granularité. Ils font une bonne description de l'importance des multiples vues intégrées à un même outil, mais ils se concentrent uniquement sur des représentations graphiques simples déjà connues. Il existe certainement plusieurs outils qui inspectent le logiciel à l'aide de la rétro-ingénierie. Par contre, ils se concentrent uniquement sur un aspect du logiciel ou encore ils s'intéressent à plusieurs aspects à travers des représentations différentes dans des fenêtres séparées. Les langages visuels, pour leur part, amènent des notions de construction de logiciels par des outils graphiques. Bien que ces notions soient intéressantes pour l'apprentissage, elles agissent à un bas niveau alors que nous désirons agir sur le squelette[2] du code à un niveau de granularité plus élevé. Ces langages réduisent aussi les possibilités de créer des programmes complexes et ne s'intéressent pas à la qualité des programmes obtenus. L'intégration des composantes de l'évaluation de la qualité à travers différentes vues et différents niveaux de qualité de l'évolution du logiciel, jumelée à la possibilité de programmer directement dans l'environnement de l'évaluation de la qualité est un nouveau concept permettant la construction de programmes plus facilement maintenables, et ce, plus rapidement.

[2]Un squelette de code est l'équivalent d'un plan pour le logiciel. Par exemple, en *Java* le squelette de code comprend les classes et les signatures de méthodes. Le code sous forme de squelette demande des modifications avant de pouvoir être exécuté.

CHAPITRE 3

APPROCHE

Les environnements de développement se sont concentrés par le passé sur des aides à la programmation de bas niveau. L'efficacité de ces outils dans l'aide directe à la programmation a explosé au cours des dernières années et il est maintenant beaucoup plus facile et convivial, même pour un néophyte, de développer des programmes. Les outils d'aide à la programmation sont en général très bien intégrés dans les environnements de développement modernes. Il s'agit même d'un critère capital dans le choix d'un outil. Par exemple, les outils de débogage permettent aux utilisateurs d'exécuter un programme pas à pas directement dans l'environnement de développement tout en leur permettant d'identifier l'endroit exact où une exception ou une erreur a été levée, et en les laissant faire des modifications sans interrompre le processus. Des outils de recherche sophistiqués ainsi que des outils d'analyse de dépendances sont aussi accessibles et intégrés dans l'interface graphique de l'éditeur de code.

Par contre, ce n'est que relativement récemment que se sont développés des outils spécifiques plus conviviaux pour l'analyse des logiciels. L'importance de l'analyse et de la conception, et non pas seulement de la correction des programmes, s'est développée avec l'augmentation de la complexité des logiciels et du volume et de l'étalement des équipes travaillant sur les projets. Le coût augmentant aussi rapidement que la complexité, il est normal que les ingénieurs logiciels s'intéressent plus à ces problèmes. Il existe plusieurs outils automatiques qui permettent d'analyser sous plusieurs aspects un produit à travers des résultats chiffrés, mais les outils qui nous intéressent le plus dans cette thèse sont ceux qui utilisent la visualisation pour présenter des informations aux utilisateurs. Ces informations les aident à répondre à une panoplie de questions sur leur tâche de développement en fournissant les informations recueillies par les systèmes de contrôle de versions et l'évolution d'un logiciel ou des données qui les aide à analyser la qualité du logiciel. Contrairement aux outils d'aide à la programmation qui eux sont parfaitement intégrés dans l'environnement de développement, ces outils sont souvent dé-

veloppés pour être utilisés à l'extérieur des environnements de développement. En effet, les activités de conception, de développement et d'analyse sont souvent vues, et en particulier par les ingénieurs logiciels, comme des étapes qu'il faut séparer pour s'assurer du bon fonctionnement d'un projet [86]. Cette vision amène par contre les programmeurs à se désintéresser des étapes d'analyse et même de la compréhension du code puisque l'activité ne semble pas les concerner ou encore leur opinion n'est pas considérée lors des prises de décisions.

Nous sommes plutôt d'avis que l'implication des programmeurs et leur connaissance des logiciels leur permettront de progresser plus rapidement en évitant les blocages pour cause de manque d'informations. De plus, la connaissance des impacts des principes de qualité au niveau de l'écriture même du code amènera des logiciels de meilleure qualité à moindre coût. En effet, plus une erreur de programmation est corrigée tard dans le processus de création d'un logiciel, plus elle sera coûteuse à régler [3]. L'inclusion d'outils d'analyse et de compréhension du code est donc pertinente dans un environnement de développement pour mettre l'emphase sur cet aspect des logiciels auprès des développeurs et rapprocher les étapes d'analyse des étapes de développement. Nous sommes convaincus que l'obtention en direct d'informations pertinentes au développement et sur la qualité des logiciels permettra d'accélérer le développement et de réduire l'accumulation de petites erreurs ou mauvaises pratiques rendant difficile la marche arrière une fois la phase d'analyse entamée.

L'approche implique donc de donner accès facilement aux informations sur plusieurs niveaux de granularité du logiciel, dans plusieurs contextes et sur plusieurs versions. L'accès à ces informations est fourni par une analyse des différents aspects du logiciel et est représenté par une visualisation comportant de multiples vues intégrées à même l'IDE. Une recherche typique d'informations par des utilisateurs voulant répondre à des questions sur le logiciel nécessite de naviguer à travers les différentes vues. Pour réduire l'effort requis pour trouver les informations, des représentations visuelles basées sur les principes de perception sont utilisées et les déplacements entre les différentes vues sont pensés pour réduire l'effort cognitif par des principes de cohérence. La recherche d'informations peut ensuite être considérée comme une série d'inspections de ces vues

et de déplacements entre ces dernières. Le but de notre approche est de réduire l'effort total consacré à une recherche d'informations. Un résumé des vues accessibles et donc des informations accessibles est disponible dans la figure 3.1. Une recherche typique d'informations peut être schématisée par une série de déplacements et d'observations dans les vues de ce cube.

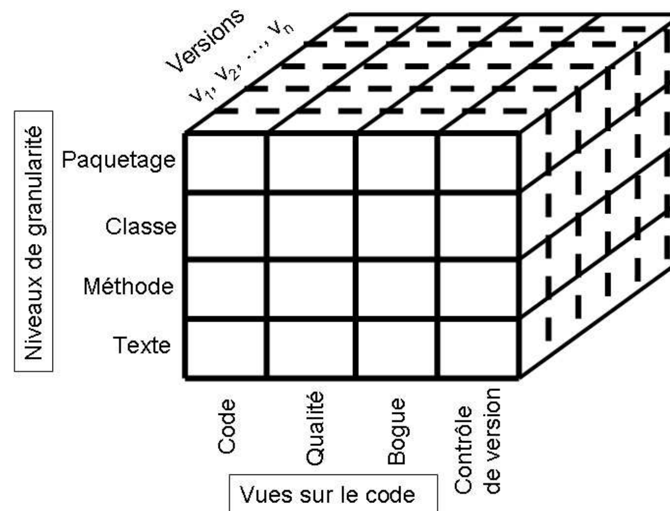

Figure 3.1 – Matrice des différentes vues atteignables selon les trois dimensions. Sur la dimension verticale, on place les différents niveaux de granularité. Sur la dimension horizontale, on trouve les différents aspects. L'évolution du logiciel s'intègre comme une troisième dimension (la profondeur) dans la matrice des visualisations possibles. Les visualisations se trouvent dans les cellules et les déplacements d'une vue à l'autre ne peuvent se faire que d'une position à ses voisines directes (horizontales ou verticales ou en profondeur).

Nous commencerons par introduire dans les sections 3.1 et 3.2 comment se fait l'extraction et la sauvegarde des informations dont les utilisateurs ont besoin. Ensuite, nous introduirons comment se font les représentations de ces informations par des visualisations pratiques basées sur les principes de perception à la section 3.3. Pour clore ce chapitre, nous discuterons de comment il est possible de réduire l'effort cognitif des recherches par la combinaison intelligente de ces visualisations et de ces informations à la section 3.4.

3.1 Métriques pour la représentation des logiciels

L'objectif de cette thèse étant la visualisation du logiciel, il faut donc déterminer quelles informations sous-jacentes il est préférable de présenter. Le code étant une série d'instructions qui est écrite dans un langage de haut niveau avec du texte, il est nécessaire d'utiliser une forme abstraite pour le représenter [48]. Il est évidemment possible d'utiliser des codes de couleurs ou de reformater le texte pour obtenir une meilleure lisibilité, mais ces stratégies n'utilisent pas le plein potentiel de la visualisation et donnent peu d'informations supplémentaires sur le code et sur sa qualité. Une bonne façon de visualiser le code est d'utiliser des métriques. Ces dernières permettent de décrire de façon efficace plusieurs facettes du code. Chacun des éléments qui seront visualisés dans notre approche possède donc un vecteur de métriques qui peut être étendu ou modifié selon les besoins des utilisateurs.

Une métrique est tout simplement un calcul effectué ou une donnée recueillie sur le code qui se rapporte à une entité. Dans notre cas, on s'intéressera aux métriques concernant les lignes de code, les méthodes, les classes ou interfaces et les paquetages. Bien que notre approche soit basée sur un vecteur de métriques qui peut changer selon les besoins, nous calculons tout de même quelques métriques de base qui sont pertinentes dans les études que nous faisons. Ces métriques sont celles proposées dans la théorie et leur calcul est simple une fois la structure du modèle interne créée. Les métriques sont classifiées selon l'aspect du logiciel (contexte) et selon le niveau d'abstraction de l'élément visé (niveau de granularité). Les sections 3.3.1 et 4.1 discutent des niveaux de granularité et des contextes du logiciel alors que plus de détails sur les façons de calculer les métriques et leur pertinence dans l'analyse du logiciel sont présentés à la section 5.2.

Le choix de représenter des métriques a deux avantages principaux. Dans un premier temps, il s'agit de données simples. En effet, il est possible de résumer avec seulement un vecteur de métriques plusieurs aspects d'un logiciel. On peut aussi cibler des informations pertinentes, par exemple le couplage dans un logiciel, en éliminant des informations superflues qui ralentiraient notre analyse comme le nom des variables. Par exemple, il est plus simple d'interpréter un nombre correspondant au couplage que d'interpréter

une liste d'éléments correspondants aux appels ou pire, de devoir chercher ces appels dans le texte d'un programme. Les métriques peuvent être des valeurs numériques ou parfois des valeurs nominales. Il est donc facile de faire des statistiques sur ces informations, ou encore de les agréger pour obtenir des informations sur l'élément au niveau de détails supérieur, ou de décortiquer la métrique pour obtenir les informations pour le niveau inférieur. Par exemple, on peut additionner les valeurs de couplage des classes d'un paquetage pour avoir une idée du couplage de ce dernier. Les mesures de tendances centrales faciles à calculer sont aussi un exemple de l'utilité des métriques dans l'analyse du logiciel.

Dans un deuxième temps, les métriques ont déjà été utilisées avec succès et à plusieurs reprises pour évaluer la qualité du logiciel [12, 65]. Les métriques ont d'ailleurs été développées dans le but avoué d'évaluer la qualité des logiciels. Il s'agit en fait de techniques empruntées à d'autres secteurs de l'ingénierie et appliquées presque directement aux logiciels. Puisque l'évaluation de la qualité figure parmi les objectifs principaux de la représentation du logiciel, il est pertinent de les utiliser dans la représentation pour répondre à ce genre de questions. Les métriques sont aussi une façon facile d'évaluer cette qualité puisque leurs valeurs peuvent être comparées à des seuils, ou encore des utilisateurs peuvent détecter rapidement des patrons visuels les impliquant.

3.2 Calcul des données en direct

La première partie de notre approche consiste à calculer les données nécessaires à l'analyse pour ensuite les montrer aux utilisateurs. Ce calcul se fait à l'intérieur même de l'environnement de développement sans affecter la performance de l'éditeur ou des utilisateurs. Ceci est différent de la technique traditionnelle où des outils externes calculent les métriques après la phase de développement ou à la demande des utilisateurs pour un instant précis.

Les métriques sont donc calculées avec les outils de l'environnement de développement et le travail est effectué au fur et à mesure que les utilisateurs écrivent le code dans les différentes classes. Bien entendu, il est important que ces calculs n'interrompent pas

ou ne ralentissent pas le travail des utilisateurs pour profiter des avantages de ces informations. Le calcul de ces informations par des outils externes est souvent plus lourd et peut prendre plusieurs minutes. Par contre, quand le travail est fait en fonction des modifications des utilisateurs, il n'est pas nécessaire de refaire les calculs au complet à chaque fois, car on peut seulement tenir compte de ce qui a été modifié. Ces changements ciblés peuvent être faits en parallèle et ne prennent qu'un instant une fois le modèle en mémoire créé.

Pour arriver à faire ces calculs en même temps que les valeurs sont modifiées, il faut conserver un modèle mémoire qui contient la structure des programmes permettant de calculer les métriques plutôt que de conserver seulement le résultat des calculs des métriques elles-mêmes. En effet, pour plusieurs métriques, il est difficile d'utiliser uniquement sa valeur et de l'incrémenter ou de la décrémenter à partir simplement des informations contenues dans une modification de programme.

La disponibilité de ces informations a deux avantages principaux. D'une part, ces informations permettent l'analyse en direct des logiciels pour prendre des décisions rapidement dans le processus de création ou des modifications peuvent être faites rapidement pour pallier une erreur déjà commise. D'autre part, elles permettent aux utilisateurs de mieux comprendre comment les différentes pratiques affectent concrètement l'évolution du développement et la qualité d'un logiciel. Ils peuvent ainsi mieux comprendre l'importance de la qualité, car ils voient l'impact direct qu'ils peuvent avoir non pas uniquement théoriquement, mais bien par des exemples tirés de leur propre code. L'enseignement des bonnes pratiques face à la qualité est donc à double sens. D'un côté, les utilisateurs comprennent mieux comment le code influence la qualité et de l'autre côté la pratique les amène à mieux saisir la théorie derrière la qualité du logiciel en l'imprégnant dans des programmes pratiques qu'ils connaissent bien. Le phénomène est similaire à celui des correcteurs orthographiques modernes qui montrent en temps réel les erreurs aux utilisateurs. Les utilisateurs de ces outils voient où se trouvent les erreurs dans leur texte et peuvent les corriger en parallèle de l'écriture, mais continuent aussi leur apprentissage en voyant plusieurs exemples de règles orthographiques appliquées dans leur propre texte.

3.3 Visualisation

Une fois les informations disponibles, il faut trouver une façon de les présenter aux utilisateurs pour qu'ils puissent être en mesure de les analyser rapidement. L'analyse rapide est indispensable non seulement pour que les utilisateurs soient efficaces et que l'activité ne soutire pas trop de temps à leur tâche de développement, mais aussi pour garder leur intérêt envers l'analyse. L'intérêt est un des facteurs principaux de la performance d'un outil d'analyse et il est d'ailleurs une des forces majeures de la visualisation [71, 77].

Tout comme le calcul des métriques et leur sauvegarde, la visualisation se doit elle aussi d'être intégrée à l'environnement de développement. De cette façon, les utilisateurs restent concentrés sur leur tâche et n'ont pas à changer d'outil et donc de contexte quand ils désirent visualiser des informations. Ce principe est expliqué dans la section 3.4 portant sur l'importance de la cohérence et en particulier celle de l'outil intégré.

3.3.1 Axes et multiples vues

Un des principes fondateurs de notre approche en visualisation est l'accès à une multitude de vues dans un même outil. Ces vues sont organisées selon trois axes et nous utilisons une métaphore de navigation le long de ces trois directions. La figure 3.1 illustre l'espace de navigation des utilisateurs cherchant des informations sur le logiciel à l'aide de la visualisation. Chaque cellule présentée à la figure 3.1 représente une vue accessible par les utilisateurs. Pour conserver la cohérence lors des déplacements des utilisateurs, les mouvements doivent se faire dans une seule direction à la fois et être rapides pour conserver une continuité. Les multiples vues servent à augmenter les informations disponibles tout en réduisant les efforts pour les chercher. Toutes les vues étant créées dans un même esprit, il est plus facile pour les utilisateurs d'interpréter les associations graphiques sans nécessairement avoir étudié ces associations pour la nouvelle représentation. Les vues sont aussi conçues pour permettre aux utilisateurs d'interagir avec l'environnement et d'accéder à plus d'informations et non pas seulement à présenter ces informations de manière prédéterminée. Il ne faut pas confondre la navigation

dans ces axes avec la navigation dans l'espace en trois dimensions d'une vue donnée. Les détails de l'implantation et de la réalisation des vues sur ces axes ainsi que la navigation entre ces dernières sont présentés dans le chapitre 4 qui traite de la visualisation.

Les vues sont créées selon des principes de perception et de performance du système visuel humain. En effet, la cohérence est importante pour permettre aux utilisateurs de ne pas perdre le fil lors des différents types de navigation dans *VERSO*, mais il faut d'abord que chaque vue individuelle soit efficace. Ces principes prennent leur source principalement dans les travaux de Healey [36–40] qui discutent d'un phénomène que nous traduisons librement par perception instantanée (*preattemptive perception*), c'est-à-dire que les attributs graphiques qui détonnent dans une image sont perçus sans qu'il y ait un besoin d'analyse de la part des observateurs. En fait, aucun balayage des éléments n'est nécessaire. Le fait de ne pas avoir à se concentrer pour interpréter les attributs graphiques laisse plus de temps aux utilisateurs pour réfléchir sur la signification de ces attributs et sur comment ils pourraient améliorer le logiciel. Dans le prochain chapitre, il sera question de nos choix graphiques effectués pour supporter ces principes. Entre autres, les éléments graphiques ont été choisis pour réduire leurs influences mutuelles et pour permettre une lecture plus rapide et plus précise des valeurs. Bien qu'appliqués à un autre domaine que la visualisation, les travaux de Rensink [78–80] sont eux aussi très pertinents sur la capacité de remarquer des changements subtils entre deux images ou dans des séquences vidéo. Ses recherches sur l'animation nous aident à mieux comprendre à quel point la cohérence est primordiale quand on veut utiliser plusieurs représentations pour effectuer une tâche qui semble pourtant très simple. Par contre, quand le principe de cohérence visuelle est présent, le système visuel humain s'adapte très bien et peut percevoir des modifications de façon instantanée. La figure 3.2 illustre ses recherches à travers un exemple.

De la même façon, Ramachandran et Hirstein [77] discutent du fascinant dans l'art en parlant de *peak shift*. Leurs recherches donnent des indices sur ce qui retient l'attention dans une image et sur ce qui plaît aux observateurs. Ceci est intéressant à deux égards. Dans un premier temps, il est important d'utiliser ce phénomène de *peak shift* pour mettre l'emphase sur certaines portions de l'image et ainsi mettre l'accent sur des

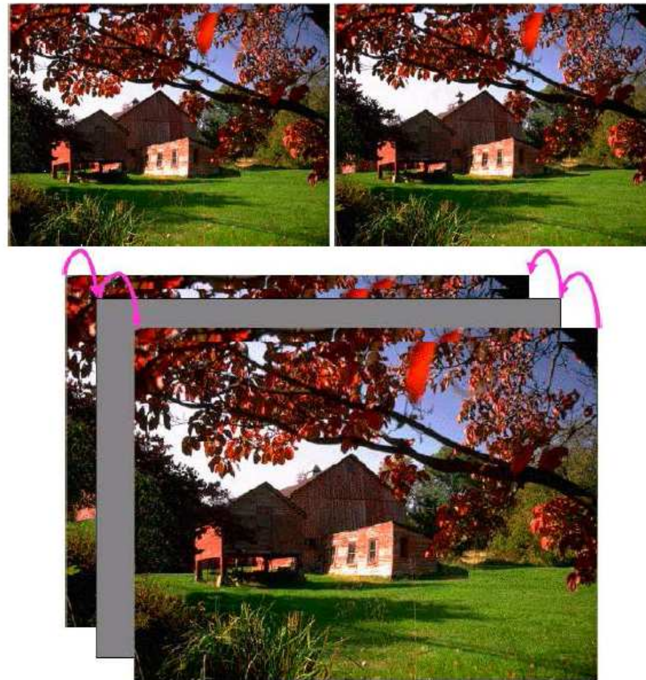

Figure 3.2 – Présentation de la cohérence. Quand deux images sont placées côte à côte (haut), il est plus difficile d'identifier rapidement les différences entre elles. Il y a cinq différences importantes entre ces deux images. Lorsque les deux images alternent au même endroit, les différences sont beaucoup plus faciles à détecter. Cependant, aussitôt qu'on insère une image de couleur uniforme entre les transitions (bas), il est beaucoup plus difficile de retrouver ces mêmes différences. Le système visuel humain est très sensible à de telles coupures dans la cohérence graphique.

informations importantes. Dans un deuxième temps, l'analyse est une activité considérée comme une corvée, souvent même considérée moins intéressante que la programmation elle-même. Le fait d'utiliser des graphiques qui répondent à certains critères énoncés par Ramachandran et Hirstein pour créer un outil de visualisation aide non seulement à le faire adopter, mais à rendre plus intéressantes des tâches jusqu'à maintenant considérées monotones. De plus, étant donné que *VERSO* est construit à même l'IDE, les gens peuvent passer d'une vue à l'autre et ainsi faire l'analyse avec des graphiques intéressants sans avoir à réserver du temps spécialement pour cette activité, réduisant encore plus les réticences liées à l'analyse des informations sur le code.

3.4 Principe de cohérence

Un des principes que nous désirons mettre de l'avant dans cette thèse est la cohérence. Ce que nous entendons par cohérence est tout ce qui aide les utilisateurs à conserver leur concentration malgré les modifications dans des sous-ensembles de données présentés. En d'autres termes, les utilisateurs se créent un contexte mental quand ils font une activité et se concentrent à l'intérieur de ces repères. Par exemple, si des utilisateurs doivent utiliser un deuxième outil pour faire une tâche donnée, ils doivent se concentrer à nouveau et refaire une structure mentale de la tâche qu'ils sont en train d'effectuer dans le nouveau contexte. Ceci risque fort de ralentir les utilisateurs les moins expérimentés. Aussi, des utilisateurs pourraient être en train de travailler sur un paquetage particulier dans leur environnement de développement et désirer consulter les métriques pour les classes du paquetage sur lequel ils travaillent. Il serait beaucoup plus facile pour eux de voir les informations directement en cliquant ou en activant une touche sur les paquetages présents dans son explorateur de paquetage. Par contre, la plupart des outils sont externes et ne permettent pas d'avoir accès aux informations sans sortir de l'environnement de développement. D'autres approches présentent les informations à l'intérieur d'un IDE ou présentent de multiples vues sur différents aspects dans un même environnement, mais trient les informations différemment dépendamment du contexte. Même une petite différence comme l'ordre des éléments ou la façon de regrouper les classes dans deux outils différents (par exemple, des classes regroupées en fonction de leur hiérarchie architecturale dans l'outil A alors qu'elles le sont en ordre alphabétique dans l'outil B) demande un effort considérable de restructuration qui devient vite fatiguant si on doit faire plusieurs aller-retours. Quand les outils utilisent des structures visuelles complexes, le problème en est accentué puisque les représentations d'entités abstraites que sont les parties d'un logiciel sont souvent différentes d'une métaphore à l'autre. Si une classe est représentée dans deux outils et est disposée de manière complètement différente dans deux représentations, la difficulté n'est plus seulement de retrouver les éléments correspondant dans les deux représentations, mais bien de reconnaître la nouvelle représentation de l'élément dont les liens avec le reste du logiciel sont potentiellement

aussi différents. Ce problème se pose évidemment aussi lors de l'utilisation de plusieurs logiciels de visualisation en même temps.

3.4.1 La cohérence liée à un outil intégré

Une forme de cohérence présente dans notre approche est le fait d'avoir un seul outil où plusieurs activités peuvent être menées simultanément. Dans un premier temps, *VERSO* permet de calculer des informations sur le code sans utiliser de programmes intermédiaires. Il n'est donc pas nécessaire pour les utilisateurs d'apprendre un programme pour générer les informations et ensuite transformer le format de sortie pour que le logiciel de visualisation l'accepte. Même des utilisateurs ayant peu de connaissances de l'outil peuvent avoir accès à ces informations sur leur logiciel alors que tout ce processus est fait en arrière-plan de façon transparente.

Ensuite, la visualisation avec ses multiples vues permettant de répondre à une panoplie de questions différentes sur plusieurs domaines est aussi pleinement intégrée à l'outil de développement. Cet accès direct évite de devoir sortir du cadre de l'environnement de développement pour faire les analyses, et de devoir se réapproprier les mécanismes du nouvel outil. D'autant plus que le processus lié à l'évaluation d'une modification rapide est fastidieux à cause des différentes étapes de lecture et d'interprétation du code en plus du reformatage des informations. D'un autre côté, avec le principe de l'outil intégré, toutes ces étapes sont évitées ou encore faites de façon instantanée. Le seul fait de changer d'environnement pour visualiser les informations demande un effort cognitif considérable. Le principe et les arguments sont les mêmes si les utilisateurs doivent requérir à plusieurs outils différents pour voir les différents contextes ou niveaux de granularité s'ils ne peuvent répondre à toutes leurs interrogations avec un seul outil spécialisé. L'accès à un outil intégré est beaucoup plus simple et évite la multiplication des apprentissages des fonctionnements de plusieurs approches.

3.4.2 La cohérence visuelle

La cohérence est un choix naturel pour l'amélioration de la visualisation du logiciel. En effet, le logiciel est une construction ayant une cohérence intrinsèque. Un logiciel ne possédant pas cette cohérence serait difficile à comprendre et à maintenir. Dans les langages de programmation, qu'ils soient basés sur des classes ou des fichiers, le code est construit de façon à ce que les fonctionnalités ou les éléments ayant des liens entre eux sont placés près les uns des autres. Cette cohérence facilite la recherche d'informations puisque des détails sur un élément peuvent être étudiés en rétrécissant son point de vue alors que des informations sur le voisinage d'une entité peuvent être trouvées en l'élargissant. La cohérence est aussi présente dans l'évolution du logiciel puisque le processus de création est incrémental. Les programmeurs partent d'une version donnée et ensuite ajoutent, modifient ou suppriment des éléments. Typiquement, les changements entre deux versions sont petits et les modifications s'opèrent graduellement. Les programmeurs n'ont donc pas à faire des efforts importants pour comprendre la prochaine version d'un logiciel. Il s'agit maintenant de transformer cette cohérence du logiciel en cohérence visuelle.

Dans cette thèse, nous nous intéressons à la cohérence visuelle présente entre les niveaux de granularité, les contextes et les versions d'un logiciel.

3.4.2.1 Cohérence dans la granularité

Ce que nous entendons par granularité dans le logiciel réside dans l'importance de division des éléments. Par exemple, nous nous intéressons dans cette thèse à ces différents niveaux : le système, le paquetage, la classe, la méthode et la ligne de code. Il existe une relation de composition entre ces différents niveaux du concept le plus large jusqu'au plus petit composant. En plus des déplacements naturels dans l'espace, les déplacements entre les différents niveaux de granularité devraient se faire à la manière d'un zoom sémantique où de plus en plus de détails et d'informations sont affichés. La cohérence entre les niveaux de granularité est importante puisqu'elle permet aux utilisateurs de demander plus de détails sur une région spécifique ou encore d'avoir une meilleure

vue d'ensemble du logiciel. Les transitions entre ces différents points de vue doivent se faire sans que les utilisateurs aient l'impression qu'ils se retrouvent face à une nouvelle représentation. Plus de détails sur la façon exacte utilisée pour atteindre ce but sont présentés dans le chapitre 4 portant sur les aspects plus techniques de la visualisation.

3.4.2.2 Cohérence par rapport au contexte

La cohérence lors du changement de contexte est inscrite dans le fait que les utilisateurs sont face à un outil intégré. Dans ce cas, non seulement ils n'ont pas à changer d'outil, mais ils restent dans un univers familier du point de vue de la structure des graphiques présentés. Bien que les associations entre attributs graphiques et métriques changent en modifiant le contexte, les nouvelles informations sont appliquées sur les mêmes objets graphiques qui sont disposés de la même façon. Plusieurs outils tendent à se spécialiser et à offrir une vue parfaitement adaptée pour un problème précis, par exemple, en changeant l'entité d'intérêt principal selon que nous sommes dans un contexte ou un autre. Plus concrètement, on pourrait utiliser la classe comme objet à représenter pour le contexte de la qualité du logiciel et utiliser plutôt l'auteur au moment de représenter les informations sur les systèmes de contrôle de versions. Dans le cas d'un analyste qui étudie un aspect précis, cette solution est certainement intéressante, mais par contre, pour des utilisateurs cherchant à comprendre ce qui se passe dans les différentes parties de leur logiciel, conserver une continuité même quand on change de contexte devient important. Cette continuité leur épargne une charge cognitive tout en leur permettant de répondre à leurs interrogations.

3.4.2.3 Cohérence dans l'évolution

Pour ce qui est de la cohérence durant l'évolution du logiciel, il s'agit d'être en mesure de suivre les changements dans un logiciel sans perdre le fil des éléments communs d'une version à l'autre. Cette cohérence est semblable au principe des dessins animés. Un film d'animation est en réalité une série d'images qui sont dessinées une par une et montrées rapidement les unes après les autres. Notre cerveau est capable de com-

prendre l'histoire qui se déroule devant nos yeux puisque les images sont dessinées très semblables les unes des autres. Les seuls changements sont les mouvements des éléments dans le film alors que la majorité de l'image reste inchangée. Si les images sont complètement différentes d'une représentation à l'autre, on pourrait difficilement suivre la progression du film. Pour l'évolution du logiciel, c'est le même principe, car s'il y a d'énormes changements dans la représentation graphique du logiciel, il nous sera impossible de suivre un élément dans le temps. Entre autres, la disposition des éléments dans l'espace devrait bouger le moins possible ou être supportée par des animations pour simplifier l'analyse. Le logiciel est construit de façon incrémentale, c'est-à-dire qu'on ajoute toujours des éléments ou on les modifie à partir d'une base de code déjà existante. Puisque le logiciel est fait de façon incrémentale, les données contiennent déjà cette cohérence et peuvent ainsi être visualisées de façon efficace. Les ajouts et les modifications apparaitront petit à petit et les images montrées seront semblables entre elles.

CHAPITRE 4

VISUALISATION

La visualisation est une des parties importantes de notre recherche de doctorat. La visualisation est la fenêtre des utilisateurs d'où ils peuvent tirer des informations sur les éléments. Une visualisation efficace permet d'analyser rapidement un logiciel avec justesse [93]. Notre visualisation utilise différents niveaux de granularité qui se déclinent avec différents contextes incluant pour chaque combinaison (niveau-contexte) une représentation de l'évolution du logiciel. Le résumé de toutes les vues possibles a été d'ailleurs abordé dans le chapitre précédent. Comme discuté auparavant, le principe unificateur est de conserver un environnement semblable entre les vues pour renforcer la compréhension des utilisateurs. Toutes les vues disponibles permettent d'augmenter la quantité d'informations visibles tout en réduisant les possibilités d'occultation et de saturation du système visuel des utilisateurs.

4.1 Granularité des représentations

VERSO permet de montrer plusieurs niveaux de granularité selon que les utilisateurs désirent une vue globale ou détaillée du logiciel. La façon principale de se déplacer d'une granularité à l'autre est de positionner la caméra à l'intérieur de l'environnement en trois dimensions. Donc plus les utilisateurs se rapprochent du plan sur lequel se trouvent les éléments, plus le niveau de granularité devient petit. On voit donc les paquetages en étant éloigné du plan pour avoir une meilleure vue d'ensemble. Ensuite, à mesure qu'on se rapproche du plan et que certaines parties du logiciel sortent du cadre de la visualisation, on voit les paquetages disparaître pour voir apparaître les classes qui les composent. Par la suite, en s'approchant encore plus des classes au moment où il n'est plus possible que d'en voir quelques-unes, les boîtes des classes disparaissent pour montrer les méthodes qui composent les classes. Le niveau des lignes de code étant montré dans une fenêtre à part avec des paradigmes graphiques légèrement différents, il est atteint à l'aide de

commandes sur les méthodes ou les classes affichées. Il est aussi possible d'imposer un niveau de granularité, peu importe la distance de la caméra. Les déplacements entre les granularités des représentations équivalent à des déplacements verticaux sur la figure 4.1 présentée dans le chapitre 3 et rappelée ici. La figure 4.2 montre trois des niveaux de granularité présentés sur un même logiciel.

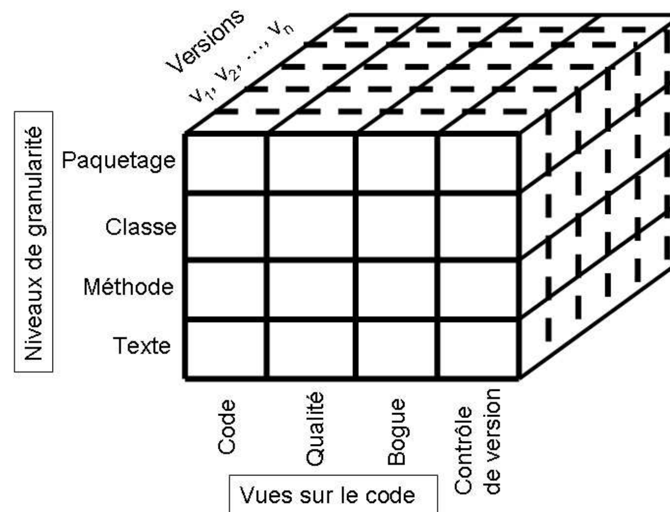

Figure 4.1 – Matrice des différentes vues accessibles. Chaque cellule représente une vue décrite dans ce chapitre.

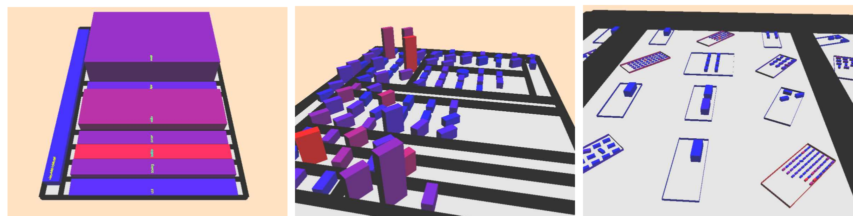

Figure 4.2 – Visualisation d'un même logiciel dans trois niveaux de granularité différents. De gauche à droite, on peut voir le niveau de paquetage, des classes puis des méthodes. Toutes les vues sont placées dans le contexte de la qualité.

4.1.1 Représentation des classes

Le premier élément auquel nous nous intéressons est la classe. Bien qu'elles soient situées au centre de l'échelle des granularités, les représentations de classes ont été chronologiquement développées en premier et elles dirigent l'aspect graphique des éléments qui se trouvent dans les autres niveaux. C'est aussi le niveau de granularité qui est le point de départ d'une analyse typique, car il permet à la fois d'avoir une vue d'ensemble sur le logiciel tout en ayant un niveau de détails pertinent.

4.1.1.1 Caractéristiques graphiques

Comme mentionné dans le chapitre précédent, les éléments d'un logiciel sont résumés à l'aide d'un vecteur de métriques les représentant. Les classes répondent évidemment à ce même principe. Certaines valeurs de ce vecteur sont associées à des caractéristiques graphiques de la représentation des classes permettant d'évaluer graphiquement la valeur des métriques. Le principe est de montrer le plus d'informations possible sans empêcher les utilisateurs d'interpréter adéquatement les informations déjà disponibles.

Dans *VERSO*, les classes sont représentées par des boîtes allongées. Ces boîtes comportent trois caractéristiques soient la hauteur, la couleur et la rotation (autour de l'axe central de l'élément pointant vers le haut). La hauteur minimum permet de bien distinguer la présence ou l'absence d'une classe. Il y a aussi une valeur maximum fixe pour la hauteur des classes. Pour la couleur, on peut utiliser n'importe quelle échelle, mais par défaut, notre valeur minimum est un bleu pur alors que notre valeur maximale est un rouge pur dans le système de couleur *RGB*. Les valeurs intermédiaires passent par une série de violet se rapprochant du bleu ou du rouge selon la proximité d'une des deux extrémités (la figure 4.5 montre trois couleurs à l'intérieur de ce gradient). Pour ce qui est de la rotation, la valeur minimum est une rotation de 0 degré alors que la valeur maximum est de 90 degrés. Des valeurs plus grandes que 90 degrés seraient difficiles à déterminer puisque l'axe de rotation se trouve au milieu de la représentation. La figure 4.3 décrit visuellement les caractéristiques graphiques utilisées pour la classe.

Les interfaces sont représentées par des cylindres pour les différentier facilement des

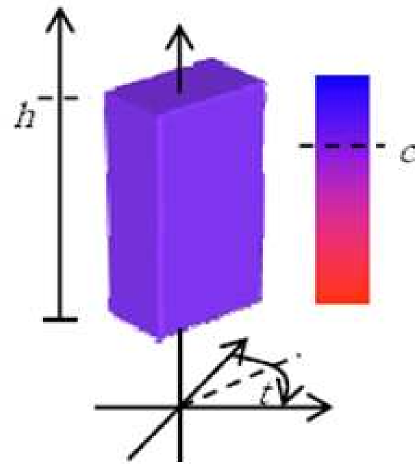

Figure 4.3 – Exemple des trois caractéristiques graphiques liées aux classes. On peut y voir la hauteur (h), la couleur (c) et la rotation (t).

classes. Évidemment, les cylindres ne possèdent pas la caractéristique graphique de la rotation puisqu'elle serait indistinguable pour ce genre de forme. Il ne s'agit pas d'un problème important puisque les interfaces sont moins complexes que les classes et ne requièrent habituellement pas autant de métriques. La distinction entre les classes et les interfaces est jugée plus importante que l'ajout de la représentation d'une troisième métrique.

De plus, il existe des maxima assignés par les utilisateurs pour les valeurs des métriques. Ceci implique que la valeur maximum des caractéristiques graphiques est alignée sur ces valeurs de métriques maximums. Ceci empêche que les valeurs extrêmes aplatissent les autres valeurs. En effet, selon la distribution, des valeurs près de la médiane pourraient être représentées par des valeurs graphiques très faibles si le maximum est disproportionné, ce qui n'est pas souhaitable lors de l'analyse. L'inconvénient de cette façon de faire est que plusieurs valeurs très élevées seront ramenées au maximum assigné par les utilisateurs lors du rendu, même si elles peuvent avoir un écart considérable. Par contre, ce n'est pas un problème important puisque les valeurs dépassant le maximum sont toutes considérées comme très grandes et les variations n'ont plus d'importance passé un certain seuil. Ces caractéristiques graphiques ont été choisies pour ne pas in-

terférer les unes sur les autres et permettre la lecture la plus juste possible des valeurs représentées.

Le réflexe naturel serait d'afficher le plus de métriques possible en utilisant plusieurs caractéristiques graphiques. Après tout, plus on montre d'informations, plus celles-ci sont captées par les utilisateurs de la visualisation. Par contre, certaines caractéristiques graphiques peuvent avoir une influence sur les autres [40]. D'ailleurs, ce principe est à la base de plusieurs illusions d'optique comme celle montrée à la figure 4.4. Dans notre cas précis, on pourrait être tenté d'utiliser la largeur et la longueur des boîtes, car nous avons trois dimensions disponibles du fait que l'environnement est rendu en trois dimensions. Par contre, en y réfléchissant bien, on se rend vite compte que la largeur aura une influence sur notre perception de la hauteur. En effet, une boîte plus large nous paraîtra plus courte. Cette intuition ainsi que d'autres facteurs d'influence du contexte sur la perception ont aussi été vérifiés par des chercheurs dans le domaine de la psychologie cognitive [98]. La couleur aussi peut se décortiquer en trois axes, mais le système visuel humain a de la difficulté à discerner ces axes quand une couleur arbitraire est présentée [32]. La figure 4.5 montre un exemple des caractéristiques graphiques de la classe à travers trois exemples.

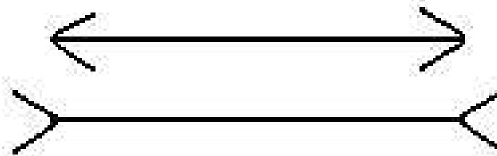

Figure 4.4 – Une illusion d'optique de Müller-Lyer [66] montrant que le système visuel humain est puissant, mais peut parfois être trompé. La plupart des gens qui voient cette image rapidement estiment que la ligne du bas est plus longue que celle du haut. Les deux lignes ont pourtant exactement la même longueur.

4.1.1.2 Placement des entités

La visualisation dont il est question dans cette thèse représente les entités en trois dimensions, mais ces dernières sont disposées sur un plan en deux dimensions. Étant

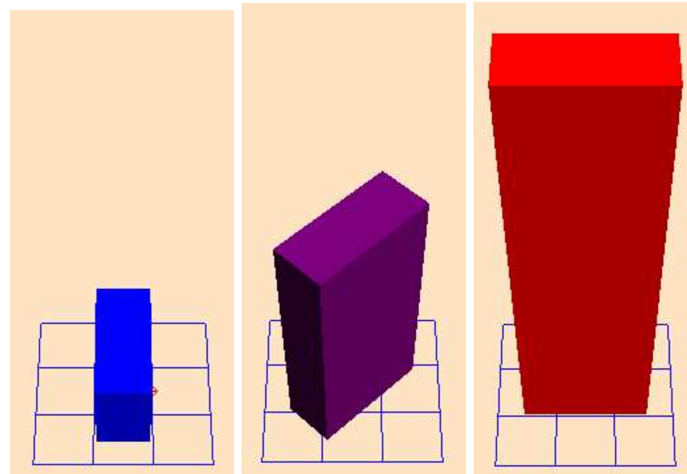

Figure 4.5 – Représentation des classes. Cette image montre des classes représentées à l'aide de *VERSO*. Les trois caractéristiques graphiques, hauteur, couleur et angle de rotation, augmentent de la gauche vers la droite dans cet exemple.

donné le choix des entités et de leurs caractéristiques graphiques, un placement en trois dimensions aurait créé trop d'occultations ou aurait complexifié la navigation de la caméra pour observer les informations nécessaires. Un autre avantage du plan est qu'il réduit la complexité des calculs pour optimiser les positions. En contrepartie, le degré de liberté ainsi perdu réduit le nombre de voisins potentiels d'un élément, ce qui empêche donc d'encoder une information supplémentaire par cette immédiate proximité.

Nous utilisons un espace rectangulaire sur le plan qui contient toutes les classes représentant un logiciel. La disposition des éléments dans ce rectangle suit la hiérarchie des paquetages et est présentée selon un algorithme modifié du *Treemap* [44]. Traditionnellement, le *Treemap* est une structure graphique en deux dimensions où les parents dans la hiérarchie contiennent leurs enfants. Son algorithme de placement essaie d'optimiser l'occupation de l'espace tout en conservant les informations sur des relations frères-enfants-parents. Le *Treemap* a été créé pour représenter des valeurs continues qui peuvent être divisées exactement. De plus, le rectangle représentant la racine est conscrit dans les limites imposées par l'espace initial. Dans notre cas, les éléments représentés utilisent un espace discret sur le plan, c'est-à-dire que leur base a toujours la même forme et la même taille. Étant donné que nous sommes dans un environnement 3D où

la navigation est possible, il n'y a pas de contraintes sur les dimensions du rectangle final représentant la racine et ses descendants. L'absence de cette contrainte nous permet d'adapter le *Treemap* avec des éléments occupant un espace discret.

Notre algorithme modifié de la structure du *Treemap* fait des séparations entre les paquetages à un même niveau et alterne entre des séparations verticales et des séparations horizontales selon que nous sommes dans un niveau pair ou dans un niveau impair. Les éléments sont ensuite placés à l'intérieur des rectangles formés par les paquetages. Si de l'espace supplémentaire est requis à cause du caractère discret des données, on agrandit tout simplement le paquetage correspondant pour obtenir l'espace nécessaire. Ceci crée des trous dans la visualisation alors qu'elle n'utilise pas l'espace de façon maximale, mais nous pensons que c'est un compromis acceptable entre la facilité d'interprétation et l'utilisation de l'espace. Nous avons d'ailleurs implémenté un algorithme de recherche de solution pour minimiser le nombre d'espaces vides dans notre représentation. Cet algorithme permet de tester plusieurs solutions plausibles et choisit la meilleure. La figure 4.6 donne un exemple concret de la façon dont le placement est créé. La figure 4.7 montre un logiciel à ce niveau de granularité.

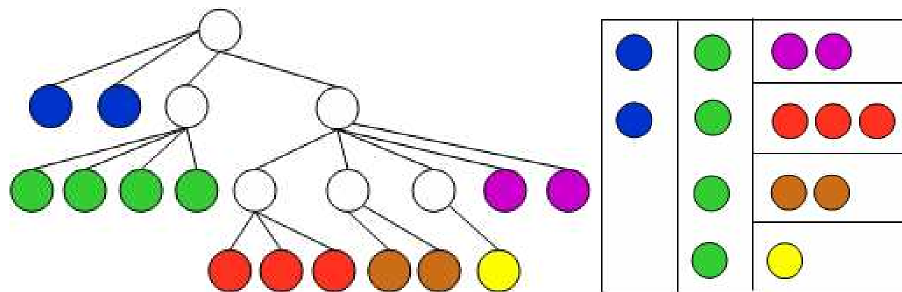

Figure 4.6 – Explication de l'algorithme modifié du *Treemap*. Sur la gauche, un arbre est représenté avec des nœuds et des arcs de la manière traditionnelle. Les feuilles sont en couleur et les autres nœuds sont blancs. Sur la droite, on voit l'équivalent de l'arbre de gauche aplani à l'aide de notre algorithme modifié du *Treemap*. Un paquetage virtuel est créé pour les classes quand leur parent contient aussi des paquetages.

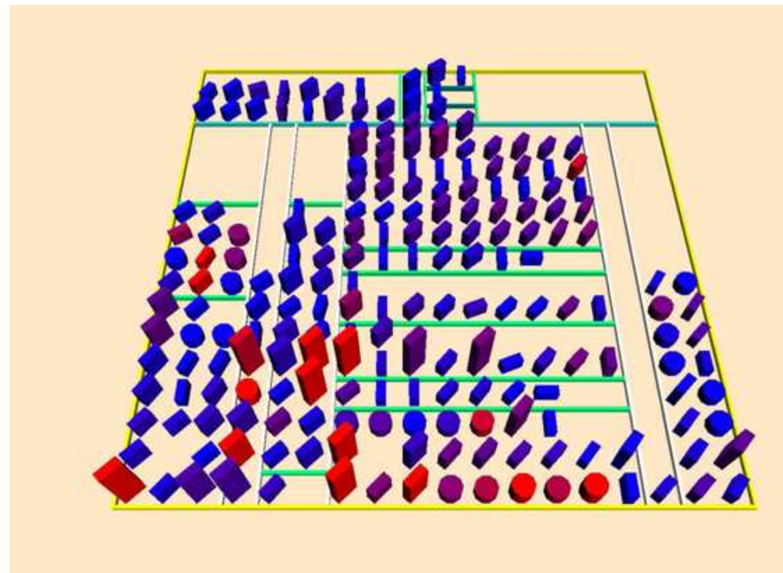

Figure 4.7 – Exemple de l'algorithme modifié du *Treemap*. Le logiciel Freemind servant à l'organisation des idées est représenté à l'aide de notre algorithme modifié du *Treemap*. La couleur est associée au couplage, la hauteur à la complexité de la classe et la cohésion à la rotation.

4.1.2 Représentation des méthodes

Une classe est composée de méthodes et d'attributs. Pour les besoins de la visualisation mentionnée dans notre thèse, nous allons uniquement nous intéresser aux méthodes. Les méthodes participent au fonctionnement d'une classe et elles représentent du code exécutable, contrairement aux attributs. De plus, il est intéressant de calculer des métriques sur les méthodes à cause de leur complexité accrue. Les attributs des classes, pour leur part, sont représentés indirectement à travers différentes métriques des classes et des méthodes comme dans le calcul de la cohésion par exemple. Ils pourraient aussi être ajoutés aux visualisations dans des travaux futurs s'ils s'avèrent pertinents pour les utilisateurs.

4.1.2.1 Caractéristiques graphiques

Les méthodes sont représentées de la même façon que les classes, elles sont simplement beaucoup plus petites. Les caractéristiques graphiques sont donc les mêmes que

pour les classes et les formes restent aussi les mêmes. Évidemment, le même raisonnement s'applique tant au sujet des maxima et minima que de l'interaction entre les différentes métriques. De plus, les méthodes se trouvent à l'intérieur des classes disposées sur l'équivalent du plancher de ces dernières. On peut les voir quand on s'approche très près des classes et de leur base. De plus, pour conserver le contexte des classes, on affiche une bande autour des classes ressemblant à un muret ou aux fondations de la classe. Ce muret conserve les caractéristiques de couleur et de rotation de la classe originale et permet de scinder les représentations des méthodes pour reconnaître rapidement leur appartenance. Étant donné que les méthodes se trouvent à l'intérieur de la représentation des classes, leur rotation est relative à ces dernières. Il est d'ailleurs beaucoup plus facile de lire la valeur de la rotation des méthodes à cause de ce contour. Il a été tenté de garder une rotation absolue dans un premier temps, mais la représentation était contre-intuitive et induisait des erreurs de lecture.

4.1.2.2 Placement des entités

Les méthodes n'ont pas une relation hiérarchique comme les classes. Elles sont en général placées dans une classe selon un ordre chronologique, un ordre d'importance ou regroupées par fonctionnalité. Ceci dépend en fait du programmeur ou des pratiques en cours dans l'entreprise. Il serait intéressant de toujours placer les méthodes selon leur fonctionnalité dans la classe, mais malheureusement ces regroupements existent seulement dans la tête du programmeur et n'apparaissent pas dans la syntaxe. Ils sont même souvent difficiles à détecter pour un expert qui aurait accès à toutes les ressources dont il a besoin. Ce genre de données reste suggestif et ne peut être décidé que par des experts ou encore approximé à l'aide d'outils en intelligence artificielle.

Pour ces raisons, nous avons placé les méthodes les unes après les autres à l'intérieur d'une classe selon l'ordre dans lequel elles apparaissent dans le texte sans ajouter de séparation ou de placement plus complexe. Les rangées sont tout simplement créées en partant du haut et à la gauche de l'emplacement occupé par la classe. Par contre, contrairement aux classes qui ont toutes une base de la même taille, les méthodes peuvent varier de grosseur selon la classe. En effet, dépendamment du nombre de méthodes dans une

classe, la taille de la base des méthodes est ajustée pour permettre à toutes les méthodes d'être présentes tout en diminuant l'espace perdu. Pour calculer la taille d'une méthode, il s'agit de trouver la plus faible valeur de $2x^2$ plus grande que le nombre de méthodes dans la classe. Le pire cas pour le gaspillage d'espace survient quand trois méthodes doivent être placées dans un espace pouvant en contenir huit. Ensuite, à neuf méthodes, on gaspille exactement la moitié de l'espace et la proportion de l'espace qu'il est possible de gaspiller dans le pire cas diminue à mesure que le nombre de méthodes dans la classe augmente. Un petit schéma est présenté à figure 4.8 pour illustrer le placement des méthodes.

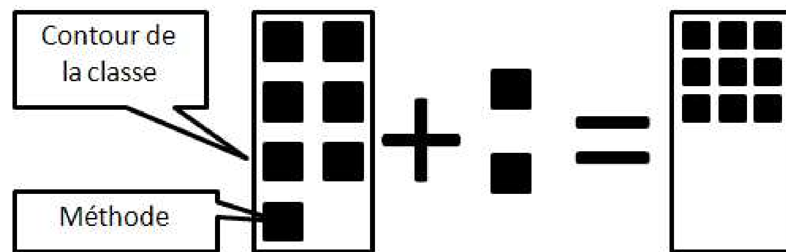

Figure 4.8 – Schéma montrant le principe du placement des méthodes. Étant donné le ratio de la base des représentations des classes, on trouve la plus petite valeur de $2x^2$ plus grande ou égale au nombre de méthodes. L'espace gaspillé sera inférieur à la moitié de la classe pour le pire cas aussitôt qu'on dépasse 10 méthodes à l'intérieur d'une classe.

Bien que plus les méthodes soient nombreuses, plus leur taille diminue, il n'est pas plus difficile de lire la métrique associée à la taille puisque les proportions sont conservées par rapport à la base. De plus, au niveau de granularité des méthodes, les utilisateurs ont tendance à se concentrer sur les méthodes d'une seule classe et comparent les métriques des méthodes de cette classe entre elles. Dans ce scénario, toutes les mesures sont évidemment équivalentes. Une autre solution aurait été de choisir la taille de la base comme étant celle de la plus petite base, et donc selon la taille de la base d'une méthode dans la classe comprenant le plus de méthodes. Ceci aurait encore une fois défavorisé le cas moyen au profit du pire cas et le gain quant aux proportions égales pour toutes les méthodes n'est pas suffisant pour justifier une telle perte d'espace. Un exemple de la représentation des méthodes est présenté à la figure 4.9.

Évidemment, les méthodes sont rendues uniquement quand la granularité des mé-

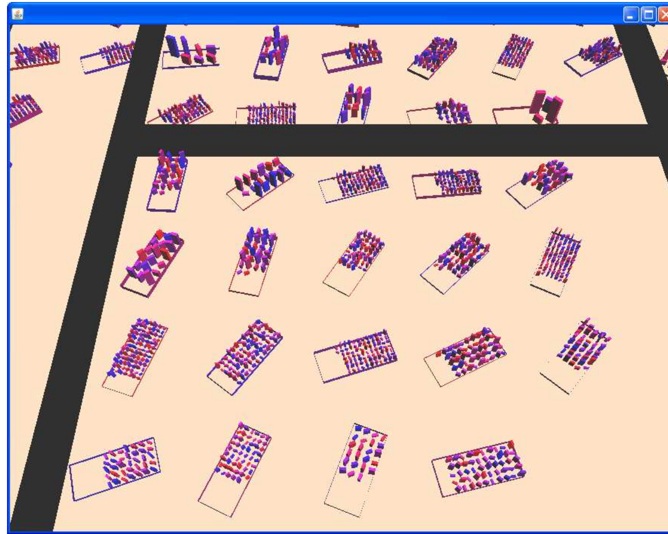

Figure 4.9 – Représentation du niveau de granularité des méthodes. On y voit les méthodes elles-mêmes avec leurs caractéristiques graphiques et une bordure représentant la classe et donnant un repère pour observer la rotation relative.

thodes est visible. Ceci réduit la charge de travail imposée à la carte graphique. Puisque nous travaillons directement avec les primitives graphiques d'*OpenGL* [72], il est possible d'obtenir des visualisations très fluides et de déplacer la caméra rapidement dans l'environnement en trois dimensions. Le fait de ne pas montrer certains éléments invisibles pour un point de vue donné nous aide à conserver cette fluidité pour la représentation de logiciels encore plus grands. De plus, le nombre de méthodes visibles est réduit par le champ de vision de la caméra puisqu'on est généralement très près du plan. Bien que l'apparition des méthodes se produise en s'approchant graduellement du plan, les utilisateurs ont l'impression qu'elles sont constamment présentes.

4.1.3 Représentation des paquetages

La représentation des paquetages est elle aussi dépendante du placement des classes. Ce niveau de granularité donne une vue d'ensemble intéressante sur tout le logiciel et permet surtout de cacher des informations qui pourraient déconcentrer ou saturer le système visuel des utilisateurs lors de certaines tâches d'analyse.

4.1.3.1 Caractéristiques graphiques

Encore une fois, les caractéristiques graphiques des paquetages sont semblables à celles des classes. Par contre, comme nous le verrons dans la prochaine sous-section, les paquetages ont une forme imposée par le niveau des classes et cette forme peut être variable. Ceci implique que le ratio entre la largeur et la longueur n'est pas toujours le même. Dans ce contexte il est difficile, voire contre-intuitif, d'utiliser la rotation pour représenter une métrique. De plus, on va voir aussi dans la prochaine section que la rotation masquerait d'autres parties du logiciel et ce n'est pas l'effet recherché.

La représentation des paquetages a donc comme caractéristiques graphiques la couleur et la hauteur. La hauteur est une caractéristique absolue et n'est donc pas relative à la forme ou à la taille de la base. La couleur agit exactement de la même façon que pour les classes. Le fait d'avoir deux caractéristiques graphiques au lieu de trois n'est pas un problème majeur puisque les paquetages dans les scénarios que nous utilisons n'ont pas souvent besoin de représenter autant d'informations. La figure 4.10 montre des exemples de paquetages comme ils sont présentés dans *VERSO*.

4.1.3.2 Placement des entités

Le placement des paquetages est prédéterminé par le placement des classes. En effet, lors de la création du *Treemap* qui assigne une position à toutes les classes, on doit inévitablement assigner une position aux paquetages qui sont composés de classes (voir la figure 4.6). On dessine d'ailleurs des séparateurs représentant ces paquetages quand on représente le niveau de granularité des classes. Les paquetages ont déjà une représentation graphique au niveau de granularité des classes. En fait, dans le niveau de granularité des classes, les paquetages sont utilisés comme base au niveau du plan et possèdent une couleur gris pâle pour éviter d'introduire un biais dans la lecture des couleurs des classes.

Il est donc facile d'augmenter la hauteur des paquetages en fonction d'une métrique et de changer leur couleur en fonction d'une autre métrique. Les séparateurs sont conservés et la base des boîtes est légèrement réduite par rapport à leur version dans la granularité des classes. Évidemment, les classes enfants des paquetages ne sont pas rendues

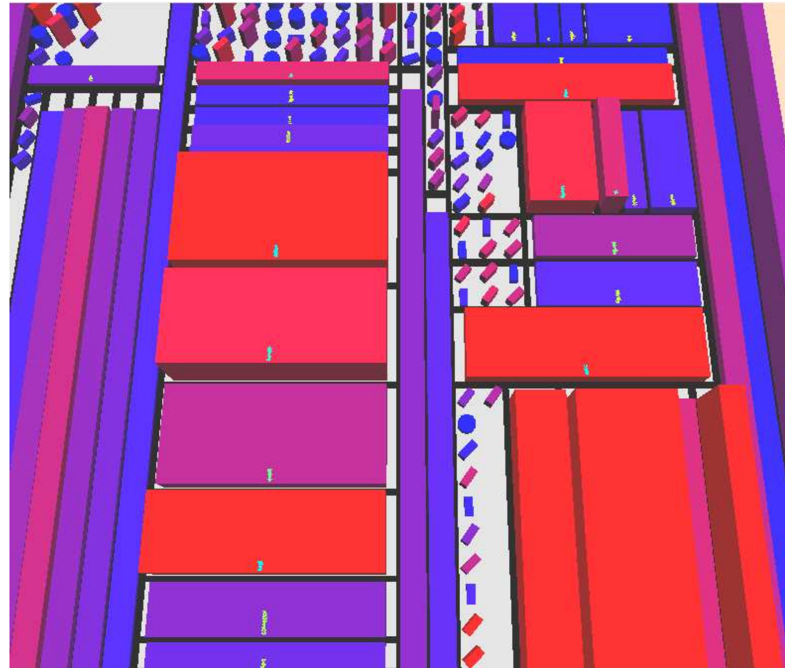

Figure 4.10 – Représentation du niveau de granularité des paquetages. Plusieurs niveaux de profondeur dans la représentation des paquetages peuvent se côtoyer. On peut ouvrir interactivement des paquetages pour voir les classes et les sous-paquetages si c'est nécessaire.

quand ces derniers sont visibles. Étant donné que les grandeurs des éléments et des paquetages sont évaluées en utilisant la même échelle, il est possible qu'une classe dans le paquetage soit plus grande que le paquetage lui-même. Ceci peut amener de la confusion lors de la visualisation et de la navigation dans les niveaux, mais c'est préférable à la solution d'avoir de très grands paquetages qui peuvent causer de l'occultation ou une coupure trop marquée d'échelle avec le niveau des classes. Le rendu de la granularité des paquetages est beaucoup plus rapide que le rendu de la granularité des classes justement parce que les classes ne sont pas visibles dans ce mode.

Puisqu'à cette granularité, les paquetages peuvent avoir plusieurs niveaux, c'est-à-dire qu'il peut y avoir des paquetages qui sont imbriqués dans d'autres paquetages, on peut, à l'intérieur même de ce niveau de granularité, naviguer entre plusieurs niveaux de détails. On peut par exemple s'intéresser aux grands paquetages placés directement sous

la racine ou encore vouloir visualiser les paquetages qui n'ont que des classes comme enfants (c'est-à-dire aucun sous-paquetage). L'interface permet de faire cette navigation aisément et il est aussi possible pour les utilisateurs de fermer certaines parties à leur guise et d'en laisser d'autres ouvertes et ainsi mélanger les niveaux. La figure 4.10 montre un exemple d'un logiciel avec le niveau de granularité des paquetages en présence.

4.1.4 Représentation des relations

Dans tous les niveaux de granularité mentionnés plus haut, il est possible de visualiser les relations. Les relations habituellement visualisées sont les liens de couplage par appel de méthodes ou utilisation de variables de type complexe. On peut aussi visualiser les liens hiérarchiques (parents-enfants) dans le cas des classes et des méthodes. Bien que ces informations ne soient pas calculées dans *VERSO* pour l'instant, il serait aussi possible de visualiser le couplage logique, c'est-à-dire les éléments qui ont tendance à être modifiés en même temps. Toute relation se présentant comme une liste de liens peut être représentée à l'aide de notre approche.

Par contre, *VERSO* ne représente pas les relations dans la forme habituelle du graphe, avec des noeuds et des arcs. Représenter tous les liens explicitement cause des problèmes pour les représentations volumineuses [41]. Les liens s'enchevêtrent et il est impossible d'en suivre un en particulier depuis son noeud de départ vers son noeud d'arrivée.

Pour remédier à ce problème, une solution en deux étapes est proposée. Dans un premier temps, il n'est pas nécessaire de montrer tous les liens en tout temps pour la grande majorité des tâches d'analyse ou de compréhension effectuées sur un logiciel. Les liens sont donc présentés sur demande des utilisateurs seulement pour éviter de saturer la vue quand ce n'est pas nécessaire. De plus, ils sont présentés uniquement pour l'entité demandée, soit la méthode, la classe ou le paquetage. Par contre, les entités pointées seront visibles, peu importe le niveau de granularité. Si un paquetage est attaché à plusieurs classes par exemple, on verra les paquetages ou les classes sous-jacentes selon que le paquetage pointé est ouvert ou fermé. Dans un deuxième temps, comme mentionné plus haut, les liens ne sont pas représentés explicitement. Même si les liens pour un seul

élément sont visibles, afficher toutes ces lignes peut cacher d'autres informations potentiellement pertinentes. Les éléments reliés sont plutôt mis en évidence en désaturant la couleur des autres éléments de la visualisation qui ne sont pas cibles de la relation. De cette façon, seuls les éléments participant à la relation ressortent du lot et peuvent être identifiés instantanément par les utilisateurs. Il s'agit en fait d'un filtre qui représente graphiquement l'appartenance d'un élément à un ensemble. La figure 4.11 montre un exemple de représentation des liens de couplage. Pour une discussion plus en profondeur de l'utilisation des filtres dans *VERSO*, il est possible de consulter notre mémoire de maîtrise [50] ainsi que les travaux de collègues [21, 45].

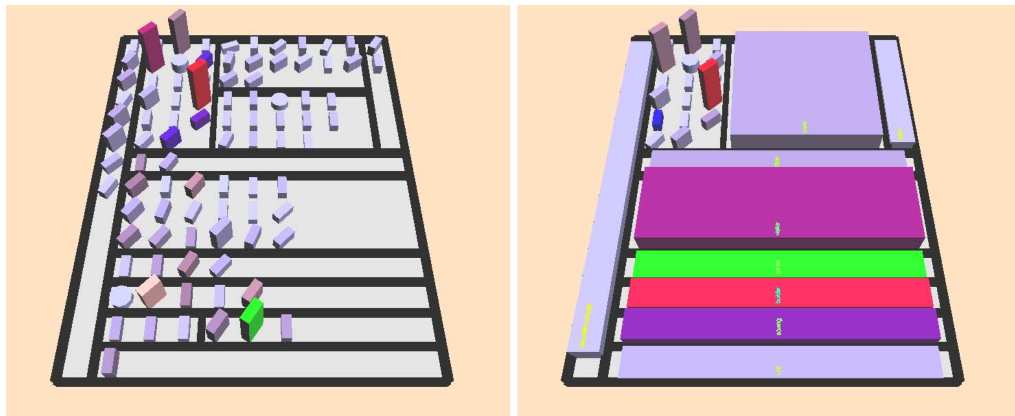

Figure 4.11 – Représentation du système de filtre pour les relations dans *VERSO*. On voit ici le même logiciel représenté au niveau de granularité de la classe et à droite avec plusieurs paquetages fermés. La visualisation des liens s'ajuste au niveau choisi par les utilisateurs.

4.1.5 Représentation des lignes de code

Il existe un dernier type de représentation qui est le plus faible niveau que nous traitons dans cette thèse. Contrairement aux trois autres niveaux de granularité, celui des lignes de code n'est pas inscrit dans le même environnement en trois dimensions que les autres. Cette représentation est donc ouverte dans un nouvel éditeur et la navigation à l'intérieur de cet éditeur n'est pas la même. L'ajout des lignes de code dans la même visualisation que les méthodes aurait été possible, mais leur grand nombre fait qu'elles

sont très lourdes à générer et très lourdes pour le rendu graphique. De plus, le contexte en trois dimensions se prête mal selon nous à la représentation sous forme de boîtes comme c'est le cas dans les autres niveaux puisque les lignes demandent moins de métriques montrées en même temps. Il est intéressant de pouvoir lire le texte en même temps que les métriques sur une ligne et il est plus pertinent de voir tout le fichier texte et de le parcourir plutôt que de voir des représentations 3D des lignes reliées à une méthode donnée seulement.

La façon choisie de représenter le texte est semblable aux techniques utilisées par une des visualisations pionnières et des plus connues : *SeeSoft* [26]. Cette visualisation faite pour représenter un grand nombre de lignes pour du code procédural affiche chaque caractère comme un point de couleur en fonction d'une métrique représentant le type de l'instruction. Évidemment, il est impossible de lire le code dans ces circonstances, mais plusieurs fichiers peuvent être montrés en même temps sur l'écran et parfois même certains logiciels un peu moins volumineux au complet. Notre approche utilise aussi le principe de bandes de couleur proportionnelles à la longueur de la ligne pour représenter les lignes de code. Bien que le type de la ligne soit aussi un choix d'association entre métrique et couleur, il existe d'autres métriques qui s'inspirent des domaines étudiés aux autres niveaux de granularité, par exemple le couplage, la complexité associée à une ligne ou encore l'auteur d'une ligne ou la date de sa dernière modification. Comme aux autres niveaux, les utilisateurs peuvent étendre les métriques proposées et faire de nouvelles associations pour observer de nouveaux phénomènes.

Dans la vue des lignes de code, seule la couleur est associée à une métrique calculée dans le logiciel. De plus, puisque les lignes de code représentent le texte, on a de facto une association entre la taille de la ligne et la taille de la bande de couleur. Il est possible d'agrandir et de rapetisser la largeur des lignes et de montrer ou masquer le texte selon qu'on se concentre plus sur le code ou la distribution des métriques dans le fichier. Dans notre cas il est possible de voir un seul fichier à la fois dans un même éditeur, mais rien n'empêche d'ouvrir plusieurs éditeurs au besoin. Jusqu'à maintenant, il s'agit d'une représentation du texte qui ne peut être modifiée à l'intérieur de cette vue. À terme, il serait intéressant de pouvoir modifier le code lui-même dans cette vue. Par contre,

pour l'instant, il est possible de changer le code avec les outils traditionnels d'*Eclipse* et de voir les changements s'opérer en temps réel. Nous avons aussi créé une légende qui indique la signification des couleurs et un traitement plus précis dans le cas où les informations sont nominales ou ordinales en choisissant un spectre de couleurs plus large permettant l'apparition de couleurs plus distinctes entre les valeurs discrètes. Un exemple de cette représentation est montré à la figure 4.12. Le travail concernant la granularité des lignes de code a été réalisé en collaboration avec Cynthia Beauchemin, une étudiante de passage au laboratoire dans le cadre d'un stage.

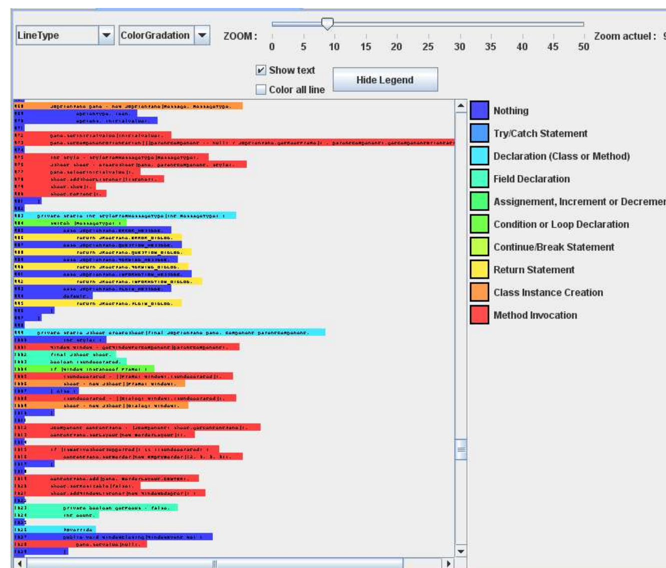

Figure 4.12 – Représentation du niveau de granularité des lignes. Dans le cas présent, on observe un fichier du logiciel *JHotdraw* et le code de couleurs est relié au type de la ligne. Si plusieurs types sont présents sur une même ligne, la couleur affichée sera celle du type qui est placé le plus bas dans la légende.

4.1.6 Navigation et déplacement entre les niveaux de granularité

Le coeur de *VERSO* est un environnement en trois dimensions. Le principal désavantage des environnements en trois dimensions est l'occultation se produisant entre les différentes entités. Effectivement, dépendamment du point de vue des utilisateurs certains éléments seront visibles et d'autres seront partiellement ou totalement cachés. Ce

problème ne survient pas dans les représentations en deux dimensions puisque tous les éléments sont soit toujours visibles ou toujours invisibles parce que placés en arrière-plan. Dans *VERSO*, nous atténuons le problème de l'occultation en utilisant des entités en trois dimensions, mais qui sont limitées à un placement sur un plan en deux dimensions. Ceci réduit beaucoup l'occultation et l'élimine presque lorsque le point de vue est positionné suffisamment au-dessus du plan.

Une autre façon de régler l'occultation est d'utiliser un système de navigation efficace. *VERSO* permet de changer de point de vue en naviguant autour de la demi-sphère se trouvant au-dessus du plan. On peut aussi regarder différents objets sur le plan en se déplaçant latéralement et s'approcher ou s'éloigner du plan pour voir plus ou moins de détails. Le zoom dans l'espace en trois dimensions est fait de façon graduelle et permet aux utilisateurs des déplacements plus petits quand ils sont près du plan et des changements plus rapides quand ils sont loin du plan. Ce point est aussi vrai pour les déplacements latéraux.

Les descriptions faites plus haut de la navigation de la caméra dans l'espace servent aussi aux utilisateurs pour naviguer dans les différentes vues qui ont été présentées dans le chapitre 3. En effet, quand les utilisateurs sont loin du plan, on suppose qu'ils veulent avoir une vue d'ensemble du logiciel et *VERSO* leur montre automatiquement le niveau de granularité des paquetages. Si les utilisateurs s'avancent vers le plan, à un certain moment, les paquetages disparaissent pour laisser la place au niveau de granularité des classes. Ensuite, si les utilisateurs continuent à s'approcher, le niveau de granularité des méthodes apparaît à son tour. Le changement automatique est pratique et convivial et n'enlève pas d'informations aux utilisateurs puisque de toute façon, la taille des éléments au niveau de granularité des méthodes ne permet pas de faire une lecture adéquate ou même approximative des métriques à de grandes distances. Le zoom de la caméra, inclus dans la navigation de l'environnement en trois dimensions agit donc aussi comme un zoom sémantique. On parle de zoom sémantique puisque les informations présentées sont différentes selon qu'on est proche ou qu'on est loin du plan. Un exemple de ce principe de zoom sémantique est présenté à la figure 4.13 par l'entremise de cartes géographiques. Par contre, si les utilisateurs le désirent, il est possible d'imposer la re-

présentation d'un niveau de granularité, peu importe la distance au plan.

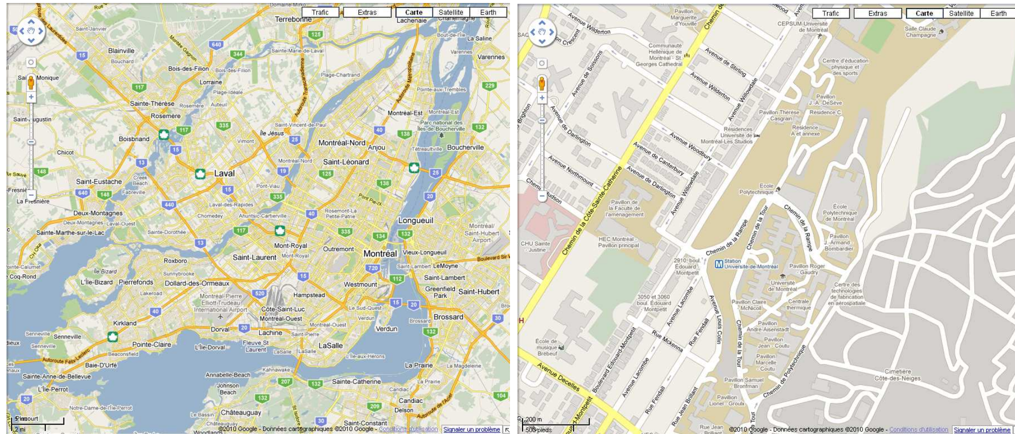

Figure 4.13 – Exemple du zoom sémantique. Le zoom sémantique est utilisé entre autres dans les cartes géographiques interactives de *Google* [31]. À gauche, on peut voir toute l'île de Montréal alors qu'à droite on voit un territoire moins vaste, mais des informations supplémentaires sont apparues comme les noms des rues, le nom des pavillons de l'Université de Montréal et les stations de métro.

4.1.7 Précisions sur la cohérence

La cohérence est effectivement importante dans les déplacements entre les différents niveaux de granularité comme c'est discuté dans la section 3.4. L'implantation de la cohérence réside dans le fait que les changements se font de façon graduelle alors que les utilisateurs sont en train de déplacer la caméra dans l'environnement. Cette impression d'être véritablement à l'intérieur de la représentation et que les mouvements affectent les informations disponibles contribue à accentuer l'effet de cohérence et sécurise les utilisateurs sur le fonctionnement de *VERSO* [83].

La cohérence est aussi présente dans le principe de la visualisation lui-même. En effet en descendant vers une granularité plus fine, les utilisateurs voient une vaste région dont la partie centrale se précise alors qu'ils perdent des informations sur les parties plus éloignées du logiciel. De cette façon, les utilisateurs peuvent facilement se situer et continuer une analyse qu'ils avaient commencée à un niveau de précision plus grand. Cette technique est évidemment préférable à deux visualisations complètement différentes pour

représenter les deux niveaux de granularité. Dans ce cas, les utilisateurs devraient changer de vue et se repositionner dans le secteur qui les intéresse. Cette activité demande de la concentration et du temps, mais demande surtout de se réapproprier la signification des éléments graphiques dans la nouvelle vue. La cohérence est aussi importante quand les utilisateurs passent d'un niveau de granularité fin à un niveau de granularité plus élevé. On part d'un élément qui reste le centre de notre visualisation et en changeant de niveau de granularité on obtient plus de contexte autour de notre objet d'étude.

4.2 Représentation des différents contextes

Nous nous reporterons encore une fois à la figure 4.1 pour discuter des changements de contexte. Dans ce cas, ces changements de contexte représentent un déplacement horizontal dans la figure. Les utilisateurs changent donc leur point de vue tout en restant au même niveau de granularité.

Les informations brutes utilisées pour la représentation des éléments sont basées sur les métriques. L'interface de visualisation a donc à sa disposition tout un éventail de métriques qui peuvent être représentées par les caractéristiques graphiques décrites dans la section 4.1. Les métriques principalement utilisées sont décrites à la section 5.2. Toutes ces métriques sont séparées en différentes catégories qui peuvent être interprétées comme des contextes différents. Par exemple, on peut calculer des métriques sur la qualité du logiciel du point de vue de la maintenance, des métriques calculées à partir des informations contenues dans les systèmes de contrôle de versions ou encore des métriques sur l'historique des bogues pour les éléments. En associant plusieurs métriques d'une même catégorie, nous obtenons des contextes. Ces contextes peuvent ensuite être utilisés pour faire des tâches d'analyse précises sur un des secteurs du logiciel. Les utilisateurs peuvent aussi naviguer d'un contexte à l'autre rapidement pour faire des analyses plus complexes.

Dans le but de faciliter la compréhension et l'adoption de *VERSO*, nous avons créé des contextes de base, c'est-à-dire des associations entre métriques et caractéristiques graphiques avec les métriques les plus intéressantes et les plus populaires. Ces contextes

prédéterminés sont la qualité, le contrôle de versions et la gestion des bogues. Ces contextes ont chacun un sous-ensemble de métriques présélectionnées représentant efficacement une catégorie pour des tâches typiques. D'un autre côté, les utilisateurs sont totalement libres de créer les associations désirées si leurs besoins sont différents. On peut donc, à l'aide d'une interface simple, modifier les associations existantes ou en créer de complètement nouvelles. Il n'est pas nécessaire de choisir les métriques dans la même catégorie si ce scénario ne répond pas à nos besoins. Pour le niveau de granularité des lignes de code, le principe est le même, sauf que les métriques sont représentées une seule à la fois alors la métrique représente le contexte à elle seule. Étant donné que le niveau de granularité des lignes de code est détaché et se trouve dans un autre éditeur, le contexte présenté n'est pas nécessairement le même que celui de la visualisation principale.

Les utilisateurs peuvent décider de changer de contexte, peu importe le niveau de granularité dans lequel ils se trouvent. Les associations entre métriques et caractéristiques graphiques seront modifiées, ce qui engendrera presque assurément des changements dans la représentation graphique. Ces changements sont apportés immédiatement sans ajouter d'images intermédiaires. Par contre, pour permettre aux utilisateurs de se retrouver, les différents contextes peuvent utiliser des codes de couleur différents. Par exemple, on peut utiliser la couleur variant du bleu au rouge quand on est dans le contexte de la qualité et changer le spectre de couleurs en allant du jaune au vert dans le contexte des systèmes de contrôle de versions. Ce changement permet d'avoir un indice visuel facile dans le cas où la tâche est interrompue ou pour rassurer les utilisateurs dans le fait que le changement de contexte a bel et bien eu lieu. Un exemple de deux contextes différents sur le même logiciel est montré à la figure 4.14.

4.2.1 Précisions sur la cohérence

Dans le cadre du changement de contexte, la cohérence est encore plus évidente que dans le cas du changement de granularité. Le changement de contexte se fait tout simplement en réaffectant de nouvelles métriques aux caractéristiques graphiques. Évidemment, la position de la caméra reste la même et le niveau de granularité reste inchangé.

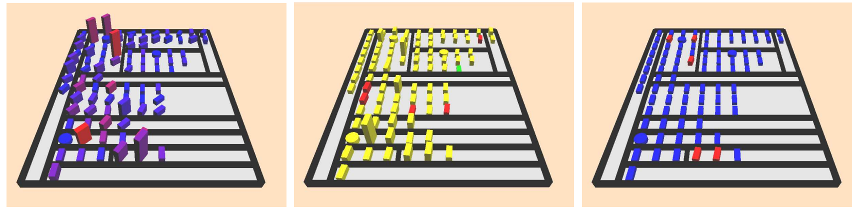

Figure 4.14 – Principe de justaposition entre les contextes au niveau de granularité de la classe. De gauche à droite, on voit le contexte de la qualité, celui des informations *SVN* et celui des bogues. Le code de couleur est changé dans le contexte *SVN* pour que les utilisateurs puissent bien différencier ce contexte.

L'important ici est que, puisque le placement n'est pas du tout modifié, on peut rester concentré sur les mêmes éléments que dans le contexte précédent. La charge cognitive pour passer d'un contexte à l'autre est donc faible et le placement constant et l'utilisation des mêmes caractéristiques graphiques en font une solution plus efficace que de posséder un outil pour chaque contexte différent. Un outil particulier à chaque contexte amènerait une spécialisation de la représentation, mais dans le contexte de l'analyse du logiciel, nous croyons que notre solution est préférable puisque plus simple et requérant un apprentissage moins lourd.

4.2.2 Évolution du logiciel

Il reste encore une dimension du cube de la figure 4.1 qui n'a pas été explorée et il s'agit de l'évolution du logiciel. Quand on se réfère à la métaphore du cube, il s'agit de la profondeur. En effet, il est possible d'explorer les différentes versions du logiciel à l'intérieur de tous les niveaux de granularité énoncés plus haut et aussi dans tous les contextes. En fait, les versions sont des photographies conservant les informations disponibles au moment où la version a été créée. On navigue à l'intérieur comme si on était en train de développer du code sauf que les représentations ne sont pas directement liées aux entités actuelles. Évidemment, il est impossible d'apporter des modifications aux anciennes versions du code puisqu'il deviendrait complexe ou tout simplement contre-intuitif de les refléter sur le code actuel.

Pour éviter toute confusion, la visualisation de l'historique des versions est présen-

tée dans une fenêtre à part contenant seulement les aspects de la visualisation sans être connectée à l'interface d'*Eclipse*. La représentation des versions est semblable à la représentation utilisée quand on s'intéresse à la version courante. On voit donc toutes les informations qu'on peut voir dans la version courante, on peut changer les associations de métriques et naviguer dans tous les niveaux de granularité. Tant que les informations étaient disponibles au moment de la création de la version, elles le seront au moment la visualisation. Dans les faits, les versions sont gardées dans des fichiers textes contenant tous les éléments à tous les niveaux et un vecteur de métriques précalculé pour chacun des éléments. On garde aussi des informations sur les relations entre les éléments et la hiérarchie. En fait, il s'agit du même processus utilisé lors de la sauvegarde de la version courante pour éviter que toutes les métriques soient recalculées (voir chapitre 5).

Pour l'instant, s'il y a des modifications apportées à la structure d'un programme (ajout, suppression, changement de nom d'élément), le placement utilisant le *Treemap* modifié sera recalculé en entier. Le calcul du *Treemap* est rapide et ne gêne pas les activités des programmeurs. Par contre, l'algorithme traditionnel du *Treemap* peut subir des réarrangements importants malgré de petits changements faits sur le code. La situation est un peu différente dans le cas de la représentation d'items occupant un espace discret sur le plan. Dans ce cas, les trous formés par cette tactique vont se remplir graduellement avant qu'il y ait modification de la structure. De plus, l'expansion possible du placement verticalement et horizontalement réduit le nombre de réarrangements importants. Il demeure néanmoins que les items tendent à changer de position dans la représentation et que ces changements arriveront encore plus fréquemment dans les premières versions d'un logiciel où il y a peu de classes à représenter.

Il aurait été possible d'éviter de recalculer complètement le placement en utilisant entre autres les techniques présentées dans l'article [53]. Ces techniques requièrent par contre d'avoir une connaissance sur toutes les versions d'un logiciel. Malheureusement, cette contrainte n'est pas respectée quand le but est de suivre les modifications d'un programme alors qu'elles sont effectuées en direct. De plus d'autres recherches ont utilisé des techniques basées sur l'influence des versions précédentes sur les placements subséquemment générés [14, 48]. Bien que ces techniques soient efficaces pour réduire

les modifications pour des approches basées sur les graphes, elles sont moins efficaces quand elles doivent s'appliquer au principe du *Treemap*.

Dans ces circonstances, le fait de recalculer le placement pour les modifications est une solution qui permet de s'adapter aux modifications en direct et qui évite d'avoir des modifications majeures trop fréquentes. L'étude d'une technique spécifique pour représenter les passages efficaces d'une version à une autre n'étant pas l'objectif principal de cette thèse, nous considérons l'application de la technique actuelle comme satisfaisante et réservons les améliorations à ce problème moins crucial pour des travaux futurs.

Pour simplifier le processus, les versions sont créées par les utilisateurs pour être interprétées par la suite par eux-mêmes ou d'autres utilisateurs. Une fois les versions créées, les utilisateurs accèdent à ces dernières et peuvent naviguer d'une version à l'autre en visualisant la prochaine version ou la version précédente. Il est aussi possible de faire de plus grands sauts en choisissant une version en particulier. Plus de détails sur la représentation des versions ainsi que des réflexions sur d'autres techniques pour représenter le temps dans le logiciel sont discutés dans notre article [53] et dans notre mémoire de maîtrise [50].

4.2.2.1 Précision sur la cohérence

Pour ce qui est de la cohérence dans l'évolution du logiciel, elle est encore une fois basée sur le fait que les informations elles-mêmes sont cohérentes. En effet, chacune des versions est construite par-dessus une version déjà existante. D'une version à l'autre, des éléments vont apparaître, disparaître ou seront modifiés. Par contre, la majorité des éléments resteront inchangés. Ce phénomène est encore plus vrai quand les versions sont créées de façon régulière et rapprochée. Le peu de changements entre les versions aide les utilisateurs à se concentrer sur les changements tout en gardant la connaissance du contexte. Les petits changements attirent l'oeil et il n'y a pas d'énergie consacrée à l'interprétation de la portion du logiciel qui est restée la même. De plus, lors des changements de versions, les deux autres coordonnées du cube restent les mêmes, c'est-à-dire le niveau de granularité et le contexte, et donc nous n'avons pas besoin de passer du temps à nous réapproprier la signification des caractéristiques graphiques. L'angle de vue de la

caméra reste aussi le même, nous n’avons donc pas à nous repositionner non plus.

CHAPITRE 5

INTÉGRATION DANS L'ENVIRONNEMENT DE DÉVELOPPEMENT

Un des ajouts les plus importants de cette thèse par rapport à l'état de l'art est l'intégration de la visualisation dans un environnement de développement pour permettre aux utilisateurs de voir directement plusieurs impacts dus aux changements qu'ils apportent au code. Ici, l'intégration est complète. Il ne s'agit pas seulement de rendre un outil disponible pour les utilisateurs, mais bien d'avoir une connexion importante entre la visualisation et l'environnement de développement et même que la visualisation soit directement une partie de l'environnement. Le prototype de l'intégration se fait à l'intérieur d'*Eclipse*.

5.1 Calcul des métriques dans *Eclipse*

Pour être en mesure de montrer des informations visuelles aux utilisateurs, il y a une première étape cruciale qui est celle de traiter les informations brutes pour qu'elles soient représentatives et utiles dans une analyse. De façon classique, les outils de visualisation ou d'analyse automatique utilisent des données générées au préalable. Ce principe est aussi inscrit dans la mouvance de faire les analyses dans une phase à part, souvent après que le produit a été déployé pour évaluer la qualité d'un produit fini. Dans ce cas, il est plus pertinent et beaucoup plus simple de récolter des données sur un produit à un moment fixé dans le temps. De cette façon, chacune des visualisations donnera assurément les mêmes résultats pour un même échantillon de données. Par contre, cette façon de voir les choses n'est pas applicable à une intégration d'un système de visualisation. Nos besoins impliquent que les données pour l'analyse soient traitées pendant que les utilisateurs effectuent leurs changements justement pour leur permettre de voir à quel point les changements sont pertinents.

Pour permettre aux utilisateurs de voir les changements aussitôt qu'ils sont effectués, il faut évidemment calculer les nouvelles informations en temps réel. En effet,

dans *VERSO*, à chaque fois que les utilisateurs sauvegardent les modifications qu'ils apportent à un logiciel, *VERSO* recalcule les métriques et les sauvegarde dans un modèle représentant le logiciel selon une série de métriques jugées pertinentes à travers différents domaines, dont la qualité, les informations sur le contrôle de versions, les bogues, etc. En réalité, ce modèle interne est composé de toutes les entités d'un logiciel, de leur hiérarchie, de leurs relations et d'un vecteur de métriques associé à chaque élément. Ce modèle est générique et peut être utilisé par n'importe quel système automatique ou semi-automatique d'analyse comme une visualisation de logiciel. D'ailleurs, *VERSO* possède déjà une manière de stocker les métriques dans un fichier texte où elles peuvent être analysées par un humain ou un ordinateur. De plus, de par sa nature, le modèle est facilement extensible, car il s'agit seulement d'ajouter de nouvelles métriques au vecteur.

Le calcul nécessaire pour obtenir les métriques est souvent simple, mais, pour certaines métriques, le calcul est plus complexe et demande du temps. C'est entre autres le cas de la métrique de couplage *CBO* [12] ($O(N^2)$) qui se doit de parcourir tous les éléments et tous les liens vers d'autres éléments pour chacune des classes. C'est pourquoi certaines stratégies doivent être adoptées pour réduire toute latence des utilisateurs dans leur démarche d'analyse.

Dans un premier temps, seulement la portion modifiée est traitée. Le calcul des métriques est donc incrémental tout comme les modifications du code qui s'ajoutent toujours sur une base déjà présente. Une autre stratégie est d'effectuer certains calculs en arrière-plan dans un autre fil d'exécution, entre autres, les calculs relatifs à l'accès des systèmes de contrôle de versions, car l'accès aux bases de données externes à travers internet est souvent trop lent. Ceci permet aux utilisateurs de continuer à travailler sans affecter leurs tâches directement. Souvent, une fois qu'ils sont prêts à regarder les résultats, la visualisation est déjà mise à jour. Sinon, elle s'adapte au fur et à mesure que les informations deviennent disponibles. Cette stratégie est aussi très utile quand les utilisateurs téléchargent un nouveau logiciel d'une taille appréciable et qu'ils veulent calculer les données tout en commençant à travailler.

Pour arriver à calculer des informations précises et obtenir ces informations dans des temps assez courts pour que les utilisateurs ne soient pas gênés, nous utilisons le

modèle interne d'*Eclipse*. En effet, *Eclipse* fait déjà une grande partie du travail pour arriver à gérer ses propres outils de référence à l'intérieur du code et la compilation automatique des programmes. Ainsi, *Eclipse* permet la complétion automatique, l'accès à une définition, ou encore l'accès aux appels d'une méthode à partir de sa définition. De plus, par différentes librairies, *Eclipse* donne accès à ces informations et à l'arbre syntaxique des logiciels. Ceci donne deux avantages marqués pour l'analyse statique dont nous avons besoin pour calculer les métriques. Dans un premier temps, les programmes *Java* sont déjà lus et interprétés pour nous et l'accès aux sous-sections de l'arbre syntaxique est facilité par des outils de parcours de graphes intégrés au modèle d'*Eclipse*. Dans un deuxième temps, une inférence de type statique est déjà effectuée et permet d'avoir accès facilement à tous les éléments dans la hiérarchie. Ceci représente en effet une économie considérable de temps et évite de refaire des outils complexes alors qu'ils ont déjà fait leurs preuves. De plus, les outils fournis par le noyau d'*Eclipse* permettent de développer une architecture qui est facilement extensible. L'accès au code étant basé sur le patron de conception *visiteur* [30] sur l'arbre syntaxique, l'ajout d'une nouvelle métrique consiste uniquement à ajouter des *visiteurs* et à faire le décompte de différents éléments visités. De plus, le mécanisme de sauvegarde d'*Eclipse* peut être utilisé pour recalculer les nouvelles métriques. En effet, la section sauvegardée peut être « attrapée » par une méthode et il est possible de recalculer les métriques seulement pour la section qui a été modifiée.

5.2 Description des métriques calculées

Notre approche est en mesure de visualiser toutes sortes d'informations décrites sous forme de métriques. Au besoin, les utilisateurs peuvent ajouter des métriques qui correspondent à leurs besoins en créant de nouveaux contextes, ou en étendant les contextes existants. Par contre, il existe toute une série de métriques que nous calculons déjà et que nous jugeons pertinentes pour faire des analyses et répondre à des questions typiques durant l'étude d'un logiciel. Ces métriques, surtout dans le cas du contexte de la qualité, ont été définies dès les débuts de l'étude de la qualité des programmes écrits dans un

langage orienté-objet, et ont été exploitées à maintes reprises dans des analyses et des expériences. Bien qu'elles demeurent d'actualité, elles ont été critiquées quant à leur applicabilité [1]. Le choix précis des métriques n'est pas au coeur des préoccupations de cette thèse et il est facile pour un programmeur d'utiliser les métriques qui correspondent le mieux à l'analyse à effectuer en modifiant les contextes. Nous présentons donc ces métriques à titre d'exemples de ce qu'il est possible d'accomplir avec *VERSO*. Les tableaux 5.I, 5.II et 5.III synthétisent les métriques principales et leur association par défaut avec les caractéristiques graphiques dans les vues prédéfinies. Les prochaines sous-sections donnent une description des métriques et de leur utilité potentielle pour répondre à des questions ou faire des analyses plus poussées.

Propriété graphique	Qualité
Couleur du paquetage	Couplage (Paquetage)
Hauteur du paquetage	Complexité
Couleur de la classe	Couplage (Classe)
Hauteur de la classe	Complexité (Classe)
Rotation de la classe	Cohésion (Classe) ou Héritage (Classe)
Couleur de la méthode	Couplage (Méthode)
Hauteur de la méthode	Complexité (Méthode)
Rotation de la méthode	Cohésion (Méthode) ou Héritage (Méthode)
Couleur de la ligne	Profondeur,Type et Destination des appels

Tableau 5.I – Associations principales entre les métriques et les caractéristiques graphiques pour les niveaux de granularité et le contexte de la qualité.

5.2.1 Couplage

Le couplage dans le cas de *VERSO* est le nombre d'entités différentes appelées par une entité donnée. Cette mesure est similaire à CBO [12], mais à sens unique. Dans le cas des classes, on compte le nombre de classes différentes appelées et dans le cas des méthodes on compte le nombre de méthodes différentes appelées incluant celles déclarées dans la même classe. Finalement, pour les paquetages, on décompte aussi le nombre de classes différentes externes puisque la hiérarchie de paquetages peut donner lieu à des nombres de classes différents par paquetage dépendamment des styles de logiciels. Par

Propriété graphique	Contrôle de versions
Couleur du paquetage	Auteur principal (Paquetage), Date
Hauteur du paquetage	Nombre de *commits* (Paquetage)
Couleur de la classe	Auteur principal (Classe)
Hauteur de la classe	Nombre de *commits* (Classe)
Rotation de la classe	Nombre d'auteurs, Date
Couleur de la méthode	Auteur principal (Méthode)
Hauteur de la méthode	Nombre de *commits* (Méthode)
Rotation de la méthode	Date du dernier changement
Couleur de la ligne	auteur, date

Tableau 5.II – Associations principales entre les métriques et les caractéristiques graphiques pour les niveaux de granularité et le contexte des systèmes de contrôle de versions.

Propriété graphique	Bogues
Couleur du paquetage	programmeur principal en charge des bogues
Hauteur du paquetage	Nombre de bogues (Paquetage)
Couleur de la classe	Programmeur principal en charge des bogues
Hauteur de la classe	Nombre de bogues (Classe)
Rotation de la classe	Nombre de bogues ouverts (Classe)
Couleur de la méthode	Programmeur principal en charge des bogues
Hauteur de la méthode	Nombre de bogues (Méthode)
Rotation de la méthode	Nombre de bogues ouverts (Méthode)
Couleur de la ligne	pas associée pour l'instant

Tableau 5.III – Associations principales entre les métriques et les caractéristiques graphiques pour les niveaux de granularité et le contexte des bogues.

contre, il ne s'agit pas d'une sommation des valeurs de couplage des classes qui composent le paquetage. On tient seulement compte des appels vers des classes différentes et celles qui sont présentes plusieurs fois comptent seulement pour une unité dans la métrique. Les valeurs des métriques de couplage pour les paquetages sont donc un peu plus petites que celles obtenues en faisant tout simplement la somme des valeurs de couplage des classes sous-jacentes.

Le couplage permet de voir si une entité communique beaucoup avec l'extérieur. Plus c'est le cas, plus des changements apportés à cette entité ou à ses voisines pour-

ront engendrer des problèmes ou encore d'autres modifications qui nécessitent un effort supplémentaire.

5.2.2 Complexité

La complexité est généralement très dépendante de la taille. La complexité de la méthode est calculée à l'aide de complexité cyclomatique de McCabe [63]. Pour les classes, il s'agit d'une somme des complexités des méthodes, ce qui revient à calculer WMC (*Weighted Method per Class*) [12] en utilisant McCabe pour le poids. La complexité des paquetages n'est qu'une somme provenant des calculs des classes incluses dans le paquetage. La complexité permet d'évaluer, avec d'autres métriques, le niveau d'effort nécessaire pour comprendre une entité et la modifier.

5.2.3 Cohésion

La cohésion d'une classe est calculée à l'aide de la métrique LCOM5 (*Lack of Cohesion in Methods*) [9] qui évalue si les méthodes de la classe utilisent conjointement les attributs de classe ou non. Le calcul complet est disponible dans l'article original [9]. Pour la cohésion des méthodes, la métrique est inspirée de LCOM5 dans le sens où l'on évalue le pourcentage d'attributs de classe utilisés par une méthode. C'est en quelque sorte la participation de la méthode à la cohésion de la classe. On prend ensuite l'inverse de la fraction pour avoir un manque de (*lack of*) cohésion comme dans le cas de la classe.

La cohésion sert à déterminer si une entité accomplit une et une seule tâche. Si une entité accomplit plusieurs tâches à la fois, elle devrait probablement être scindée. Il sera difficile de retrouver les entités où il faut ajouter des instructions ou encore corriger un bogue dans le cas où les rôles sont mal définis.

5.2.4 Héritage

Il s'agit d'informations sur les liens d'héritage entre les entités. Dans le cas de la classe, on utilise DIT (*Depth in Inheritance Tree*) [12] qui est la longueur du chemin pour atteindre la racine depuis une classe donnée dans l'arbre d'héritage. En *Java*, on est en

présence d'un parent unique qui est identifié à l'aide du mot-clé *extends*. La définition de l'héritage pour les méthodes s'inspire beaucoup de DIT. On utilise plutôt la redéfinition des méthodes au lieu de prendre le parent de la classe dans ce cas.

Les connaissances sur l'héritage sont importantes lors de la modification des éléments. Un élément haut dans l'arbre d'héritage aura tendance à impacter un spectre beaucoup plus large qu'une entité très spécialisée au bas de l'arbre d'héritage.

5.2.5 Auteur principal

L'auteur principal est déterminé essentiellement par le nombre de *commits* [1]. Si un auteur a soumis le plus grand nombre de *commits* pour une entité (méthode, classe, paquetage), il est réputé être l'auteur principal de cette entité. Dans le cas des paquetages, il s'agit évidemment d'une somme. Pour le cas d'une ligne de code, c'est la dernière personne qui a soumis un *commit* pour cette ligne dans le système de contrôle de versions qui en sera l'auteur principal. Bien que cette façon de calculer ne pointe pas toujours vers la personne la plus compétente sur une entité donnée, elle représente en général une très bonne approximation. Il s'agit d'une valeur nominale contrairement à plusieurs autres métriques.

L'auteur principal est utile pour identifier la personne responsable d'une entité dans le but de lui demander des précisions et ainsi réduire le temps passé à essayer de comprendre son fonctionnement.

5.2.6 Programmeur principal en charge des bogues

Le programmeur principal en charge des bogues est la personne attitrée à déboguer le plus grand nombre de bogues ouverts sur une entité. C'est utile quand on a besoin de consulter un collègue sur l'avancement de la solution à un problème d'une entité qu'on désire modifier ou utiliser.

[1] Forme de validation permettant d'envoyer les modifications d'un texte (principalement code) sur le serveur dans un logiciel de contrôle de versions.

5.2.7 Nombre de bogues

Il s'agit du nombre de bogues déclarés pour une entité donnée. Dans le cas des bogues ouverts, on compte seulement les bogues qui sont encore non résolus au moment d'évaluer le logiciel. C'est important pour connaître les entités qui causent le plus de problèmes dans un logiciel.

5.2.8 Nombre de *commits*

Il s'agit du nombre de *commits* indépendants pour une entité dans le logiciel. Similairement à d'autres métriques, au niveau de granularité des paquetages, un *commit* impactant plusieurs classes dans un même paquetage sera compté une seule fois. Pour le niveau des méthodes, il faut qu'au moins une ligne de la méthode ait été modifiée pour que le *commit* soit comptabilisé.

Le nombre de *commits* représente l'activité qu'il y a eu sur une entité. Une entité qui change beaucoup peut introduire des problèmes dans les autres parties du logiciel. En partenariat avec les informations sur le couplage, ces classes sont à surveiller par un programmeur qui les utilise, pour ne pas se retrouver avec des parties incompatibles.

5.2.9 Date

Cette métrique représente la date du dernier *commit* sur une entité. Alternativement, la date peut être remplacée par le numéro attribué au *commit* (ces numéros sont distribués dans l'ordre d'apparition des *commits*). La date peut s'appliquer à tous les niveaux de granularité y compris les lignes de code. Elle est utile pour savoir si des éléments sont restés inchangés depuis longtemps et qu'ils sont maintenant peut-être incompatibles avec le reste du logiciel, ou encore si un changement récent a pu introduire une faille dans le logiciel.

5.2.10 Métriques de lignes de code

Les métriques de lignes de code sont traitées différemment des autres métriques dans le sens où une seule d'entre elles peut être montrée à la fois. La profondeur est le niveau

d'imbrication des instructions et est considérée comme une forme de complexité. Le type donne des informations sur la nature des instructions présentes sur la ligne de code, par exemple, s'il s'agit d'un appel de méthode, d'une condition ou encore d'une déclaration de variable. La destination des appels se rapporte plus au couplage et détermine si les appels sont effectués dans la même classe, dans une classe d'un même paquetage, dans une classe d'un autre paquetage, etc. La date et l'auteur se comportent comme décrits dans les niveaux de granularité plus élevés.

Une vue sur les lignes de code nous permet de décortiquer les problèmes identifiés par des agencements de métriques des niveaux de granularité plus élevés et permet de faire des actions directes et pointues.

5.3 Incorporation de la visualisation

Auparavant, la fenêtre de visualisation de *VERSO* était une entité en soi et était utilisée dans une analyse post-développement. On la démarrait donc comme une application normale et la fenêtre pouvait être ajustée selon les besoins des utilisateurs, mais l'utilisation typique incitait les analystes à utiliser tout l'écran. Pour que l'intégration de *VERSO* dans *Eclipse* soit intéressante, la solution simple de démarrer une application qui communique avec la base de données d'*Eclipse* ne convient pas. En effet, avec cette solution, même si les données sont transférées efficacement, les utilisateurs doivent changer de fenêtre constamment pour voir les informations dans *VERSO* et consulter les autres éléments de l'IDE *Eclipse* pour faire des modifications au code par exemple.

La fenêtre de *VERSO* est donc ajoutée à même l'interface d'*Eclipse*. Pour que la visualisation soit intéressante, il faut qu'elle ait une grandeur acceptable. C'est la raison pour laquelle nous avons choisi de placer la fenêtre de *VERSO* au même endroit où on retrouve les éditeurs dans lesquels le code est écrit (voir figure 5.1 pour un exemple). En fait, la fenêtre de *VERSO* est implantée exactement comme un éditeur et il est mentionné dans la section 5.4 qu'elle a effectivement des fonctions d'édition. Les utilisateurs peuvent déplacer ou ajuster cet espace selon leurs besoins. La visualisation des lignes de code se comporte exactement de la même façon et on peut même avoir plusieurs onglets

de ce type pour visualiser plusieurs fichiers.

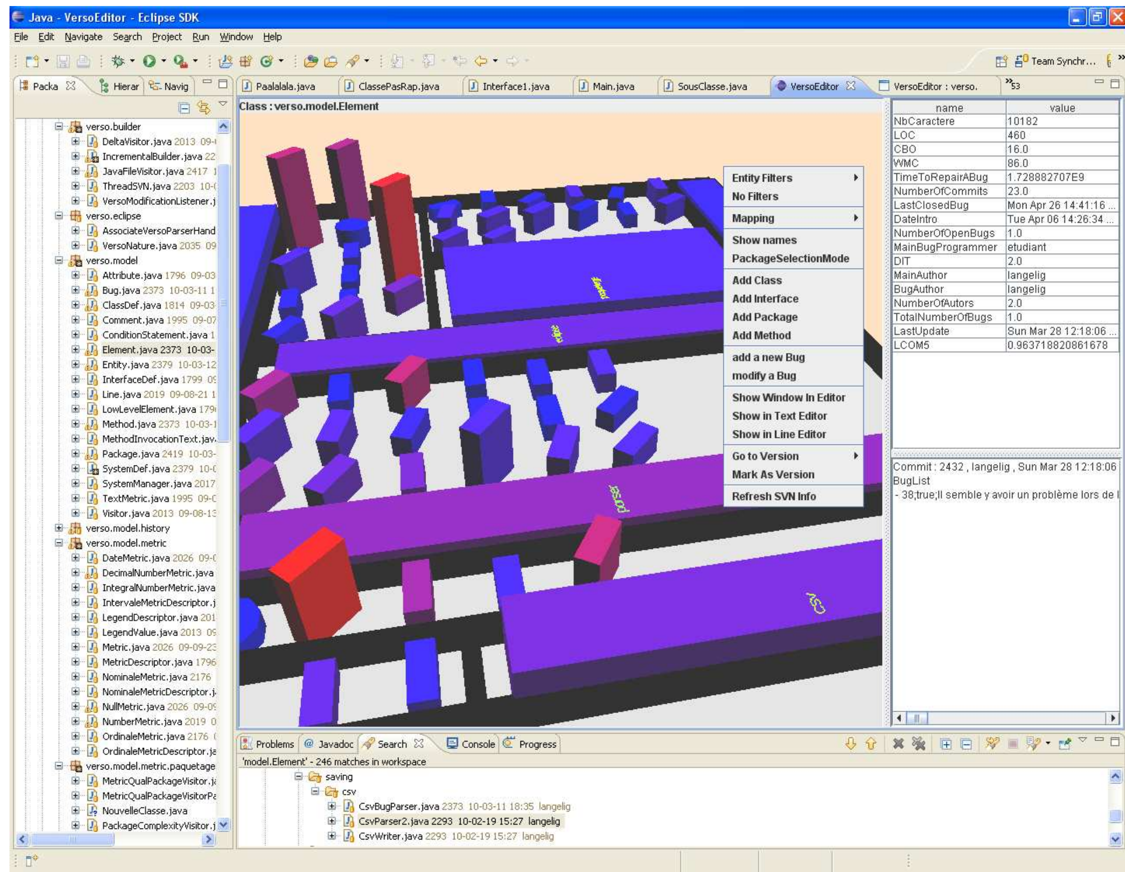

Figure 5.1 – Exemple de *VERSO* intégré dans *Eclipse*. Cette figure montre comment *VERSO* est intégré à l'environnement de développement et on peut voir les ramifications entre les deux interfaces.

5.4 Interface graphique et ramifications avec *Eclipse*

L'interface graphique de *VERSO* est principalement composée d'un canevas où apparaît la visualisation (voir figure 5.1). Dans le haut de ce canevas se trouve un champ où est écrit le nom de l'élément sélectionné, que ce soit une classe, un paquetage ou une méthode. Il s'agit d'une manière simple d'obtenir le nom des éléments sans encombrer la représentation en trois dimensions de l'environnement. La sélection de l'élément se fait en cliquant sur la fenêtre en deux dimensions et *VERSO* détermine l'élément sélec-

tionné en projetant le clic en trois dimensions. Il y a une autre section sur le côté droit de la visualisation, divisée en deux parties. Dans la partie du haut, on trouve une liste de toutes les métriques relatives à un élément avec leurs valeurs correspondantes. Il s'agit d'une façon intéressante de pallier le manque de précision de la visualisation ou d'avoir plus d'informations sur les valeurs extrêmes. La deuxième partie se trouve au bas de ce panneau et donne des informations sur les *commits* ainsi que sur les bogues. Ce panneau est bien sûr relié à l'élément qui a été sélectionné par les utilisateurs et montre les informations spécifiques à cet élément. Un exemple de l'interface est montré à la figure 5.1.

En plus de ces informations et de l'interaction qui est faite à partir de la sélection, *VERSO* utilise aussi d'autres fonctions de l'interface graphique pour permettre l'interaction dans le monde en trois dimensions. En sélectionnant l'écran, il est évidemment possible de se déplacer comme indiqué dans la section 4.1.6. Des menus permettent aussi d'afficher les noms à même la représentation en trois dimensions, de changer le mode de sélection des classes pour ouvrir et fermer les paquetages et d'imposer la visualisation d'un certain niveau de granularité ou d'un certain niveau de paquetage. D'autres menus permettent de modifier le contexte de la visualisation comme mentionné dans le chapitre 4 sur la visualisation. C'est aussi à même ces menus qu'il est possible de modifier ou de créer de nouveaux contextes. Un menu est présent pour ouvrir la visualisation des lignes de code sur un élément sélectionné et pour repositionner la caméra à la position de départ si les utilisateurs se perdent dans la représentation.

Un des avantages importants de l'intégration de *VERSO* dans *Eclipse* est qu'il puisse y avoir une interaction entre les interfaces des deux outils. Ceci est encore une fois reflété dans la sélection des éléments. En effet, à chaque fois que des utilisateurs cliquent sur un élément classe, méthode ou paquetage dans *VERSO*, cet élément est sélectionné dans l'explorateur de paquetages d'*Eclipse*. De plus, comme mentionné plus tôt, tout changement apporté au code est répercuté par la visualisation aussitôt que les utilisateurs sauvegardent leurs données ou qu'une sauvegarde automatique est lancée dans l'environnement de développement. De plus, il est possible, à partir des menus et des fonctionnalités d'*Eclipse*, d'associer un projet à *VERSO* pour l'analyse des métriques en temps réel et aussi d'ouvrir la fenêtre de la visualisation elle-même. En plus d'avoir cette

communication directe entre les deux outils, il existe aussi une collaboration indirecte du fait qu'ils cohabitent dans les mêmes fenêtres. Les utilisateurs peuvent alors compléter leur analyse des métriques dans *VERSO* avec une recherche textuelle à l'aide d'*Eclipse* ou le suivi d'un lien pour obtenir la définition d'un appel de méthode. Il ne s'agit donc pas d'opposer la façon de travailler de l'environnement de développement *Eclipse* avec celle de *VERSO*, mais bien d'avoir deux outils qui se complètent.

5.5 Ajouts et suppressions d'éléments

L'intégration de *VERSO* dans *Eclipse* ne permet pas seulement de faire de l'analyse et d'interagir avec des informations déjà calculées. On peut aussi créer des éléments avec l'interface de *VERSO* comme si les éléments avaient été créés manuellement ou par les aides automatiques d'*Eclipse*. Il est donc possible de créer des paquetages, des classes et des méthodes à même la vue graphique sans sortir de l'espace de la visualisation. Aussitôt que les éléments sont créés, ils apparaissent à l'écran et on voit leurs caractéristiques.

La façon d'implanter ce comportement est d'utiliser les menus de *VERSO* pour lancer la création des objets. Ensuite l'aide automatique d'*Eclipse* apparaît pour la création des classes. Par contre, plusieurs champs sont déjà remplis en fonction des éléments qui sont sélectionnés dans la représentation graphique. Par exemple, une nouvelle classe sera placée dans le bon paquetage automatiquement. Il ne restera que le nom à ajouter et les dépendances hiérarchiques. La création des méthodes consiste en un formulaire indépendant des fonctionnalités d'*Eclipse* qui permet d'insérer les méthodes dans la classe sélectionnée. On peut aussi ajouter des paramètres et des types de retour automatiquement. Il aurait été possible d'utiliser encore plus d'aspects visuels dans la création des éléments, par exemple, une barre d'outils qui répond à des événements de glisser-déplacer pour ajouter les éléments dans les bonnes composantes. Il n'est pas exclu d'ajouter ces fonctionnalités dans des travaux futurs. Dans tous les cas, le code doit être écrit dans les éditeurs textes traditionnels d'*Eclipse*, car seuls les squelettes de classe ou de méthode sont créés. Le niveau de granularité du code lui-même n'est pas traité dans la création

automatique, car moins intéressant du point de vue de l'analyse du logiciel et de ses différentes métriques.

On peut aussi ajouter des informations sur des bogues dans la base de données de *VERSO* (qui seront éventuellement à corriger). Le principe est semblable aux autres éléments. Il est possible de faire des sélections et un formulaire sera en partie rempli automatiquement. Évidemment, une des parties les plus importantes de la déclaration d'un bogue est le commentaire qui le décrit en détail. Il n'existe pas de façon visuelle de créer ce commentaire. Dans le formulaire, il est aussi possible et même souhaitable d'ajouter des éléments reliés au bogue, c'est-à-dire des éléments qui peuvent avoir causé le problème ou des éléments à modifier. Ces éléments sont très pertinents pour ceux qui veulent corriger les bogues et nous permettent de calculer plusieurs métriques qui pourront être visualisées par la suite, par exemple, le nombre de bogues reliés à une entité donnée. Ce système de déclaration de bogue a été construit principalement parce que les outils généralement utilisés comme *BugZilla* [11] n'ont pas une structure ni une déclaration qui s'intègrent bien à notre besoin de calcul de métriques, où l'entité principale est l'élément (classe, méthode ou paquetage). Encore une fois, quand le bogue est ajouté ou modifié et que les utilisateurs sont dans la vue correspondant au contexte des bogues, ils voient les changements reflétés instantanément.

CHAPITRE 6

ÉVALUATION

Ce chapitre présente une évaluation de notre approche. On y discute aussi de différents types d'évaluation des systèmes de visualisation. Cette comparaison nous permet de mieux situer l'évaluation de *VERSO* par rapport aux autres recherches et de mieux apprécier les résultats obtenus. Deux études distinctes sont présentées et nous concluons sur les apports de *VERSO*.

6.1 Contexte

La création d'un système de visualisation ou même d'un environnement pour l'aide à la programmation demande une évaluation particulière pour s'assurer que ce qui a été développé répond bien à des principes qui sont utiles dans la pratique. À cause de la nature de l'approche employée, l'évaluation doit être faite à partir des besoins des programmeurs susceptibles d'utiliser *VERSO*. L'évaluation empirique est primordiale pour s'assurer que le système répond bien à des critères d'efficacité, de performance et de précision puisqu'il n'est pas possible de faire une preuve mathématique ou encore de comparer les résultats avec un oracle, comme c'est le cas dans d'autres domaines de l'informatique.

Une des premières façons d'évaluer un système est de se poser des questions sur les fondements de l'outil. En effet, si les principes utilisés dans les interfaces de l'outil suivent des principes qui ont été démontrés ou évalués favorablement, de façon individuelle ou à l'intérieur d'autres systèmes, ces éléments, une fois regroupés, devraient être utiles pour les programmeurs. C'est le cas par exemple des principes de perception instantanée qui ont été rigoureusement évalués par Healey et Enns [40] et d'autres chercheurs à plusieurs reprises. Plusieurs autres principes comme les avantages de la présentation des informations dans un outil unique ou encore la présentation des informations en ligne ont été discutés en profondeur dans le chapitre 3. Le souci de cohésion dans

tous les axes d'études de *VERSO* contribue aussi à confirmer la qualité de l'approche et sa probable adoption.

L'évaluation des logiciels de visualisation et des recherches basées sur des interfaces utilisateurs n'est jamais facile. Sensalire *et al.* [88] se sont aussi intéressés aux démarches d'évaluation des outils de visualisation du logiciel. Le principal problème est que la solution évidente, c'est-à-dire une étude à long terme avec des professionnels attitrés à une tâche précise, directement comparable à un groupe témoin, est très coûteuse en terme de ressources et demande la participation sur une longue période d'un chercheur qui se doit d'être neutre. Même dans ces conditions idéales, plusieurs facteurs peuvent influencer la perception des sujets dans l'expérience ou encore plusieurs facteurs externes à l'étude peuvent influencer les résultats. Les résultats recueillis sont ainsi souvent des données qualitatives basées sur la rétroaction des sujets. Il serait possible de recueillir des indicateurs quantitatifs en utilisant le temps pour compléter un projet ou encore ses coûts. Par contre, ces données sont trop dépendantes des différences entre les projets et il est impossible d'obtenir de telles conditions dans un contexte professionnel à long terme. Les études sont donc souvent basées sur une simulation ou encore sont simplement des études de cas sur une petite équipe sans avoir recours à un comparatif direct. Il existe quelques rares études avec des résultats quantitatifs, mais elles s'intéressent à un sous-ensemble précis d'une approche et démontrent l'efficacité de ce sous-ensemble uniquement, ou encore démontrent une propriété isolée et non pas l'efficacité générale du système. Des exemples de telles études sont donnés à la section 6.2.

Un autre problème qui survient dans l'évaluation des systèmes de visualisation et des interfaces usager est celui du manque de comparatif ou du flou de ce qui devrait être comparé à l'outil. Par exemple, est-ce qu'un outil de visualisation devrait être comparé à d'autres outils de visualisation, ou encore à une technique plus traditionnelle d'analyse du logiciel ? Est-ce qu'on doit le comparer avec des techniques automatiques ou des techniques manuelles ? Quand la visualisation présente des informations prétraitées comme c'est le cas des métriques avec *VERSO*, doit-on comparer les performances des utilisateurs avec des outils qui précalculent ces informations ou doit-on encore une fois comparer avec des outils de programmation traditionnels ou donner accès aux informa-

tions à travers d'autres formes d'outils ? Si l'on choisit de comparer avec d'autres outils de visualisation, c'est encore une fois compliqué. Souvent, les outils de visualisation avec lesquels il est possible de se comparer n'ont pas été évalués de façon rigoureuse non plus. De plus, la plupart des outils de visualisation sont très spécialisés et il est difficile de trouver des outils qui se comparent au nôtre, et surtout de trouver des questions ou des activités pour les sujets qui ne vont pas directement biaiser les résultats en faveur d'un outil ou d'un autre.

La forme d'évaluation la plus fréquente des systèmes de visualisation est la capacité d'un outil à découvrir des phénomènes lors de l'observation d'informations. Ces observations sont souvent faites par un des concepteurs des outils de visualisation. La deuxième technique la plus utilisée est de laisser des utilisateurs travailler ou faire une tâche précise à l'aide de l'outil et de leur demander leurs impressions par la suite. La principale raison pour laquelle les chercheurs utilisent cette façon de faire est le manque de temps. Pour avoir des études sérieuses et exhaustives, il faudrait que de tierces personnes s'intéressent à des outils existants.

6.2 Revue de quelques évaluations en visualisation

6.2.1 Théorie

Dans un premier temps, Munzner [67] propose un modèle imbriqué pour construire des visualisations et les valider par la suite. Munzner discute de menaces à la validité dans chacun des quatre niveaux qu'elle présente. Contrairement au sens généralement accepté pour les menaces à la validité, ces menaces ne concernent pas uniquement la validité d'une expérience, mais bien la validité de tout le processus de construction d'un outil de visualisation, incluant potentiellement l'évaluation. Son modèle d'analyse contient quatre étapes (caractérisation du problème, abstraction des données et des opérations, encodage et interactions, et algorithmique), et chacune des étapes englobe la prochaine. Un problème dans une étape entraînera inévitablement l'invalidation des étapes subséquentes.

Dans la première étape de caractérisation du problème, il faut s'assurer de répondre

à un besoin réel et de bien comprendre ce besoin. La validation se fait en interrogeant les utilisateurs potentiels et en observant le taux d'adoption de l'outil une fois complété. L'étape de l'abstraction des données et des opérations se valide en récoltant des informations anecdotiques de la part des utilisateurs du domaine et plus tard en observant des utilisateurs qui font des tâches qui correspondent à leurs besoins et qui ne sont pas imposées par l'évaluateur. L'étape de l'encodage et des interactions est celle qui fait l'objet du plus grand nombre d'articles en visualisation (si ce n'est pas l'entièreté des articles portant sur des outils). Il s'agit en fait de la visualisation elle-même et de la représentation graphique des informations. Cette fois-ci, on peut utiliser une étude avec des tâches contrôlées puisque l'étape précédente de la pertinence des opérations est supposée être valide. Souvent on veut chronométrer le temps requis pour les tâches et mesurer le nombre d'erreurs des utilisateurs. Une validation simple est aussi de vérifier si on ne viole aucun des grands principes de l'efficacité perceptuelle. On peut aussi analyser les images générées qualitativement ou parfois quantitativement en observant par exemple la présence d'occultations ou de liens entremêlés. L'étape des algorithmes ne requiert pas de sujets, il s'agit de voir si l'algorithme se comporte comme il a été pensé et s'il est suffisamment robuste et rapide pour les besoins de la visualisation. Munzner ne décourage pas le développement de recherches ne concernant pas toutes les étapes, mais précise que ces recherches devraient mentionner les hypothèses faites aux étapes précédentes. Curieusement, Munzner ne mentionne pas ou très peu l'idée de comparaison à travers son analyse, ce qui semble pourtant au coeur de la preuve de l'utilité d'une nouvelle approche.

Selon cette grille d'analyse, *VERSO* se situerait aux deuxième et troisième étapes. D'ailleurs, l'analyse du respect des principes de perception a déjà été faite et nous verrons plus loin dans ce chapitre que deux expériences ont été conduites, une qui réfère à des tâches précises dans un environnement contrôlé et une seconde qui consiste à observer des utilisateurs alors qu'ils font leur propre travail.

Sensalire *et al.* [88] discutent aussi des bases de l'évaluation des outils de visualisation, mais cette fois se concentrent sur la visualisation du logiciel. Dans ce cas, on met l'emphase sur les pièges à éviter et sur les solutions à des problèmes rencontrés lors de

l'élaboration et de l'exécution d'une expérience. On y discute des tâches, des participants, de l'outil choisi, de la durée que devrait avoir une expérience, etc. Par exemple, on mentionne que les participants devraient avoir une expérience semblable pour éviter de faire des post-traitements sur les résultats obtenus. Les conseils présentés fournissent une marche à suivre intéressante et applicable dans nos propres recherches.

6.2.2 Visualisation en dehors du logiciel

Dans leur recherche, Bresciani et Eppler [8] tentent de démontrer que la visualisation est bénéfique pour l'organisation des idées dans le domaine de la gestion. Cette étude est particulièrement intéressante puisque dans ce cas-ci, les auteurs de l'étude ne sont pas les auteurs des outils de visualisation. Ils sont donc moins biaisés dans leur interprétation des résultats. Par contre, les principes de visualisation sont beaucoup plus simples que ceux habituellement utilisés dans la visualisation du logiciel. Les auteurs ont fait une expérience avec trois catégories de groupes. La première catégorie contient les groupes de contrôle qui n'utilisent pas d'outil graphique, la deuxième catégorie est celle des groupes qui utilisent une visualisation sous-optimale, et la troisième catégorie regroupe ceux qui utilisent une visualisation optimale. Les deux visualisations sont choisies à l'aide d'un questionnaire sur les préférences de 116 gestionnaires. Dans ce cas, les termes *optimale* et *sous-optimale* utilisés par les auteurs pour qualifier la visualisation font référence aux résultats de ce questionnaire. Ils ont formé 26 groupes de 5 personnes pour un total de 131 sujets. Ils ont assigné chaque groupe à une catégorie et les groupes devaient échanger des idées, des stratégies et des expériences passées sur des questions reliées à la gestion. Ils ont ensuite évalué les groupes et les gestionnaires individuels sur leur productivité (nombre d'idées), la qualité des idées, l'apprentissage (se souvenir d'idées déjà présentées), la satisfaction et la participation. Ils ont extrait des données réelles quantitatives pour la productivité et l'apprentissage et des données qualitatives et subjectives pour les autres éléments évalués. Les expérimentateurs ont finalement découvert que les gestionnaires sont plus productifs avec une visualisation que sans support. Il n'y a pas de différence significative entre les visualisations optimales et celles sous-optimales, par contre. D'un autre côté, ces chiffres ne se sont pas reflétés dans les taux de satisfaction.

Ce fait s'explique parce que les utilisateurs n'avaient pas de comparatif et ne pouvaient donc pas relativiser leurs réponses.

Tory *et al.* [92] proposent une expérience pour déterminer si les utilisateurs retiennent mieux les informations présentées comme des points avec une couleur et une position en deux dimensions (nuage de points colorés) ou en ajoutant une analogie de géographie. Deux analogies sont comparées. Dans un premier temps, on utilise des contours pour représenter les hauteurs (cartes topographiques), et dans un deuxième temps, on présente les données carrément en trois dimensions avec la hauteur et la position associées aux données. Cette expérience est purement quantitative. On demande aux gens de regarder une série de huit images et de les mémoriser, ensuite on repasse une série de huit images et on demande aux sujets d'indiquer si l'image était présente dans la première série. On compte ensuite tout simplement les bonnes et les mauvaises réponses. Bien que la méthodologie soit intéressante, il n'est pas évident que la capacité de mémorisation soit directement corrélée avec la facilité d'utilisation et d'interprétation. Les auteurs ont découvert que la représentation simple par points colorés est meilleure du point de vue de la mémorisation, et qu'il n'y a pas de différences significatives entre les deux analogies géographiques. Un problème qui n'est pas mentionné par les auteurs est que les points en deux dimensions utilisent la couleur. Plusieurs points sont pâles et se fondent dans l'arrière-plan, alors que la couleur utilisée dans les autres visualisations est placée dans l'espace sous les points et non pas sur les points eux-mêmes. Le fait que moins de points attirent l'attention peut aussi simplifier la mémorisation.

Ware [94] utilise lui aussi une expérience perceptuelle pour évaluer si ses *textons quantitatifs* fonctionnent mieux en combinaison avec une couleur unidimensionnelle par rapport à l'utilisation de deux composantes de la couleur pour représenter des cartes bivariées. Pour ce faire, il demande à 14 sujets d'identifier les valeurs à un endroit donné pour les deux cartes superposées selon cinq schémas différents, dont trois différents *Q-tons* et deux schémas utilisant deux dimensions de la couleur. Il demande de faire 12 observations pour chacun des schémas pour un total de 60 observations. La distance avec les vraies valeurs est considérée, mais pas le temps de réponse. Les sujets ont pris en moyenne 8 secondes pour faire chaque observation. Les *Q-tons* ont été les plus efficaces

selon les résultats présentés.

6.2.3 Outils de développement

Fritz et Murphy [29] ont fait un travail de recherche très intéressant sur les questions typiques que se posent les développeurs. Ils ont aussi développé un outil qui, sans être une visualisation à proprement parler, est certainement une nouvelle interface usager permettant de répondre à ces questions. Dans un premier temps, ils ont interviewé onze professionnels en développement de logiciels pour connaître des questions que les développeurs se posent lors de la programmation. Ces questions ne sont pas directement liées à la syntaxe du code. Par exemple : quelle personne a travaillé sur cette section du code avant moi ou quelles fonctions du logiciel ont changé récemment ? Ils ont ainsi compilé 78 questions que les utilisateurs se posent généralement, mais seulement une faible quantité de ces questions ont été évoquées par plus d'un professionnel. Ils ont ensuite choisi huit de ces questions provenant de différents sous-domaines et ont demandé à 18 participants de trouver les réponses à ces questions en utilisant leur outil. Ils ont calculé le temps requis pour répondre à chaque question dans le but de prouver la facilité d'utilisation de leur outil. Ils ont aussi enregistré les actions des utilisateurs sur l'écran pour récupérer des informations anecdotiques sur la façon réelle d'utiliser l'outil. Ils ont finalement fait une entrevue avec les utilisateurs une fois l'expérience terminée pour amasser des données qualitatives.

Bragdon *et al.* [7] ont utilisé deux types d'expériences pour évaluer leur outil *Code Bubbles*. La première est une étude quantitative qui compte le nombre de méthodes qui sont visibles en même temps à l'aide de leur logiciel. Ils comparent ensuite avec le nombre de méthodes visibles pour l'environnement de développement *Eclipse*. Ils réussissent effectivement à afficher plus de méthodes en même temps à l'aide de *Code Bubbles* dans la grande majorité des cas. Ils montrent aussi que les sujets doivent faire moins d'opérations d'interface usager par méthode affichée. Bien que ces résultats soient quantitatifs, significatifs et facilement comparables avec les valeurs d'autres outils de développement, le lien entre ces améliorations et l'impact réel sur l'efficacité de développement dans leur environnement reste à démontrer. La deuxième expérience consiste

en des tâches contrôlées qui sont reliées aux fonctionnalités offertes par leur logiciel. Il y avait 6 activités composées de sous-tâches simples à effectuer, 23 professionnels ont participé à l'expérience et ils ont pris en moyenne 1,5 heure pour effectuer ces tâches. Les sujets devaient ensuite répondre à un questionnaire qualitatif en utilisant une échelle de 1 à 5 sur leur appréciation de l'outil pour chacune des activités. Dans des questions plus ouvertes, ils devaient mentionner pourquoi ils aimaient ou n'aimaient pas l'outil, donner des suggestions, et mentionner s'ils pensaient qu'il était envisageable d'utiliser l'outil dans leur travail. Les résultats concernant les appréciations sont très positifs.

Dagenais *et al.* [17] ont conduit une expérience pour déterminer les besoins des nouveaux arrivants dans une équipe travaillant sur un projet qui leur est inconnu. Cette recherche n'a pas de lien direct avec la visualisation si ce n'est que selon leurs conclusions, la visualisation serait utile pour une compréhension initiale d'un projet. Par contre, il y a des énoncés intéressants dans la façon dont l'expérience a été menée et au sujet de la discussion sur les difficultés rencontrées quand de nouvelles personnes s'intègrent à un projet. Il serait intéressant d'en tenir compte avant de commencer un projet de visualisation ou encore pour déterminer si l'on s'attaque aux vrais problèmes des utilisateurs que tente de résoudre la visualisation. Pour déterminer les besoins de cette clientèle, ils utilisent le principe de la théorie enracinée (traduction libre de *grounded theory*). Avec cette stratégie, on part avec des hypothèses initiales qu'il faut raffiner en cours de route à l'aide des données récoltées. Plutôt que d'utiliser des techniques empiriques classiques, on choisit des critères de plus en plus précis pour les participants dans le but qu'ils soient en mesure de répondre aux questions permettant de raffiner le modèle. La théorie est dite enracinée parce qu'elle part de l'empirique pour créer une théorie cohérente. Cette technique paraît idéale pour les phases exploratoires d'un projet de visualisation où il faut déterminer les problèmes à régler et les informations à représenter.

6.2.4 Visualisation du logiciel

Ogawa et Ma [71] présentent une étude sur *code swarm*. Au lieu d'utiliser une expérience plus traditionnelle avec des sujets, les auteurs ont plutôt choisi d'inspecter plusieurs vidéos pour y retrouver des informations pertinentes et de rendre leur outil dispo-

nible pour évaluer la réponse du public en laissant aussi les utilisateurs visualiser leurs propres réalisations. En publiant des vidéos qu'ils avaient conçues à l'aide de l'outil, des internautes ont aussi pu laisser des commentaires. Les auteurs ont ensuite analysé ces commentaires. L'analyse était qualitative plutôt que quantitative. Ils ont constaté que plusieurs vidéos créées par leur outil sont apparues sur le web. Il s'agit de données moins contrôlées, mais avec le nombre de personnes ayant utilisé leur outil efficacement, ils ont une meilleure idée de l'adoption par la communauté.

6.3 Expérience 1 : Étude de cas *in vitro*

Cette première expérience vise à évaluer si *VERSO* peut être bien compris en dehors du cercle restreint des personnes qui travaillent directement à sa conception. Les sujets de cette expérience ne connaissaient pas *VERSO*, mais avaient une connaissance préalable de la programmation et de la qualité du logiciel. Cette expérience comportait plusieurs sujets dans un univers contrôlé. Les tâches demandées étaient artificielles, bien qu'inspirées de situations réelles. Cette expérience visait principalement à évaluer la facilité d'apprentissage de *VERSO* et à tester l'appréciation des usagers face à *VERSO*. Puisque les sujets devaient effectuer des tâches non liées à l'accomplissement d'autres objectifs, comme un projet d'un cours ou un livrable industriel, ils ont pu se concentrer sur l'évaluation de *VERSO* et nous donner de bons conseils sur son amélioration.

6.3.1 Méthode

6.3.1.1 Les sujets

L'expérience demandait à 28 étudiants d'utiliser *VERSO* pour faire quelques tâches simples sur du code qu'ils ne connaissaient pas. Ils étaient séparés en dix équipes (huit équipes de trois personnes et deux équipes de deux personnes). Les participants devaient se consulter et s'entendre pour répondre aux questions. Les participants provenaient d'une classe de troisième année du baccalauréat en informatique et étaient inscrits dans un cours traitant de la qualité du logiciel. Ces participants étaient de niveaux de compétence variés en programmation, mais la très grande majorité avait suivi le même cursus

académique et était sur le point d'accéder au marché du travail ou de poursuivre leurs études au niveau supérieur. Toutes les équipes devaient remplir les mêmes tâches dans le même contexte, il n'y avait donc pas de groupe test pour cette expérience. Les sujets recevaient 5% supplémentaire dans leur note finale du cours en échange de leur participation à l'expérience. Les points étaient donnés systématiquement et n'étaient en aucun cas liés avec les réponses données ou les résultats obtenus lors de l'expérience. L'activité n'était pas obligatoire.

6.3.1.2 Le temps

L'expérience a duré deux heures et de ces deux heures, une période d'initiation à *VERSO* d'une durée d'environ 15 minutes permettait aux participants d'apprendre les bases de l'outil et surtout de se familiariser avec son interface graphique. Le temps pour répondre aux tâches individuelles n'était pas chronométré ni pris en compte dans les résultats. Par contre, la réussite des tâches dans le temps imparti est considérée dans les résultats de l'expérience.

6.3.1.3 Le lieu et les ordinateurs

Lors de l'expérience, les participants étaient regroupés dans un laboratoire habituellement réservé aux étudiants des cycles supérieurs. La salle était dédiée pour la durée de l'expérience. Les ordinateurs utilisés sont des machines conventionnelles relativement récentes, mais construites pour une charge de travail de bureau, et non pas pour obtenir des graphiques performants ou fournir une puissance de calcul hors du commun. Tous les ordinateurs n'étaient pas exactement équivalents, mais très semblables en matière de performances. Tous les ordinateurs avaient été démarrés préalablement à l'expérience et *VERSO* y avait été installé la veille. Le contexte de travail contenait l'outil de développement *Eclipse* augmenté de l'outil de visualisation *VERSO*. Cet environnement était déjà démarré lors de l'arrivée des sujets et les métriques avaient été précalculées pour permettre de commencer à travailler tout de suite et éviter des surprises reliées à l'ordinateur utilisé ou d'attendre un calcul des métriques ralenti par une panne.

6.3.1.4 Les questions

Un questionnaire à remplir sur papier a été utilisé à la fois pour vérifier si les sujets étaient en mesure d'effectuer les tâches et pour recueillir leurs commentaires sur *VERSO* et l'expérience. Il y avait un questionnaire par équipe sur le lieu de travail. La première partie du questionnaire comportait les tâches à accomplir, qui étaient faites directement sur une sous-partie de *VERSO* qui concerne l'analyse du logiciel pour le calcul des métriques. Il s'agit donc d'une auto-analyse de *VERSO*.

Dans un premier temps, les participants devaient répondre à quatre questions d'analyse au sujet d'informations indispensables à connaître avant de commencer une tâche ou sur des menaces probables à la qualité du logiciel du point de vue de la maintenance. Nous n'attendions pas une réponse exacte et unique pour chaque question, mais bien un raisonnement. Les questions de cette première partie sont présentées à la figure 6.1

1. Quel est le paquetage le plus important ? Pourquoi ?
2. Il y a un paquetage obsolète dans le système. Lequel ? Pourquoi ?
3. Sur quelle classe y a-t-il eu le plus de travail selon vous ? Pourquoi ?
4. Si vous aviez la possibilité de faire le *refactoring* de deux ou trois classes seulement, lesquelles choisiriez-vous ? Pourquoi ?

Figure 6.1 – Questions expérience 1, partie 1. Questions concernant l'analyse du logiciel.

Ensuite, il y avait deux bogues de nature sémantique à corriger. Les sujets devaient entre autres rechercher les bogues avant de les corriger à partir d'un descriptif de la situation problématique. Les questions suivantes portaient sur un ajout d'une petite fonctionnalité et sur un *refactoring* plus complet d'une sous-section du logiciel. Pour chaque question, les participants devaient écrire la réponse et décrire l'impact du changement effectué sur la qualité du logiciel. Les questions se trouvent à la figure 6.2.

Pour ce qui est des changements, les participants ne pouvaient malheureusement pas confirmer que les changements proposés étaient corrects en testant le logiciel ou en le démarrant tout simplement. Le logiciel étudié ne pouvant pas être exécuté en dehors du contexte d'un projet d'*Eclipse* et la sortie étant soit récupérée directement par l'affichage graphique ou très complexe, il aurait été impossible de les laisser tester leurs réponses.

1. Les résultats pour *CBO* donnent toujours des valeurs correctes ou supérieures à ce qui est attendu. Le problème apparaît lors de la création des métriques. Dans ce cas précis, on considère que *CBO* est le nombre de classes différentes appelées par une classe donnée.
2. Lors de la lecture des bogues depuis un fichier, le système ne trouve systématiquement pas les éléments (classes) et les paquetages. La liste des entités reliées reste donc toujours vide. Aucun bogue n'apparaît durant la visualisation non plus.
3. Dans la vue sur les métriques de contrôle (*SVN*), on voit qu'il existe la métrique *MainAuthor* qui nous permet de connaître la personne qui travaille le plus sur une méthode, une classe ou un paquetage. Nous aimerions cette fois avoir la métrique *LastAuthor* qui nous indique la dernière personne à avoir travaillé sur une entité. (Vous pouvez uniquement implanter la méthode pour les éléments(classes))
4. Les paquetages et les systèmes se comportent souvent de la même façon. Dans les deux cas, ils peuvent contenir une hiérarchie de paquetages et d'éléments. Étudiez la possibilité de créer un parent commun (ou une hiérarchie commune) aux deux classes et commencez votre implantation.

Figure 6.2 – Questions expérience 1, partie 2. Activités concernant l'exécution de tâches dans le logiciel.

Dans un deuxième temps, après avoir effectué toutes les tâches ou bien en toute fin de période si le temps manquait, les sujets devaient remplir un questionnaire subjectif de sept questions. Un premier type de questions concerne la proportion de leur utilisation de *VERSO* pour répondre à chacune des questions, c'est-à-dire s'ils considèrent avoir plutôt utilisé les fonctionnalités de *VERSO* ou plutôt les fonctionnalités déjà existantes dans l'environnement de développement *Eclipse*. Un deuxième type de questions concerne leur appréciation de *VERSO* face à son utilité ou sa capacité à aider les utilisateurs à trouver des réponses et à sa facilité d'utilisation pour résoudre une tâche. Un dernier type de questions laisse de la place aux sujets pour s'exprimer sur les fonctionnalités les plus utiles, les points négatifs et positifs de *VERSO* ainsi que pour laisser des commentaires sur l'expérience. Un résumé de ces questions est présenté à la figure 6.3. Le questionnaire utilisé pour l'expérience est présenté en annexe I. Il contient plus de détails sur les tâches et les choix de réponse pour certaines questions.

1. Pour chacune des questions présentées dans la partie A, cotez l'utilité de *VERSO* pour répondre aux exigences de la question (1 - pas utile, 10 - très utile).
2. Pour chacune des questions présentées dans la partie A, déterminez le pourcentage approximatif d'utilisation de *VERSO* face aux fonctionnalités déjà présentes dans *Eclipse*.
3. Quelles fonctionnalités (tous outils confondus) ont été les plus utiles lors de l'expérience ?
4. Quelles fonctionnalités de *VERSO* ont été les plus utiles lors de l'expérience ?
5. Cotez l'utilité générale de *VERSO*. (1 - pas utile, 10 - très utile)
6. Y a-t-il des fonctionnalités absentes de *VERSO* que vous auriez aimé utiliser lors de l'expérience ?
7. Commentaires et suggestions sur *VERSO*.

Figure 6.3 – Questions expérience 1, partie 3. Questions générales sur l'appréciation de *VERSO*.

6.3.2 Les résultats

Nous résumons les résultats de l'expérience dans cette section. Dans un premier temps, nous présentons les taux de réussite et d'échec pour chacune des équipes et chacune des questions. Il s'agit seulement de confirmer qu'ils ont réussi l'objectif demandé aux fins de comparaison avec le reste de leurs réponses. Nous allons considérer trois valeurs ordinales dans le tableau suivant : *non* signifie que l'objectif n'a pas été réussi, *ok* signifie que l'objectif a été pratiquement réussi ou que les utilisateurs étaient sur la bonne voie, mais qu'il restait des erreurs mineures qu'ils auraient pu corriger s'ils avaient pu tester le programme, et finalement *bien* signifie que l'objectif a été bien réussi et qu'il pourrait être intégré comme modification dans la base de données du code. Une valeur manquante signifie que les participants n'ont pas commencé l'activité dû à un manque de temps. Les équipes sont différenciées pour permettre de faire des liens entre les différentes questions. Le questionnaire en annexe I peut ensuite être consulté pour comparer les résultats avec les tâches demandées. Les résultats pour cette partie sont présentés dans le tableau 6.I.

Nous présentons aux tableaux 6.II et 6.III, les réponses des participants concernant l'utilité de *VERSO*. Ensuite, nous montrons le pourcentage d'utilisation de *VERSO* correspondant pour chacune des activités à réaliser. Des valeurs d'agrégation statistiques ont été ajoutées pour des fins de discussion dans la section 6.3.3.

Équipe	Q1a	Q1b	Q1c	Q1d	Q2a	Q2b	Q3	Q4
Équipe A	bien	ok	ok	ok	non	non	ok	
Équipe B	bien	ok	bien	ok	ok	bien	ok	
Équipe C	bien	non	ok	ok	non	non	ok	
Équipe D	ok	bien	ok	ok	bien	non	non	
Équipe E	ok	non	ok	bien	bien	non	bien	ok
Équipe F	ok	ok	ok	bien	ok	ok		
Équipe G	bien	bien	bien	ok	bien	bien	ok	
Équipe H	bien	bien	bien	bien	bien	ok	ok	ok
Équipe I	bien	bien	ok	ok	bien	bien	ok	ok
Équipe J	bien	bien	bien	bien	bien	ok	bien	ok

Tableau 6.I – Tableau des réussites des activités. Dans ce tableau, on peut voir si les équipes ont réussi les activités. La valeur *non* signifie que les objectifs ne sont pas rencontrés, la valeur *ok* signifie qu'il reste quelques petites erreurs, mais que l'équipe est sur la bonne voie et la valeur *bien* signifie que l'activité est réussie. Si la case est vide, l'équipe n'a pas eu le temps de commencer l'activité.

Les données, présentées au tableau 6.IV, concernent la perception de l'utilité générale de *VERSO*. Il s'agit d'une réponse qui concerne toutes les activités et les sujets devaient se baser sur leur expérience globale.

Les derniers résultats pour cette expérience concernent les questions à court développement, soient les questions 3, 4, 6 et 7. Puisqu'il s'agit de questions à court développement sans théorie précise, les réponses de toutes les équipes étaient différentes. Par contre, nous avons regroupé plusieurs réponses sous des thèmes communs. En faisant le total de certaines questions, on s'aperçoit qu'il dépasse parfois 10. C'est seulement que quelques équipes ont abordé plus d'un thème dans leur réponse. Les résultats pour les questions 3 et 4 sont présentés dans le tableau 6.V. Quant aux résultats pour les questions 6 et 7, ils sont discutés rapidement à la prochaine section, mais ils sont trop variés pour les présenter ici.

La majorité des équipes ont utilisé les deux heures complètes pour répondre au questionnaire. Les participants ont eu besoin d'aide ponctuelle pour utiliser l'interface usager de *VERSO*. Les sujets semblaient enthousiastes de participer à l'expérience et étaient in-

Équipe	Q1a	Q1b	Q1c	Q1d	Q2a	Q2b	Q3	Q4	Médiane	Moyenne
Équipe A	9	6	9	8	10	7	10		9	8,4
Équipe B	7	4	8	8	6	6	2		6	5,9
Équipe C	10	10	9	10	10	10	10		10	9,9
Équipe D	8	5	8	6	7	6	6	6	6	6,5
Équipe E	8	8	6	9	7	9	7	8	8	7,8
Équipe F	10	10	8	7	1	1			7,5	6,2
Équipe G	9	5	8	9	7	7	5		7	7,1
Équipe H	9	7	5	9	1	1	1	1	3	4,3
Équipe I	8	6	8	9	8	7	3	2	7,5	6,4
Équipe J	8	5	9	8	3	3	1	1	4	4,8
Médiane	8,5	6	8	8,5	7	6,5	5	2		
Moyenne	8,6	6,5	7,8	8,3	6,1	5,8	5	3,3		

Tableau 6.II – Tableau de l'utilité perçue de *VERSO* pour les activités. Les réponses devaient être à l'intérieur d'une échelle de 1 à 10 et ne concernaient que les activités individuelles.

téressés à en connaître plus sur *VERSO*. Il n'y a pas eu de mortalité[1] durant l'expérience.

6.3.3 Discussion

Dans un premier temps, même sans tenir compte des résultats, le fait d'avoir eu 28 participants à cette étude de cas est déjà un succès. Les gens se sont présentés en grand nombre ce qui démontre un certain intérêt. Même s'il y avait une récompense sous la forme de points, elle était plutôt modeste. De plus, toutes les équipes ont réussi à terminer la majorité des activités prévues. Ceci est intéressant puisqu'avec un très court tutoriel (15 minutes) et une aide sporadique sur certains points techniques de l'interface et non sur les principes de visualisation, cela a permis à tous les participants de répondre à de courtes questions, mais suffisamment complexes pour causer problème en l'absence d'un outil d'aide à l'analyse. Un des objectifs de l'expérience, c'est-à-dire de prouver que *VERSO* peut être utilisé par n'importe quel programmeur, a été rempli assez aisément. La visualisation et la transformation des données en valeurs graphiques sont effectivement capables de réduire la complexité de l'analyse. De plus, l'intérêt général démontré pour

[1] Personnes quittant avant que l'expérience ne se termine.

Équipe	Q1a	Q1b	Q1c	Q1d	Q2a	Q2b	Q3	Q4	Méd.	Moy.
Équipe A	1	1	1	1	0,2	0	0		1	0,6
Équipe B	0,7	0,5	0,9	0,7	0,3	0,6	0,1		0,6	0,54
Équipe C	1	1	1	1	1	1			1	1
Équipe D	1	1	1	1	0,5	0	0	0	0,75	0,56
Équipe E	0,8	0,8	0,7	0,9	0,3	0,25	0,6	0,9	0,75	0,66
Équipe F	1	1	1	1	0,75	0,75			1	0,92
Équipe G	0,9	0,5	0,9	0,9	0,7	0,7	0,5		0,7	0,73
Équipe H	0,9	0,7	0,5	0,9	0	0	0	0	0,25	0,38
Équipe I	0,5	0,2	0,8	0,9	0,6	0,6	0,1	0	0,55	0,46
Équipe J	1	1	1	1	0,3	0,3	0	0	0,65	0,58
Médiane	0,95	0,9	0,95	0,95	0,4	0,45	0,05	0		
Moyenne	0,89	0,78	0,89	0,93	0,46	0,42	0,15	0,15		

Tableau 6.III – Proportion d'utilisation de *VERSO* pour chacune des activités. Ici on ne présente que la valeur perçue d'utilisation de *VERSO*. Pour obtenir le pourcentage d'utilisation des autres outils disponibles (fonctionalités d'*Eclipse*), il faut faire le complément. Les valeurs sont présentées entre 0 et 1.

Équipe	A	B	C	D	E	F	G	H	I	J	Médiane	Moyenne
Cote	8	7	10	7	9	9	7	8	9	6	8	8

Tableau 6.IV – Cote d'appréciation de l'utilité pour *VERSO*. Les réponses devaient être à l'intérieur d'une échelle de 1 à 10 et concernaient le prototype de *VERSO* en général.

l'expérience est encourageant pour l'éventuelle adoption de *VERSO* par l'industrie.

Un autre résultat intéressant est la réponse des équipes à la question 5 qui demande de coter l'utilité de *VERSO* (voir le tableau 6.IV). Sur une échelle de 1 à 10, les équipes ont répondu par une médiane et une moyenne de 8 et la valeur la plus faible est de 6. Cette valeur vient d'une équipe qui a donné des cotes faibles de manière générale. Cette forte médiane témoigne encore une fois de l'intérêt favorable face à *VERSO*.

Les questions 3 et 4 concernent les fonctionnalités préférées ou les plus utiles lors des activités (voir le tableau 6.V). Ces deux questions sont essentiellement les mêmes sauf que dans le premier cas, la question est générale et dans le deuxième cas, elle est restreinte aux fonctionnalités de *VERSO*. Des dix équipes, seulement deux ont préféré des fonctionnalités qui ne sont pas reliées à *VERSO*. Ces résultats sont très encourageants

Concepts Q3	Fréquence	Concepts Q4	Fréquence
Rep. graphique	2	Rep. graphique	8
Chang. d'asso.	5	Chang. d'asso.	3
Filtres	1	Filtres	2
Calcul des métriques	1	Bogues	2
Eclipse	2		

Tableau 6.V – Fonctionnalités les plus utiles. Présentations des réponses aux questions 3 et 4. Les réponses ont été placées par thème et la fréquence représente le nombre d'équipes ayant abordé ce thème.

quant à l'appréciation de *VERSO* et la pertinence de ses fonctionnalités pour étudier le logiciel et répondre à des questions lors du développement. De plus, les sujets ont mentionné en majorité la représentation graphique des éléments comme fonctionnalité importante et intéressante. Il s'agit en effet du coeur du concept entourant l'utilisation de *VERSO* et de la fonctionnalité la plus importante. Il est agréable de constater qu'elle semble particulièrement utile aux yeux de sujets en étant à leur premier contact avec *VERSO*.

Quand on regarde les données pour les questions individuelles (tableau 6.II et 6.III), on constate une utilité forte pour les premières questions et les résultats vont ensuite en diminuant vers les dernières questions. Les sujets ont vu immédiatement l'utilité de la visualisation pour répondre à des questions d'analyse, mais l'ont moins vu pour répondre aux questions où des modifications étaient requises. Ce sentiment est compréhensible puisqu'ils devaient utiliser l'éditeur traditionnel d'*Eclipse* pour écrire le nouveau code. Par contre, beaucoup d'équipes ont utilisé la visualisation pour trouver l'endroit pertinent dans le code où la modification devait être faite et ont pu évaluer cette modification par la suite en observant les changements dans la représentation graphique. Bien que ces phénomènes aient été directement observés par ceux qui opéraient l'expérience, ils n'ont pas été reflétés dans les résultats. Certaines équipes n'ont pas du tout tenu compte de cet aspect en évaluant l'utilité à 1 sur une échelle de 10. Le tableau 6.III contenant les pourcentages d'utilisation explique en partie cette réaction. La plupart des équipes ont considéré que *VERSO* demeurait utile même s'il était moins présent quand venait le temps de

répondre aux questions, alors que pour certaines équipes l'utilité et le pourcentage d'utilisation étaient directement reliés. C'est peut-être dû à une mauvaise compréhension de ce qu'est l'utilité d'un outil. On a par contre remarqué que les gens ont trouvé *VERSO* utile en moyenne, ce qui compense les résultats des questions individuelles. Le nombre de personnes ayant eu l'occasion de terminer les dernières questions peut aussi avoir eu une influence sur les résultats des dernières activités.

Pour ce qui est du tableau 6.I mentionnant les réussites et les échecs, on peut voir que de façon générale les participants ont bien réussi les questions. Le grand nombre de *ok* peut être expliqué par le fait que les participants n'avaient pas la possibilité de tester leur programme et ne pouvaient donc pas faire une légère correction facile à détecter dans d'autres circonstances. Les échecs sont surtout concentrés dans la question 2b qui demandait une abstraction plus complexe, et qui est plus reliée au niveau d'expérience en programmation qu'à l'utilisation de *VERSO*. Il est difficile de tirer plus d'informations de ce tableau et il semble que la réussite ou l'échec ne soit pas relié aux autres résultats. Puisque le pourcentage d'utilisation est subjectif, l'absence de lien entre ces deux variables ne présente pas un problème.

Les questions 6 et 7 concernent des améliorations et des commentaires sur *VERSO*. Comme prévu, les réponses sont très variées. Parmi les réponses les plus fréquentes, on retrouve des commentaires sur l'amélioration du mouvement de la caméra. Pour les développeurs de *VERSO* qui utilisent constamment le contrôle de la caméra, il ne s'agissait pas d'une question importante puisqu'ils l'utilisaient sans problème. L'étude a permis de constater que ces contrôles pouvaient être plus difficiles à assimiler pour les nouveaux utilisateurs de *VERSO*. Il s'agit d'une information qu'il aurait été impossible d'obtenir sans cette étude. Les contrôles ont d'ailleurs été légèrement modifiés pour changer les interactions à l'aide de la souris. Au lieu que la caméra suive les mouvements de la souris, le point de vue est maintenant plutôt déplacé vers le point d'intérêt des utilisateurs.

6.3.4 Menaces à la validité

Pour cette expérience, les résultats sont intéressants, mais la présence de certaines menaces à la validité n'est pas exclue. Dans un premier temps, bien que les questions

n'aient pas été générées pour favoriser *VERSO*, les questions ont tout de même été rédigées par le concepteur de *VERSO*, alors qu'il avait l'outil en tête. Les questions représentent des activités semblables à des cas réels, mais elles restent insérées dans un contexte artificiel. Pour certaines questions impliquant des modifications, les participants n'étaient pas en mesure de tester leurs changements. Ceci peut avoir influencé le niveau de confiance des participants dans un sens ou dans l'autre. Les sujets formaient eux-mêmes leur équipe et certaines disparités dans les niveaux de connaissances ont pu se créer dû à ce phénomène. Par contre, ce facteur est atténué par le fait que les sujets avaient tous une formation très semblable et une connaissance des principes de la qualité issus des mêmes présentations théoriques. Aussi, les participants devaient mentionner leur utilisation de *VERSO* comparativement à celle des fonctionnalités traditionnelles d'*Eclipse*, mais ils n'ont pas eu de comparatif soit en matière d'outils d'analyse automatiques ou encore d'autres outils de visualisation. Ensuite, les sujets recevaient quelques points supplémentaires pour se présenter à l'expérience et répondre aux questions sérieusement. Ceci peut avoir influencé l'opinion de certains et avoir modifié leur enthousiasme pour l'expérience. Durant cette expérience, les sujets étaient des étudiants en dernière année du baccalauréat. Ils peuvent avoir des réactions différentes de celles de programmeurs de l'industrie. Par contre, à ce niveau et particulièrement pour une expérience contrôlée où les tâches concernent un logiciel que les participants ne connaissent pas, les étudiants se comportent de façon semblable à des programmeurs juniors devant un environnement inédit. Par contre, nous pensons que ces facteurs ont eu peu d'influence sur les résultats à cause des réponses variées ayant été reçues et du grand nombre de participants.

6.4 Expérience 2 : Étude de cas *in vivo*

Cette deuxième expérience visait à évaluer *VERSO* dans un environnement moins contrôlé pour s'assurer qu'il soit utilisable dans un contexte industriel pour des projets plus complexes. Il y a moins de sujets qui étaient évalués, mais sur une période plus longue. Cette fois-ci, les sujets avaient à développer un logiciel complet. Les sujets

avaient aussi une plus grande liberté pour accomplir les tâches et devaient développer une partie du projet à l'aide de *VERSO* et l'autre partie avec un environnement traditionnel (*Eclipse*). Les performances de *VERSO* seront évaluées par l'entremise de l'étude d'outils reliés au développement, le logiciel résultant lui-même et les commentaires recueillis auprès des sujets et à la fin de l'expérience. Il y a eu beaucoup moins de contrôle sur les variables de l'expérience, ce qui rend les résultats plus difficiles à interpréter, mais ils sont beaucoup plus généralisables par la suite que pour la première expérience.

6.4.1 Méthode

6.4.1.1 Les sujets

L'expérience demandait à dix sujets d'utiliser *VERSO* pour réaliser un projet comptant pour 40 % de l'évaluation d'un cours gradué de génie logiciel. Des dix personnes disponibles au départ de l'expérience, huit ont décidé de continuer à assister au cours et donc de participer à l'expérience. Il y a donc eu deux équipes de quatre personnes. Dans le but de réduire le biais important de l'expérience en programmation et en réalisation de projet, les équipes ont été formées pour équilibrer le plus possible les forces. Les participants devaient remplir un questionnaire stipulant entre autres leur niveau d'expérience en programmation avec le langage *Java* et leur connaissance face au domaine pour lequel on leur demandait de créer un logiciel. Les questions sont présentées à la figure 6.4.

1. Quel est votre niveau d'expérience en programmation ?
2. Quel est votre niveau d'expérience en *Java* ?
3. Quel est votre niveau d'expérience avec l'IDE *Eclipse* ?
4. Quel est votre niveau d'expérience avec le travail en équipe à travers les systèmes de contrôle de versions ?
5. Quel est votre niveau d'expérience avec les réseaux de Petri ?

Figure 6.4 – Expérience préalable. Questions utilisées pour connaître l'expérience préalable des participants et former les équipes.

Le tableau 6.VI contient les réponses des utilisateurs au questionnaire de classement initial et le questionnaire lui-même est présenté à l'annexe II. Il contient plus de détails et les choix de réponses aux questions. Les sujets assistaient tous au même cours gradué en

génie logiciel avec des préalables bien définis, mais venaient de cheminements parfois différents. Les deux équipes devaient effectuer le même projet, mais dans un premier temps le groupe A utilisait l'environnement de développement *Eclipse* équipé du plugiciel *VERSO* alors que le groupe B utilisait uniquement *Eclipse* sans le plugiciel *VERSO*. La condition de l'utilisation ou non de *VERSO* a été échangée pour la deuxième phase du projet, mais la composition des groupes est restée la même. La réalisation du projet dans le cours était obligatoire pour tous les étudiants puisqu'elle était contributive pour une partie importante de la note finale. La participation à l'expérience était elle aussi obligatoire, mais les efforts supplémentaires à fournir par rapport à la seule réalisation du projet étaient négligeables. Bien entendu, les activités liées à l'expérience ou les résultats de l'expérience n'étaient pas tenus en compte dans la note sur le projet ou pour la participation individuelle de chacun.

Sujet	Groupe	Q1	Q2	Q3	Q4	Q5
Sujet 1	A	c	c	c	b	a
Sujet 2	A	c	b	b	b	b
Sujet 3	A	c	c	b	a	b
Sujet 4	A	c	c	c	b	b
Sujet 5	B	c	c	c	b	a
Sujet 6	B	c	c	a	b	b
Sujet 7	B	c	c	c	b	a
Sujet 8	B	d	d	d	c	a

Tableau 6.VI – Expérience préalable au projet pour la 2e étude de cas. Pour les questions 1 à 3, les choix de réponses vont de *a- Aucune expérience* à *d- Expert*. Pour la question 4, les choix de réponses vont de *a- Pas d'expérience* à *c -Plus d'une année d'expérience*. Pour la question 5, les choix vont de *a-Aucune (expérience)* à *d-Expert* des réseaux de Petri.

6.4.1.2 Le temps

Pour cette expérience, les sujets devaient réaliser un projet qui s'étalait sur une grande partie de la session d'automne 2010 (durée d'un peu plus de trois mois). Dans les faits, les étudiants du cours avaient toute la liberté pour gérer leur temps sur le pro-

jet. Puisque les étudiants ont d'autres activités à réaliser durant la session, on imagine bien qu'ils travaillaient à temps partiel sur le projet. Une partie du travail des étudiants consistait à comptabiliser le temps pour chacune des activités.

6.4.1.3 Le lieu et les ordinateurs

Comme pour la gestion de leur temps, les étudiants n'avaient pas de contraintes sur le lieu de travail et les ordinateurs à utiliser. Les participants utilisaient d'ailleurs leur ordinateur personnel pour faire les travaux. Ça a donc occasionné plusieurs configurations matérielles différentes parfois utilisées sous des systèmes d'exploitation différents. Le prototype de *VERSO* a été développé sur un ordinateur précis et fonctionnait très bien sur des ordinateurs semblables. Il avait été peu testé sur d'autres configurations. Bien que quelques problèmes aient été rencontrés lors de l'installation sur les ordinateurs des participants, il a été possible de trouver des solutions à ces problèmes et d'installer *VERSO* et de l'utiliser sur toutes les configurations qui se sont présentées.

6.4.1.4 Les activités et les questions

Les étudiants devaient réaliser un projet impliquant la création d'un outil d'édition graphique pour les réseaux de Petri. Le logiciel à réaliser devait être basé sur le cadre d'applications *JHotdraw*. *JHotdraw* permet justement de créer ce genre d'éditeur graphique. Dans la première partie, les participants devaient créer les mécanismes sous-jacents à l'éditeur et ils devaient aussi faire l'animation du réseau de Petri en respectant un critère supplémentaire de délai pour les transitions. Dans la deuxième partie, les participants devaient créer le graphe des marquages et analyser certaines propriétés à partir d'un réseau de Petri déclaré à l'aide des outils qui ont été développés dans la première partie. Les étudiants devaient présenter leur avancement à mi-parcours et en fin de parcours aux fins d'évaluation. Une description complète des deux parties à réaliser se trouve en annexes III et IV.

Au cours de l'expérience, les sujets devaient participer à certaines activités pour permettre la cueillette de données. Premièrement, les sujets devaient travailler avec *Eclipse*

avec ou sans *VERSO* selon la période étudiée. Même pour l'équipe qui n'utilisait pas *VERSO*, *Eclipse* a été imposé comme outil de développement pour avoir un outil de comparaison constant entre tous les sujets. Deuxièmement, les participants devaient utiliser le système de contrôle de versions *SVN* et se connecter à un répertoire créé spécialement pour l'occasion. Troisièmement, les sujets devaient remplir des feuilles de temps pour témoigner de chacune des activités auxquelles ils prenaient part dans le but d'éventuellement comparer des activités similaires lors de l'analyse des données. Ces feuilles de temps sont remplies sur le web avec un serveur où le logiciel *timesheetnextgeneration* [2] est installé. Il est possible pour l'administrateur des feuilles de temps d'obtenir des rapports sur les activités et d'avoir accès au calendrier de chacun des utilisateurs. Pour éviter un important biais, les informations collectées à l'aide des feuilles de temps servaient seulement aux fins de l'expérience et n'étaient pas reflétées dans les notes individuelles des équipes.

À la fin de l'expérience, les participants devaient remplir un court questionnaire sur leur utilisation et leur appréciation de *VERSO*. Contrairement au questionnaire de la première expérience qui contenait les tâches à accomplir, ce questionnaire servait uniquement à recueillir les impressions et les préférences des sujets. Dans ce questionnaire, on leur posait entre autres des questions sur leur utilisation personnelle de *VERSO* et des autres outils à leur disposition (voir figure 6.5).

1. Selon quelle fréquence avez-vous utilisé le logiciel de contrôle de versions ?
2. Selon quelle fréquence avez-vous rempli les feuilles de temps ?
3. Comment avez-vous utilisé *VERSO* ?
4. Combien avez-vous entré de bogues dans le système ?
5. Avez-vous utilisé la fonctionnalité permettant de créer des versions ?

Figure 6.5 – Questions expérience 2, partie 1. Questions sur l'utilisation des outils et de *VERSO* durant l'expérience.

Ensuite, il y avait quelques questions sur l'utilisation des multiples vues et de la navigation entre ces vues, incluant une description d'un scénario typique d'utilisation de *VERSO*. Cette question était importante en raison du fait que les étudiants travaillaient

[2] http ://www.timesheetng.org/

chez eux et il était impossible de les observer au fur et à mesure. Les questions pour cette catégorie sont présentées à la figure 6.6.

1. Avez-vous utilisé la vue de la qualité ?
2. Avez-vous utilisé la vue du système de contrôle de versions ?
3. Avez-vous utilisé la vue des bogues ?
4. Quelle vue a été la plus utile pour vous ?
5. Avez-vous utilisé le niveau de granularité sur les lignes de code ?
6. Avez-vous utilisé le niveau de granularité des méthodes ?
7. Avez-vous utilisé le niveau de granularité des classes ?
8. Avez-vous utilisé le niveau de granularité des paquetages ?
9. Quel niveau de granularité a été le plus utile pour vous ?
10. Avez-vous utilisé les outils de navigation entre les vues ?
11. Jugez-vous pertinent d'avoir plusieurs vues dans un même outil plutôt qu'avoir accès à plusieurs outils ?
12. De quelle façon avez-vous principalement utilisé *VERSO* ? Décrire un scénario typique.

Figure 6.6 – Questions expérience 2, partie 2. Questions concernant l'utilisation des différentes vues de *VERSO*.

Il y a eu des questions sur l'amélioration du processus de développement en utilisant *VERSO* (voir figure 6.7). Une des questions les plus importantes de ce type était de savoir si la présence de *VERSO* avait augmenté leur intérêt pour les informations nouvellement disponibles et s'il y avait eu des répercussions sur le produit final, entre autres en matière de qualité. Finalement, il y avait une section pour l'évaluation subjective de *VERSO* dont une des questions les plus importantes était l'adoption éventuelle de *VERSO*. Les questions pour cette section sont présentées à la figure 6.8.

Le questionnaire complet se trouve à l'annexe V et donne des détails supplémentaires sur les questions ainsi que des choix de réponses pour certaines questions. Il permet de bien suivre les explications des sections 6.4.2 et 6.4.3.

6.4.2 Résultats

En observant les informations sur les feuilles de temps et sur l'historique de *SVN*, on constate que les sujets ont entré de courts commentaires durant leurs périodes de travail. La longueur des commentaires est d'une phrase ou moins de façon générale pour les deux

1. En temps normal, quel est votre niveau d'intérêt pour la qualité reliée au code et sa maintenance ?
2. Pour ce projet, quel a été votre niveau d'intérêt pour la qualité reliée au code et sa maintenance ?
3. De quelle façon *VERSO* a modifié votre niveau d'intérêt pour la qualité reliée au code et à sa maintenance ?
4. De quelle façon *VERSO* a modifié votre performance quand vous développiez le logiciel ? Comparez les deux périodes : celle où vous aviez accès à *VERSO* et celle où vous utilisiez seulement *Eclipse*.
5. De quelle façon *VERSO* a modifié la qualité du programme développé ? Comparez les deux périodes : celle où vous aviez accès à *VERSO* et celle où vous utilisiez seulement *Eclipse*.

Figure 6.7 – Questions expérience 2, partie 3. Questions sur l'influence de *VERSO* durant le développement du logiciel.

groupes. Par contre, au moins un participant du groupe A n'a entré aucun commentaire et seulement la durée de la tâche dans le système de carte de temps. Les comportements étaient semblables pour les commentaires des *commits*, le groupe B étant légèrement plus loquace que le groupe A dans les deux cas. Il n'y a pas de différences majeures pour les deux équipes dans le style et la longueur des commentaires en considérant la première partie et la deuxième partie du travail (la présence de *VERSO* n'aurait donc pas eu d'influence à ce sujet). Dans les deux équipes, il y a un débalancement entre le nombre de *commits* des participants et on retrouve dans chaque équipe une personne qui se démarque quant au nombre de ses *commits*. Les *commits* se concentrent généralement vers la fin des étapes avec une répartition un peu plus uniforme pour le groupe B. Dans ce groupe, il y a eu un peu plus de *commits* dans la première partie (où ils n'avaient pas accès à *VERSO*) que dans la deuxième partie.

Le contenu des commentaires concerne essentiellement des tâches accomplies et ne se rapporte pas directement à des activités reliées à *VERSO*, peu importe s'il s'agit de la période où les équipes y avaient accès ou non. Pour le groupe B, les commentaires comportent parfois l'expression *refactoring* qui peut se rapporter à une activité qui découle d'une observation dans *VERSO*. Fait intéressant à noter : cette expression est présente uniquement dans les commentaires des feuilles de temps remlies dans la deuxième partie

1. Est-ce que selon vous *VERSO* était juste et précis dans sa représentation de la qualité et des informations sur le code ?
2. Quelle était votre fonctionnalité favorite dans *VERSO*
3. Cotez l'utilité générale de *VERSO* (1-pas utile, 9-très utile)
4. Après la période de développement et l'apprentissage de *VERSO* complété. Cotez l'utilité générale de *VERSO* (1-pas utile, 9-très utile)
5. Est-ce que l'état de prototype de *VERSO* a pu nuire à votre expérience ?
6. Si une version complète et finale d'un logiciel comme *VERSO* était disponible gratuitement avec votre éditeur de programme favori, est-ce que vous seriez tenté de l'utiliser ?
7. Y a-t-il des fonctionnalités manquantes de *VERSO* que vous auriez aimé utiliser ?
8. Avez-vous des commentaires ou des suggestions pour l'amélioration de *VERSO* ?
9. Avez-vous des commentaires généraux en tant que participant à l'expérience ?

Figure 6.8 – Questions expérience 2, partie 4. Questions sur l'appréciation générale de *VERSO*.

du projet où ce groupe avait accès à *VERSO*. L'expression débogage revient aussi souvent dans les deux équipes, mais il ne semble pas y avoir de relation entre la présence de ce mot clé et la condition d'avoir accès à *VERSO* ou non.

Le nombre d'heures pour les deux équipes est très inégal entre les participants. Il est difficile de dire si c'est dû au fait que les cartes de temps n'ont pas toujours été remplies ou si cette inégalité représente les heures réellement consacrées au projet. Dans le cas du groupe A, le nombre d'heures total consacrées à la première partie est inférieur au nombre de la deuxième partie (environ 200 contre environ 230). Pour le groupe B, les deux périodes sont comparables sauf qu'ils ont inclus le temps pour faire la présentation et pour l'installation de *VERSO* dans la deuxième partie, alors qu'une activité de ce genre n'a pas été répertoriée pour la première partie. Il s'avère donc que les deux équipes auraient pris moins de temps quand elles avaient accès à *VERSO*.

Ensuite, les travaux des deux équipes ont été observés par les évaluateurs à l'aide de *VERSO* une fois la remise finale effectuée. Nous avons aussi observé le logiciel à mi-parcours pour voir s'il y avait de l'évolution dans la qualité ou un changement dans les comportements quand *VERSO* était disponible ou non. Il s'agit d'une application intéressante de *VERSO* hors du strict développement de logiciel. De plus, *VERSO* pourrait être utilisé de façon similaire dans les revues par les pairs effectuées dans l'industrie

puisqu'il permet d'analyser le logiciel tout en étant intégré dans l'environnement de développement.

Dans un premier temps, on pouvait voir que le projet du groupe A comportait légèrement plus de classes que le projet du groupe B. Le projet du groupe B avait aussi une complexité générale plus grande. On a aussi pu confirmer le débalancement entre le nombre de *commits* selon les programmeurs dans les deux projets. Dans le groupe A, les indices traditionnels de qualité se sont légèrement détériorés dans la phase deux du projet. Dans ce groupe, il y a par contre eu plus de classes effacées ou modifiées par rapport à la version 1. Dans le groupe B, on a aussi remarqué une légère détérioration des indices de qualité et plus particulièrement du couplage.

Au niveau des deux groupes, il y avait une domination d'un programmeur dès la première moitié du projet. Il est ensuite devenu le programmeur principal pour pratiquement toutes les classes du logiciel. Il s'agit probablement d'une personne qui était le chef d'équipe. Pour finir, les deux équipes ont réussi la majorité des objectifs à réaliser dans les deux parties du travail. Le groupe B a eu une performance légèrement meilleure que le groupe A. La figure 6.9 montre un exemple de visualisation tiré des deux projets dans le contexte de la qualité au niveau de granularité des classes.

Ensuite viennent les résultats des questionnaires. Le résumé des réponses concernant l'utilisation des différentes vues est présenté dans le tableau 6.VII. Les réponses des participants concernant la performance et la qualité sont résumées dans le tableau 6.VIII. Un dernier tableau (6.IX) présente les réponses aux questions concernant l'évaluation subjective de *VERSO* par les participants. Six participants ont dit avoir utilisé *VERSO* une fois le logiciel terminé pour l'évaluer et le visualiser. Les deux autres participants ont mentionné qu'ils l'utilisaient après chaque séance de programmation.

Pour les résultats des réponses à court développement, six participants sur huit disent utiliser *VERSO* de temps en temps pour évaluer si des classes avaient besoin d'un *refactoring* en réponse à la question sur la description d'un scénario typique. Pour ce qui est des fonctionnalités préférées, les participants ont répondu au nombre de quatre que la représentation des éléments était la plus intéressante et deux ont répondu que les filtres représentant les liens étaient les plus intéressants. Les deux autres participants n'ont pas

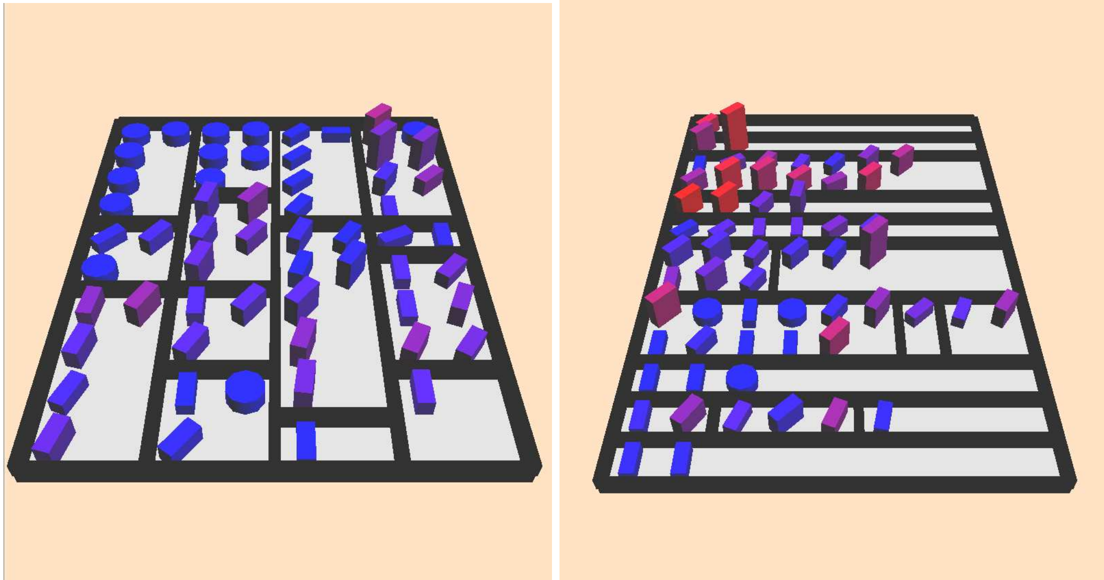

Figure 6.9 – Visualisation des deux projets liés à l'expérience 2. Exemple de visualisation avec *VERSO* utilisé pour évaluer et étudier les projets dans l'expérience. On voit ici le niveau de granularité des classes et le contexte de la qualité. Le projet du groupe A se trouve à gauche alors que celui du groupe B se trouve à droite.

répondu à la question. Pour ce qui est des commentaires généraux, plusieurs participants ont mentionné la lourdeur de l'application. Un participant a relevé que *VERSO* était très utile dans les phases de maintenance et un autre a mentionné qu'il n'avait pas besoin de la visualisation puisqu'il savait déjà où se trouvaient les problèmes dans le code.

6.4.3 Discussion

En regardant les informations sur les cartes de temps et l'historique des *commits*, il est difficile de conclure quoi que ce soit à partir des commentaires. Le nombre de *commits* est aussi peu représentatif pour la question qui nous intéresse. Par contre, le fait que les participants aient utilisé moins d'heures quand ils avaient accès à *VERSO* est une donnée intéressante. Ceci impliquerait qu'ils ont réussi à mieux s'organiser pour prendre moins de temps à l'aide de la visualisation. Il faut par contre relativiser ce résultat avec les autres facteurs en jeu comme le débalancement en terme de difficulté des deux étapes et le fait que les participants devaient régler les problèmes rencontrés dans la première

Sujet	Groupe	Contexte	Granularité	Navigation	Outil unique
Sujet 1	A	qualité	m,c,p	oui	5
Sujet 2	A	qualité	m,c,p	non	4
Sujet 3	A	qualité	m,c,p	non	3
Sujet 4	A	qualité	c,p	non	4
Sujet 5	B	qualité et SVN	l,m,c,p	non	4
Sujet 6	B	qualité	p	oui	4
Sujet 7	B	qualité	c,p	oui	4
Sujet 8	B	qualité et SVN	m,c,p	non	3

Tableau 6.VII – Tableau de l'utilisation des vues dans *VERSO*. Les niveaux de granularité sont identifiés par l-ligne de code, m-méthode,c-classe et p-paquetage. Les participants devaient aussi indiquer s'ils avaient utilisé la navigation ou non et s'ils jugeaient pertinent d'avoir accès à toutes ces vues dans un outil unique (échelle de 1 à 5 où 1 indique pas pertinent et 5 très pertinent).

partie quand ils ont commencé la deuxième partie. Dans un deuxième temps, il est intéressant de voir que le groupe B a uniquement parlé de *refactoring* quand ils ont eu accès à *VERSO*. Cette préoccupation pour l'amélioration de la qualité est peut-être due à la présence de la visualisation dans leur environnement de travail. L'accessibilité et la représentation imagée les auraient donc poussés à améliorer leur logiciel avant de le remettre.

L'évaluation de la qualité et des informations sur l'historique des fichiers à l'aide de *VERSO* donne des informations intéressantes sur les projets, mais ne permet pas de conclure sur l'amélioration de la performance des programmeurs. Les légères détériorations observées pour la deuxième partie des deux groupes sont vraisemblablement dues à l'augmentation du nombre d'éléments puisque la deuxième partie devait se construire par-dessus la première. Par contre, l'évaluation nous a permis de voir des phénomènes intéressants dans le code des équipes, par exemple les distributions des responsabilités qui ont semblé évoluer de la même façon dans les deux groupes avec un meneur (sujet 8 et sujet 4) qui prend les rênes du travail dans les deux cas. Quand on regarde les données sur l'expérience antérieure au projet, on remarque que ces meneurs se considéraient comme de très bons programmeurs, ce qui vient confirmer nos observations.

Les données sur l'utilisation des différentes vues présentées au tableau 6.VII sont

Sujet	Groupe	Influence intérêt	Influence qualité	Influence performance
Sujet 1	A	+	=	-
Sujet 2	A	=	=	=
Sujet 3	A	+	+	+
Sujet 4	A	+	=	=
Sujet 5	B	=	+	=
Sujet 6	B	+	+	=
Sujet 7	B	=	=	-
Sujet 8	B	=	=	=

Tableau 6.VIII – Tableau sur la qualité et la performance dans *VERSO*. Les participants indiquaient de quelle façon *VERSO* et la représentation graphique les avaient influencés quant à l'intérêt sur la qualité, la qualité réelle du logiciel produit et la performance en programmation. Les symboles +, = et - sont utilisés en lieu et place de *augmentation*, *maintien* et *diminution*.

difficiles à interpréter. Dans quelques cas, les participants ont dit qu'ils avaient utilisé plusieurs vues différentes à travers plusieurs contextes et plusieurs niveaux de granularité, alors qu'ils répondent ne pas avoir utilisé la navigation entre les vues. Or, c'est impossible puisque pour accéder à d'autres vues, il faut utiliser cette navigation. Il se peut que cette question ait été mal comprise par certains participants. De plus, six des huit participants ont mentionné avoir utilisé *VERSO* seulement pour évaluer le projet à la fin, alors qu'encore une fois six des huit participants ont décrit un scénario qui impliquait des observations dans *VERSO* beaucoup plus fréquentes, c'est-à-dire au minimum après chaque séance de programmation. Il se peut encore une fois que les choix de réponses de cette question aient été mal compris. Une donnée encourageante par contre réside dans le fait que les participants trouvent pertinent d'avoir plusieurs vues dans un même outil, ce qui est une des idées fondatrices de notre approche. La médiane des valeurs est *pertinent* ou 4 sur une échelle de 5, et les valeurs sont généralement élevées.

Pour ce qui est des données sur l'amélioration de la qualité dans le logiciel, on note que l'intérêt est resté stable ou a augmenté en présence de *VERSO*. Les données sont très semblables quand on interroge les participants sur l'augmentation réelle de la qualité pour le logiciel développé. Pour ce qui est de la performance, les deux participants ayant répondu que la performance avait diminué, ont fait référence aux ralentissements de leur

Sujet	Groupe	cote avant	cote après	Influence prototype	Adoption
Sujet 1	A	3	6	oui	oui
Sujet 2	A	6	7	non	oui
Sujet 3	A	7	7	oui	oui
Sujet 4	A	6	6	oui	oui
Sujet 5	B	6	6	non	oui
Sujet 6	B	6	6	non	oui
Sujet 7	B	7	7	oui	oui
Sujet 8	B	3	4	non	non
Médiane		6	6		
Moyenne		5,5	6,1		

Tableau 6.IX – Tableau sur l'évaluation subjective de *VERSO*. Les cotes avant et après la période d'apprentissage sont comprises dans une échelle de 1 à 9. On demandait aussi aux participants si le fait que *VERSO* était un prototype avait influencé leur cote et s'ils adopteraient un outil de visualisation intégré dans l'éventualité qu'une version complète existe.

ordinateur dans leurs commentaires dus au fait que *VERSO* devait parfois recalculer les informations sur *SVN*. Ce n'est donc pas à leur capacité d'écrire du code rapidement qu'ils faisaient nécessairement référence.

Pour les résultats sur l'utilité globale de *VERSO* (voir le tableau 6.IX), les données sont encourageantes puisqu'on obtient des moyennes de 5,5 et 6,1 et des médianes de 6 (sur une échelle allant de 1 à 9) une fois la période d'apprentissage passée. Quelques participants ont aussi dit que le statut de prototype de *VERSO* avait influencé la note donnée (la note aurait donc pu augmenter). Une seule personne donne une note inférieure à 6. Il s'agit du sujet 8 qui est le seul à avoir annoncé qu'il était un expert en programmation. Ce genre de personnes ont l'habitude d'aimer les outils qu'ils utilisent déjà et avec lesquels ils sont déjà performants. Kuhn *et al.* [49] ont d'ailleurs une discussion similaire sur les différences entre le rapport aux outils des experts et des néophytes. C'est aussi pour des raisons semblables que quelques programmeurs utilisent encore des éditeurs basés sur le texte sans les encadrer dans un environnement de développement moderne. Le commentaire laissé par le sujet 8 est d'ailleurs éloquent à ce sujet. Pour résumer, il a dit qu'il n'avait pas besoin de logiciel de visualisation puisqu'il savait déjà où se trou-

vaient les éléments problématiques dans le logiciel. Bien que *VERSO* s'adresse aussi aux programmeurs experts, il est moins utile pour les petits projets et pour les intervenants qui ont une connaissance profonde du logiciel. Il s'adresse plutôt à des utilisateurs qui devaient répondre à plusieurs questions avant de commencer à travailler sur une section en particulier. Finalement, le résultat le plus intéressant est que tous les participants sauf le sujet 8 seraient intéressés à utiliser *VERSO* ou un outil de visualisation semblable s'il était disponible en version complète. Ce résultat est aussi pertinent pour tout le domaine de la visualisation, car il montre le bien-fondé de toute la recherche qui y est effectuée.

6.4.4 Menaces à la validité

Bien qu'une attention particulière ait été portée à ce qu'il y ait le moins de biais possible dans l'expérience, les ressources restreintes et la nature de l'étude font qu'il est impossible d'être parfait. Tout comme la première expérience, il existe un risque dû à l'hétérogénéité des sujets. Ce problème est atténué par le questionnaire de l'expérience préalable. Par contre, une fois l'expérience et les évaluations du cours passées, on a remarqué quelques différences entre les équipes. Ceci représente un problème puisqu'il y avait seulement deux équipes et huit participants à l'expérience. De plus, il s'agit d'une validation croisée, mais dû au peu de contrôle des expérimentateurs sur les tâches effectuées par les sujets, il est difficile de comparer les activités faites par une même personne avec et sans l'utilisation de *VERSO*. Lors de cette expérience, le biais dû au fait que les participants ont une récompense et qu'ils sont poussés à répondre favorablement n'est pas présent. Par contre, étant donné que l'expérience était sans répercussion, certains sujets répartis dans les deux équipes n'ont peut-être pas mis l'effort nécessaire pour essayer sérieusement *VERSO* et ont aussi rempli les conditions liées à *SVN* et aux feuilles de temps avec moins de précision. Durant cette expérience, les sujets étaient des étudiants gradués. La question de savoir si les étudiants sont effectivement de bons représentants de développeurs réels dans l'industrie a été discutée, entre autres dans [10]. Dans notre cas, nous utilisons des étudiants gradués qui ont des connaissances comparables à des programmeurs juniors. Bien que la façon de répartir les périodes de développement soit différente pour les étudiants dans un cours, nous pensons que ces facteurs n'ont pas

influencé la perception face à un outil d'aide au développement visuel. De façon générale, cette expérience a les avantages et les inconvénients qui sont opposés à ceux de la première expérience dû au relâchement du contrôle sur les variables.

6.5 Évaluation globale et conclusions

VERSO permet de voir le logiciel d'une autre façon en présentant graphiquement les informations typiques nécessaires au développement et les aspects de sa qualité. Le fait de présenter les informations différemment amène à faire de nouvelles découvertes qui peuvent difficilement être décelables en regardant des colonnes de chiffres ou même en utilisant des outils automatiques sophistiqués. De plus, le fait d'avoir une représentation graphique, générant des stimuli intéressants, permet aux gens de s'intéresser plus facilement à l'outil et aux tâches de développement et d'analyse. Les personnes externes au projet sont naturellement attirées par cette nouvelle façon de voir le logiciel et cette curiosité nous amène beaucoup de commentaires et d'interactions. Si on fait référence à l'article de Muzner [67] présenté en détail dans la section 6.2, on peut voir que *VERSO* a suivi en grande partie les étapes décrites et certaines des évaluations pour ces étapes.

Une hypothèse importante avant même de commencer à réfléchir à la visualisation est que les métriques utilisées sont des représentants pertinents d'un logiciel et de sa facilité de maintenance. C'est-à-dire que ces informations sont pratiques et permettent de répondre aux questions typiques liées au développement ainsi que la création de meilleurs logiciels. Cette hypothèse n'a pas été validée dans ce travail, mais bien par la littérature abondante sur le sujet se trouvant même en dehors de la visualisation.

Dans un premier temps, *VERSO* a été conçu avec les principes de perception à l'esprit. Le fait d'utiliser moins de caractéristiques graphiques pour s'assurer que celles qui sont visibles aient le moins d'interférence possible est un exemple de ces principes de perception. Un autre exemple est la présence du principe de cohérence qui est inscrit dans un système où les multiples vues sont au coeur de la stratégie d'analyse. Une attention particulière a donc été portée à ce qu'il n'y ait pas de sauts cognitifs entre les différentes vues réduisant ainsi les efforts lors de l'analyse.

Dans un deuxième temps, bien que *VERSO* ne soit pas encore à *source ouverte*, nous l'avons rendu disponible à quelques personnes qui en ont fait la demande directement. Entre autres, une équipe de l'université de Nancy l'a utilisé avec succès pour visualiser des informations sur leur logiciel et pour enseigner la qualité du logiciel de façon graphique à des étudiants au niveau universitaire. De plus, quelques équipes ont fourni des informations pour qu'on puisse les visualiser et les analyser. Le fait que *VERSO* soit populaire bien qu'il n'ait pas été rendu disponible à tous montre une certaine forme d'adoption et surtout d'utilisation pour résoudre des problèmes réels. Bien que sa diffusion soit restreinte pour l'instant, chaque fois que nous montrons *VERSO* à un groupe de personnes, une partie de ce groupe désire l'essayer pour visualiser ses propres informations.

Tout comme plusieurs autres outils de visualisation, *VERSO* a aussi été utilisé avec plusieurs logiciels par ses développeurs pour chercher des informations anecdotiques ou des phénomènes cachés dans le code. Trouver ce genre d'informations dans des logiciels réels permet de confirmer que l'outil est effectivement capable d'analyser des logiciels et de les améliorer. C'est entre autres le cas d'informations trouvées lors de la validation de la portion d'animation et d'évolution des logiciels. On peut trouver des exemples dans l'article [53] où l'on voit certains patrons d'évolutions relatifs au logiciel *Freemind*. On y voit une classe grossir de plus en plus à travers les versions pour devenir problématique au niveau de la qualité, ou encore des fonctionnalités qui changent de classes. Lors d'un concours d'outils [51] dans le cadre de la conférence *Vissoft* 2007, des observations sur la qualité ont été faites sur le logiciel *Azureus*. De plus, plusieurs logiciels ont été évalués dans le cadre du mémoire de maîtrise d'un collègue [21] sur la détection d'antipatrons de conception à l'aide de *VERSO*. D'ailleurs dans la première partie de la première expérience décrite à la section 6.2, on demande aux sujets d'identifier eux-mêmes de petits phénomènes reliés à la qualité du logiciel qu'ils sont pour la grande majorité en mesure de découvrir en utilisant les représentations graphiques de *VERSO*.

Pour finir, deux études de cas ont été conduites et sont décrites en détail dans les sections 6.3 et 6.4. On peut faire ressortir de ces études que des programmeurs sont facilement capables d'utiliser *VERSO* avec un minimum d'entraînement. Les utilisateurs

sont aussi capables d'utiliser *VERSO* dans un contexte où les tâches sont réelles. Les commentaires sur sa facilité d'utilisation montrent aussi que *VERSO* est un outil que les gens apprécient et qui simplifie le processus de développement. La deuxième expérience a montré que *VERSO* peut être utilisé pour créer un logiciel au complet sans que les tâches de recherche soient dirigées. Les deux expériences montrent aussi que les participants apprécient *VERSO* (8 sur 10 et 6,1 sur 9) et le trouvent très utile pour répondre à des questions dans la pratique. L'intention des utilisateurs d'adopter un outil de visualisation pour les aider à répondre à toutes sortes de questions à l'intérieur du logiciel, y compris des questions sur la qualité et sur les informations générées par les logiciels de contrôle de versions montre l'utilité de *VERSO*. Le principe des multiples vues à l'intérieur d'un même outil a été jugé pertinent par les participants.

VERSO a donc été évalué de différentes façons et les résultats nous permettent de croire qu'il est d'une grande utilité pour l'analyse de code et qu'il peut s'insérer dans un processus de développement en répondant aux multiples questions des programmeurs. L'article de Munzner [67] a permis de voir quel type de validation peut être utilisé pour chaque étape et la couverture des étapes semble complète sauf pour la dernière concernant l'efficacité, qui n'a pas eu besoin d'évaluation précise. En effet, le prototype de *VERSO* répond dans des temps perçus comme raisonnables par les utilisateurs pour un logiciel de plusieurs centaines de classes. Seule la première lecture d'un gros logiciel inconnu prend quelques secondes pour analyser la structure et quelques minutes pour rapatrier l'information *SVN*, le reste des modifications n'étant pas perceptible aux utilisateurs. La navigation dans l'environnement en trois dimensions s'effectue d'ailleurs avec fluidité.

CHAPITRE 7

CONCLUSION

Dans cette thèse, nous avons traité de l'intégration d'une visualisation dans un environnement de développement possédant déjà de multiples fonctionnalités. Les programmeurs sont en mesure d'utiliser *VERSO* pour répondre à leurs questions tant au niveau de la conception que de l'analyse du logiciel, et ce, tout en ajoutant du code, en déboguant ou en essayant de comprendre le code d'une autre personne. La visualisation du logiciel est basée sur une série de métriques qui couvrent différentes facettes du développement. Ces métriques sont calculées et présentées automatiquement pour que les utilisateurs aient une idée des caractéristiques du logiciel en direct lors de sa création et sa modification. Ils sont en mesure de faire cette évaluation à l'aide d'un simple coup d'oeil ou d'une petite analyse. Deux études de cas menées auprès d'utilisateurs de *VERSO* montrent qu'il est efficace et qu'il répond bien à leurs besoins.

7.1 Réalisations

Une première réalisation de cette thèse est le calcul incrémental des métriques en direct pour obtenir un modèle du logiciel à jour qui s'affiche après chaque modification. Les métriques sont calculées en partie à l'aide du modèle interne d'*Eclipse*. Elles couvrent plusieurs aspects dont la complexité, le couplage, la cohésion, les auteurs, la charge de travail sur une entité et les bogues rattachés à une entité. Les métriques sont conservées sous forme d'un vecteur pour chacune des entités : classes, interfaces, méthodes et paquetages. D'autres informations nominales comme une liste de parents et une liste d'éléments liés sont aussi présentes dans le modèle. Ce modèle est très générique et il peut être utilisé avec n'importe quelle technique d'analyse basée sur les métriques. Il est facilement extensible puisque le vecteur de métriques peut s'allonger sans aucune modification nécessaire dans le reste du programme. La visualisation, en outre, est prête à accueillir de nouvelles métriques sans modifier son implantation.

Une deuxième réalisation par rapport à nos travaux antérieurs est la création des vues pour les niveaux de la méthode, des lignes de code et des paquetages. Les paquetages peuvent aussi être présentés à différents niveaux d'imbrication, et il est possible de les ouvrir et de les fermer interactivement pour que plusieurs niveaux différents soient visibles en même temps et ainsi accommoder le niveau de détail requis par les utilisateurs sur les différentes parties du logiciel. Les visualisations énumérées ont été créées dans un but de continuité avec le niveau des classes déjà existant. La forme des boîtes en trois dimensions a donc été conservée. Le niveau des lignes de code conserve aussi le même esprit, mais utilise une représentation en deux dimensions pour permettre une lecture du texte sous-jacent. Il y a aussi la création d'un mécanisme pour visualiser des métriques reliées à différents contextes ou encore les métriques choisies par les utilisateurs. Ceci augmente les informations disponibles dans la visualisation sans pour autant saturer le système visuel des utilisateurs. Des mécanismes pour créer et visualiser des versions du logiciel sont disponibles afin que les utilisateurs suivent l'évolution de leur logiciel en ayant accès à la même quantité et qualité d'informations que dans la visualisation de la version courante.

Une troisième réalisation est la connexion entre les deux premières phases, c'est-à-dire le calcul des métriques en direct et son reflet dans la visualisation par l'entremise d'un modèle commun. Ces deux éléments sont réunis à même l'interface graphique d'*Eclipse* qui a été étendue pour montrer la visualisation dans le même espace qu'un éditeur de texte. Les mécanismes de plugiciels d'*Eclipse* permettent aux deux parties d'échanger leurs informations et d'afficher un résultat cohérent. Les programmeurs n'ont donc pas besoin d'utiliser deux outils et d'alterner leur attention d'une fenêtre à l'autre, ce qui leur permet d'analyser et de continuer leur tâche de programmation en même temps. Les interfaces des deux outils ont été alignées pour afficher les mêmes informations entre autres en ce qui a trait aux mécanismes de sélection. De plus, il est possible de créer des classes, des méthodes et des paquetages à même l'interface graphique. Ceci permet de générer le squelette du logiciel sans écrire de code dans le langage de programmation de plus bas niveau. Quand les éléments sont manipulés dans la vue graphique, les changements sont immédiatement reflétés dans le modèle d'*Eclipse*, le code est compilé,

l'élément apparaît dans l'interface d'*Eclipse* et dans la visualisation de *VERSO*. Tout ça en un instant.

Une quatrième réalisation est le fait d'avoir conduit deux expériences dans le but d'évaluer *VERSO*. La première expérience comportait 28 sujets qui effectuaient des tâches dirigées et qui ont donné leur opinion sur l'utilité de *VERSO*. Puisque les sujets ne connaissaient pas du tout *VERSO*, cette expérience a montré la facilité d'apprentissage de l'outil et que des néophytes étaient en mesure d'accomplir avec succès des tâches directement reliées à l'analyse et aux modifications du logiciel. On a aussi vu que parmi les dix équipes qui ont participé à l'expérience, la grande majorité a trouvé que *VERSO* était utile et agréable pour ces tâches typiques. La deuxième expérience comportait huit sujets qui ont évalué *VERSO* dans un contexte de travail personnel sans tâches directement imposées. Lors de cette expérience, plusieurs variables étaient observées comme l'utilisation de *SVN* et l'emploi du temps à l'aide d'un système de cartes de temps permettant ainsi d'analyser l'utilisation de *VERSO*. Les résultats sont encore une fois prometteurs puisque sept des huit participants étaient prêts à adopter un système de visualisation comme *VERSO* pour travailler dans le futur. La majorité des participants a trouvé qu'il était utile lors du développement. Les sujets ont aussi confirmé la pertinence d'avoir de multiples vues intégrées dans un même outil.

7.2 Contributions

La première contribution de cette thèse est la définition d'un ensemble de vues adaptées au besoin des utilisateurs pour l'analyse et le développement de leur logiciel. En plus de définir ces vues, nous les avons catégorisées selon des cellules dans un cube en trois dimensions (voir figure 3.1). Chacune des cellules représente une vue et la navigation à l'intérieur de cet espace de vues offre un cadre permettant de résoudre des problèmes d'analyse. Les vues ont trois caractéristiques (dimensions), c'est-à-dire le niveau de granularité, le contexte et les différentes versions du logiciel. En conceptualisant les changements de vues comme des déplacements d'une cellule à une autre, on peut à tout moment se situer dans la panoplie de vues, et on peut connaître à tout mo-

ment les vues accessibles à partir d'une vue donnée. Par exemple, si l'on se trouve dans une vue au niveau des paquetages, on passera par le niveau de granularité de la classe avant d'arriver au niveau des méthodes. Ce modèle de définition permet de mieux saisir l'importance des différents éléments qu'il est possible de montrer à l'aide d'une visualisation et d'identifier chaque vue à ses caractéristiques pour lui attitrer plus facilement une tâche.

La deuxième contribution est l'utilisation de multiples vues sur le logiciel qui sont toutes rattachées par le principe de cohérence. Il existe donc un fil conducteur entre ces vues qui contribue à ce que les utilisateurs ne soient pas dépaysés en passant d'une vue à l'autre. Le principe de cohérence est présent tout aussi bien dans l'axe de la granularité et des contextes que dans celui de la visualisation de l'historique des versions. Pour l'axe de la granularité, le principe de cohérence permet d'obtenir plus de détails sur un élément observé ou plus d'informations sur les éléments autour de lui. De plus, les vues dans les différentes granularités ont été développées avec des concepts similaires (une boîte avec les mêmes caractéristiques graphiques) pour que les utilisateurs fassent appel aux mêmes schémas de perception dans un niveau ou un autre. Des indices graphiques réduisent l'apparition d'ambiguïtés au sujet du niveau dans lequel on se trouve. Pour l'axe du contexte, on conserve aussi la cohérence lors des déplacements puisqu'on visualise les mêmes éléments qui sont disposés sur le plan de la même façon. Puisque la caméra demeure fixe sur les mêmes éléments, les utilisateurs risquent moins de perdre leur concentration sur la tâche qu'ils sont en train d'accomplir. Pour l'évolution, la cohérence se retrouve à l'intérieur même du processus de création du logiciel. Effectivement, le logiciel grandit et est modifié de version en version en gardant toujours une base par rapport à la prochaine version, ce qui implique que la visualisation sera modifiée au fil des versions de la même façon puisqu'elle suit les nouvelles données du logiciel.

Le fait que *VERSO* soit intégré dans *Eclipse* joue un rôle important dans la cohérence puisque les utilisateurs n'ont pas à apprendre plusieurs outils et ils n'ont pas à se réapproprier les concepts visuels chaque fois qu'ils veulent faire une analyse. De plus, il n'est pas nécessaire de changer de fenêtre ou d'avoir une fenêtre spécifique pour chaque tâche qu'on désire accomplir.

La troisième contribution est le développement de vues efficaces qui tiennent compte des principes de perception et permettent au système visuel humain de détecter certains phénomènes instantanément, comme une couleur ou une hauteur anormale reflétant des situations concrètes sur le logiciel. Les différentes vues utilisent des caractéristiques graphiques qui réduisent les interférences les unes sur les autres tout en montrant une bonne quantité d'informations. De plus, la visualisation, avec l'aide principalement des algorithmes de placement, permet de montrer une vue d'ensemble d'un logiciel complet tout en étant en mesure de faire des tâches d'analyse assez précises. Ensuite, l'interaction avec *VERSO* permet aux utilisateurs de personnaliser leur analyse et de visualiser les combinaisons de métriques de leur choix. Par ailleurs, un système de calcul de métriques efficace a été développé permettant de calculer les informations en direct et donc de les visualiser en direct. *VERSO* calcule des données de façon incrémentale pour obtenir les informations sans délai. Le système de calcul des métriques est facilement extensible et utilisable pour toutes sortes d'applications. Il a d'ailleurs été utilisé et modifié par quelques collègues pour générer des données aux fins d'analyses dont certaines ont été publiées [6, 22, 33, 45]. *VERSO* a été utilisé par les universités de Nancy et de Malaga en plus d'avoir servi dans des analyses de qualité par la société française *SNCF*.

La quatrième contribution consiste à combiner ces composantes pour créer une nouvelle façon de voir la programmation. Les activités d'analyse de programmation ont toujours été vues séparément, et ces problèmes ont été traités par des outils différents. Avec *VERSO*, il est possible de programmer tout en suivant les caractéristiques de son logiciel, ou encore d'utiliser la visualisation pour interpréter le contexte et évaluer l'endroit de la prochaine action à poser. Encore une fois, le fait d'avoir une seule interface où sont regroupés des services intégrés aide à réunir ces tâches et incite les utilisateurs à se soucier de la qualité dès le début du développement. La possibilité d'ajouter des éléments à même la représentation graphique atténue encore plus la ligne entre l'analyse et le développement.

Cet outil unique a été évalué de façon empirique à deux reprises. Ceci a permis de montrer son utilité dans des contextes comparables à ceux de développement réel du logiciel. De plus, ceci a permis de montrer qu'il est possible de valider un outil de

visualisation aussi complexe que *VERSO*. Les commentaires recueillis au cours des deux expériences sont non seulement pertinents pour l'amélioration de *VERSO*, mais peuvent aussi être utilisés pour montrer les besoins d'utilisateurs réels concernant des outils de visualisation de tout genre.

7.3 Travaux futurs

Il faut comprendre que *VERSO* reste un prototype. Donc, sa stabilité, l'esthétique de son interface et ses performances ne sont pas encore tout à fait optimisées. Il faudrait donc choisir les fonctionnalités importantes dans *VERSO* et les améliorer pour qu'elles soient stables et efficaces. D'ailleurs, des améliorations de *VERSO* par d'autres développeurs au sein de l'équipe devraient être intégrées puisqu'elles lui ont apporté des fonctionnalités additionnelles intéressantes.

Dans les améliorations discutées au sein de l'équipe, il y a entre autres celle de Simon Bouvier [6], qui consiste en la représentation explicite des liens dans le logiciel. Bien qu'il soit possible de visualiser les liens dans la version actuelle, on peut seulement voir les éléments reliés à l'élément sélectionné. Il faut donc explorer plusieurs éléments pour se faire une idée des ramifications entre les différentes parties du logiciel. Une représentation explicite des liens est difficile à cause de l'encombrement que cela crée. Il existe par contre des techniques, notamment par le groupement hiérarchique de liens (entrepris par Simon Bouvier), qui peuvent diminuer l'impact de cet enchevêtrement. La représentation explicite des liens donne une image plus complète du couplage d'un logiciel et serait probablement plus indiquée pour répondre à certaines tâches d'analyse qui s'attardent principalement sur les liens dans le système.

La création et la suppression d'éléments pourraient être améliorées pour permettre une plus grande interactivité avec le logiciel visualisé. En effet, bien que *VERSO* permette de créer des éléments dans la vue graphique, il n'est pas possible aujourd'hui d'ajouter un comportement à ces éléments. Il y a bien eu quelques essais de programmation graphique qui ont souvent été très proches des langages de haut niveau actuels, mais ces essais sont restés très modestes et se sont surtout adressés à une clientèle restreinte.

La déclaration d'un langage complètement graphique est très complexe et demanderait probablement plus d'une thèse complète à développer. Plusieurs perfectionnements dans l'automatisation pourraient s'ajouter à partir des représentations graphiques déjà existantes. Par exemple, il serait possible d'avoir une interface graphique pour certains *refactorings* classiques qui sont déjà implantés dans quelques éditeurs. Il serait aussi intéressant d'avoir des indices graphiques lors de la complétion automatique qui existe déjà dans plusieurs outils de développement pour visualiser les éléments plutôt que de se fier uniquement au texte.

Jusqu'à maintenant, *VERSO* a été utilisé principalement pour représenter l'analyse du logiciel, surtout du point de vue de la structure du logiciel et de ses intervenants. De par sa versatilité, *VERSO* pourrait être utilisé avec plusieurs types de source de données pour représenter le logiciel dans d'autres domaines d'activités. On pense entre autres à des informations sur la performance du logiciel sur de grands jeux de tests ou encore l'étude du comportement dynamique d'un logiciel alors qu'il est exécuté. Bien que *VERSO* présente des classes statiques, rien n'empêche de montrer des informations sur des instances de classe si les informations sont disponibles. Les changements aux fichiers d'entrée et au modèle de *VERSO* seraient minimes pour parvenir à ce genre de résultats. De plus, *VERSO* organise les informations de façon hiérarchique et présente des éléments qui possèdent un vecteur de métriques. La majorité des informations encodée sous cette forme peuvent être représentées avec *VERSO*. La visualisation est donc polyvalente et devrait être utilisée pour étudier n'importe quel type d'informations ou encore pourrait être utilisée comme objet de comparaison pour d'autres visualisations plus spécialisées.

Pour finir, *VERSO* a été évalué de différentes manières, mais toujours sur de courtes périodes. Il faudrait que plusieurs professionnels l'utilisent durant plusieurs projets de grande envergure pour qu'il soit possible d'évaluer son impact sur la programmation. Bien souvent, ces études s'échelonnant dans le temps sont très difficiles à réaliser et demandent des ressources considérables. C'est encore plus difficile quand il s'agit d'études comparatives. D'ailleurs, dans le monde des environnements de développement, les outils sont souvent adoptés avant qu'ils ne soient évalués de façon aussi exhaustive. Une

étude à long terme dans un contexte réel sur une longue période ne serait pas seulement bénéfique pour *VERSO*, mais aiderait à mieux comprendre les besoins des utilisateurs pour les visualisations et pour les environnements de développement évolués.

7.4 Quelques réflexions sur la visualisation du logiciel dans l'avenir

Cette section présente quelques réflexions sur la visualisation du logiciel tirée de l'expérience acquise durant notre thèse de doctorat. Il s'agit d'observations et d'opinions sur le domaine ainsi que des idées sur les orientations futures.

Dans un premier temps, les acteurs de la visualisation du logiciel et de la qualité du logiciel devront s'interroger sur les raisons pour lesquelles ces domaines ne font qu'une très lente percée dans les milieux industriels. En effet, malgré les promesses d'économies potentielles des ingénieurs logiciels, trop peu d'analyses sont faites dans l'industrie concernant la maintenance d'un logiciel. On s'en tient souvent à des tests de correction ou à des tests de performance. Dans plusieurs entreprises, il existe des exigences exprimées sous la forme de conventions de code ou conventions liées aux commentaires. Ces mécanismes devraient améliorer les lectures du code subséquentes, mais il demeure qu'elles ont trop peu d'influence et qu'elles restent au niveau purement syntaxique. Pour ce qui est de la visualisation, les nouveaux environnements de développement contenant des visualisations simples comme les arbres de sélection semblent être adoptés rapidement sans même qu'on ait besoin de prouver leur efficacité véritable par rapport aux anciens éditeurs se concentrant sur le texte simple. Par contre, les environnements de visualisation complexes sont peu utilisés par les programmeurs ou les analystes de l'industrie. Ils sont utilisés dans des contextes expérimentaux ou de façon sporadique par curiosité plus que pour leur apport dans un travail concret. Ce fait survient malgré les commentaires souvent très positifs qui sont récoltés dans les expériences en visualisation. Dans le futur, il faudra donc analyser cette réticence et trouver le moyen d'obtenir une plus grande adoption de ces outils en se rapprochant des besoins reliés à la programmation de tous les jours.

Les ingénieurs logiciels soucieux de la qualité sont souvent vus d'un mauvais oeil par

les autres programmeurs puisqu'ils sont appelés à critiquer le code de leurs collègues. Ils peuvent aussi être mal vus de la part des gestionnaires puisque leurs commentaires concernent souvent des parties de logiciels qui fonctionnent déjà correctement et dont les modifications engendreraient des coûts supplémentaires à court terme. Une façon de remédier à ce problème serait de remettre la responsabilité de l'évaluation de la qualité au point de vue de la maintenance dans les mains des programmeurs de première ligne en leur donnant les outils nécessaires. Il serait aussi intéressant de rattacher un prix à ce genre de qualité, car en ce moment on considère pratiquement uniquement le nombre de fonctionnalités ou le nombre de lignes de code.

Enfin, les visualisations devront être simplifiées pour être acceptées à grande échelle par la communauté des développeurs. Il faut garder en tête que la visualisation devrait résumer et simplifier les informations disponibles et non pas la complexifier. S'il faut mettre du temps pour apprendre un outil de visualisation, il faut qu'il y ait un retour en productivité. En structurant les informations différemment, on peut faire ressortir certains phénomènes de ces dernières. Par contre, si seulement une poignée d'initiés sont capables d'interpréter ces images, la complexité supplémentaire n'aura servi à rien. Les informations doivent aussi être disponibles instantanément pour que les utilisateurs n'aient pas à changer de contexte pour réagir à ce que la visualisation présente. La simplicité se transformera alors en accessibilité, puis en adoption. Il faudra aussi que la visualisation se rapproche des préoccupations des programmeurs, c'est-à-dire la sémantique des programmes et les fonctionnalités. Dans cette optique, le fichier ne restera probablement pas l'élément d'intérêt principal dans les visualisations du futur. Il s'agit plus ou moins d'un vestige laissé par la façon traditionnelle de programmer avec des éditeurs de texte et des compilateurs qu'on lance manuellement. Tout comme la compilation maintenant faite automatiquement par les environnements de développement, et qui pointe même les erreurs aussitôt qu'elles sont écrites, la visualisation de la qualité du logiciel et l'analyse en général devront s'effacer dans l'outil de programmation pour former un tout homogène.

BIBLIOGRAPHIE

[1] Jehad Al Dallal et Lionel C. Briand. An object-oriented high-level design-based class cohesion metric. *Inf. Softw. Technol.*, 52:1346–1361, December 2010.

[2] Thomas Ball et Stephen G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

[3] Douglas Bell. *Software Engineering, A Programming Approach*. Addison-Wesley, 2000.

[4] Dirk Beyer et Ahmed E. Hassan. Animated visualization of software history using evolution storyboards. Dans *WCRE '06 : Proceedings of the 13th Working Conference on Reverse Engineering*, pages 199–210, 2006.

[5] Thomas Bladh, David A. Carr et Matjaz Kljun. The effect of animated transitions on user navigation in 3d tree-maps. Dans *IV '05 : Proceedings of the Ninth International Conference on Information Visualisation*, pages 297–305, 2005.

[6] Simon Bouvier. Utilisation de la visualisation interactive pour l'analyse des dépendances dans les logiciels. M.sc. thesis, Département d'Informatique et Recherche Opérationnelle, Université de Montréal, 2011.

[7] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra et Joseph J. LaViola, Jr. Code bubbles : rethinking the user interface paradigm of integrated development environments. Dans *ICSE '10 : Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 455–464, 2010.

[8] Sabrina Bresciani et Martin J. Eppler. The benefits of synchronous collaborative information visualization : Evidence from an experimental evaluation. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1073–1080, novembre 2009.

[9] Lionel C. Briand, John W. Daly et Jürgen Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1), 1998.

[10] Lionel C. Briand, Yvan Labiche, Massimiliano Di Penta et Han (Daphne) Yan-Bondoc. An experimental investigation of formality in uml-based development. *IEEE Trans. Softw. Eng.*, 31:833–849, October 2005.

[11] Bugzilla. Bugzilla, 2010. `http ://www.bugzilla.org/`.

[12] Shyam R. Chidamber et Chris F. Kemerer. A metric suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):293–318, June 1994.

[13] J. Chuang, D. Weiskopf et T. Moller. Hue-preserving color blending. *IEEE Transactions onVisualization and Computer Graphics*, 15(6):1275 –1282, nov.-dec. 2009.

[14] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts et Kevin Wampler. A system for graph-based visualization of the evolution of software. Dans *SoftVis '03 : Proceedings of the 2003 ACM symposium on Software visualization*, page 77, 2003.

[15] Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove et Kevin Christiansen. Alice : lessons learned from building a 3d system for novices. Dans *CHI '00 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 486–493, 2000.

[16] Bas Cornelissen, Andy Zaidman, Arie van Deursen et Bart Van Rompaey. Trace visualization for program comprehension : A controlled experiment. Dans *ICPC '09 : IEEE International Conference on Program Comprehension*, pages 100–109, 2009.

[17] Barthélémy Dagenais, Harold Ossher, Rachel K. E. Bellamy, Martin P. Robillard et Jacqueline P. de Vries. Moving into a new software project landscape. Dans *ICSE*

'10 : Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pages 275–284, 2010.

[18] Marco D'Ambros et Michele Lanza. Reverse engineering with logical coupling. Dans *WCRE '06 : Proceedings of the 13th Working Conference on Reverse Engineering*, pages 189–198, 2006.

[19] Marco D'Ambros et Michele Lanza. Software bugs and evolution : A visual approach to uncover their relationship. Dans *CSMR '06 : Proceedings of the Conference on Software Maintenance and Reengineering*, pages 229–238, Washington, DC, USA, 2006. IEEE Computer Society.

[20] Marco D'Ambros, Michele Lanza et Harald Gall. Fractal figures : Visualizing development effort for cvs entities. Dans *VisSoft '05 : Proceedings IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 46–51, septembre 2005.

[21] Karim Dhambri. Détection visuelle d'anomalies de conception dans les programmes orientés objet. Mémoire de maîtrise, Département d'Informatique et Recherche Opérationnelle, Université de Montréal, décembre 2007.

[22] Karim Dhambri, Houari A. Sahraoui et Pierre Poulin. Visual detection of design anomalies. Dans *CSMR '08 : European Conference on Software Maintenance and Reengineering*, pages 279–283, 2008.

[23] Marian Dörk, Sheelagh Carpendale, Christopher Collins et Carey Williamson. Visgets : Coordinated visualizations for web-based information exploration and discovery. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1205–1212, 2008.

[24] Geoffrey Draper et Richard Riesenfeld. Who votes for what ? a visual query language for opinion data. *IEEE Transactions on Visualization and Computer Graphics*, 14:1197–1204, 2008.

[25] Eclipse. Eclipse, 2010. `http ://www.eclipse.org/`.

[26] Stephen G. Eick, Joseph L. Steffen et Jr. Eric E. Sumner. Seesoft – a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[27] Jean-Daniel Fekete et Catherine Plaisant. Interactive information visualization of a million items. Dans *INFOVIS '02 : Proceedings of the IEEE Symposium on Information Visualization*, page 117, 2002.

[28] Michael Fischer et Harald Gall. Evograph : A lightweight approach to evolutionary and structural analysis of large software systems. Dans *WCRE '06. 13th Working Conference on Reverse Engineering, 2006.*, pages 179 –188, oct. 2006.

[29] Thomas Fritz et Gail C. Murphy. Using information fragments to answer the questions developers ask. Dans *ICSE '10 : Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 175–184, 2010.

[30] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[31] Google. Google maps, 2010. `http ://maps.google.ca/`.

[32] Nathan Gossett et Baoquan Chen. Paint inspired color mixing and compositing for visualization. Dans *INFOVIS '04 : Proceedings of the IEEE Symposium on Information Visualization*, pages 113–118, 2004.

[33] Hassen Grati, Houari A. Sahraoui et Pierre Poulin. Extracting sequence diagrams from execution traces using interactive visualization. Dans *WCRE '10 : Working Conference on Reverse Engineering*, pages 87–96, 2010.

[34] Orla Greevy, Stéphane Ducasse et Tudor Gîrba. Analyzing software evolution through feature views : Research articles. *J. Softw. Maint. Evol.*, 18:425–456, November 2006.

[35] Dick Hamlet et Joe Maybee. *The Engineering of Software : A Technical Guide for the Individual*. Addison-Wesley, 2000.

[36] Christopher G. Healey. Choosing effective colours for data visualization. Dans *Vis'96 : Proceedings of Visualization*, pages 263 –270, oct. 1996.

[37] Christopher G. Healey. Building a perceptual visualisation architecture. *Behaviour and Information Technology*, 19:2000, 1998.

[38] Christopher G. Healey, Kellogg S. Booth et James T. Enns. Visualizing real-time multivariate data using preattentive processing. *ACM Trans. Model. Comput. Simul.*, 5(3):190–221, 1995.

[39] Christopher G. Healey, Kellogg S. Booth et James T. Enns. High-speed visual estimation using preattentive processing. *ACM Trans. Comput.-Hum. Interact.*, 3 (2):107–135, 1996.

[40] Christopher G. Healey et James T. Enns. Large datasets at a glance : Combining textures and colors in scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):145–167, 1999.

[41] Danny Holten. Hierarchical edge bundles : Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12: 741–748, September 2006.

[42] Andri Ioannidou, Alexander Repenning et David C. Webb. Agentcubes : Incremental 3d end-user development. *J. Vis. Lang. Comput.*, 20(4):236–251, 2009.

[43] Yuntao Jia, Jared Hoberock, Michael Garland et John Hart. On the visualization of social and other scale-free networks. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1285–1292, 2008.

[44] Brian Johnson et Ben Shneiderman. Tree-maps : a space-filling approach to the visualization of hierarchical information structures. Dans *VIS '91 : Proceedings of the 2nd conference on Visualization '91*, pages 284–291, 1991.

[45] Jamel Eddine Jridi. Formulation interactive des requêtes pour l'analyse et la compréhension du code source. Mémoire de maîtrise, Université de Montréal, 2010.

[46] Claire Knight et Malcolm Munro. Virtual but visible software. Dans *IV '00 : Proceedings of the International Conference on Information Visualisation*, page 198, 2000.

[47] Arno Krueger, Christoph Kubisch, Bernhard Preim et Gero Strauss. Sinus endoscopy - application of advanced gpu volume rendering for virtual endoscopy. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1491–1498, 2008.

[48] Adrian Kuhn, David Erni, Peter Loretan et Oscar Nierstrasz. Software cartography : thematic software visualization with consistent layout. *J. Softw. Maint. Evol.*, 22: 191–210, April 2010.

[49] Adrian Kuhn, David Erni et Oscar Nierstrasz. Embedding spatial software visualization in the ide : an exploratory study. Dans *SOFTVIS '10 : Proceedings of the 5th international symposium on Software visualization*, pages 113–122, 2010.

[50] Guillaume Langelier. Visualisation de la qualité des logiciels de grandes tailles. Mémoire de maîtrise, Université de Montréal, 2006.

[51] Guillaume Langelier et Karim Dhambri. Visual analysis of azureus using verso. Dans *VisSoft '07 : International Workshop on Visualizing Software for Understanding and Analysis*, pages 163–164, 2007.

[52] Guillaume Langelier, Houari Sahraoui et Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. Dans *ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 214–223, 2005.

[53] Guillaume Langelier, Houari Sahraoui et Pierre Poulin. Exploring the evolution of software quality with animated visualization. Dans *VLHCC '08 : Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 13–20, 2008.

[54] Michele Lanza et Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29:782–795, September 2003.

[55] Michele Lanza et Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. Dans *Langage et modèles à objets*, pages 135–149, 2002.

[56] Adrian Lienhard, Tudor Girba, Orla Greevy et Oscar Nierstrasz. Test blueprints - exposing side effects in execution traces to support writing unit tests. Dans *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pages 83–92, 2008.

[57] Gerard Lommerse, Freek Nossin, Lucian Voinea et Alexandru Telea. The visual code navigator : An interactive toolset for source code investigation. Dans *INFOVIS '05 : Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization*, page 4, Washington, DC, USA, 2005. IEEE Computer Society.

[58] Mircea Lungu, Michele Lanza, Tudor Gîrba et Romain Robbes. The small project observatory : Visualizing software ecosystems. *Sci. Comput. Program.*, 75:264–275, April 2010.

[59] Alan M. MacEachren. *How Maps Work : Representation, Visualization and Design*. Guilford Press, New York, 1995.

[60] Andrian Marcus, Louis Feng et Jonathan I. Maletic. 3d representations for software visualization. Dans *SoftVis '03 : Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–ff, 2003.

[61] Lucia Mason. Fostering understanding by structural alignement as a route to analogical learning. *Instructional Science*, 32(6):293–318, November 2004.

[62] David Mayerich, Louise Abbott et John Keyser. Visualization of cellular and microvascular relationships. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1611–1618, 2008.

[63] Thomas J. McCabe. A complexity measure. Dans *ICSE '76 : Proceedings of the 2nd international conference on Software engineering*, pages 407–, 1976.

[64] Cédric Mesnage et Michele Lanza. White coats : Web-visualization of evolving software in 3d. Dans *VISSOFT '05 : Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, page 15, 2005.

[65] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien et Anne-Francoise Le Meur. Decor : A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36:20–36, 2010.

[66] F.C. Muller-Lyer. Optische urteilstäuschungen. *Archiv für Physiologie Suppl.*, pages 263–270, 1889.

[67] Tamara Munzner. A nested process model for visualization design and validation. *IEEE Transactions on Visualization and Computer Graphics*, 15:921–928, November 2009.

[68] Quang Vinh Nguyen et Mao Lin Huang. A space-optimized tree visualization. Dans *INFOVIS '02 : Proceedings of the IEEE Symposium on Information Visualization*, page 85, 2002.

[69] Oscar Nierstrasz, Stéphane Ducasse et Tudor Gĭrba. The story of moose : an agile reengineering environment. *SIGSOFT Softw. Eng. Notes*, 30:1–10, September 2005.

[70] Stephen C. North. Incremental layout in dynadag. Dans *GD '95 : Proceedings of the Symposium on Graph Drawing*, pages 409–418, London, UK, 1996. Springer-Verlag.

[71] Michael Ogawa et Kwan-Liu Ma. code_swarm : A design study in organic software visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15: 1097–1104, 2009.

[72] OpenGL. Opengl, 2010. `http ://www.opengl.org/`.

[73] Michael J. Pacione, Marc Roper et Murray Wood. A novel software visualisation model to support software comprehension. Dans *WCRE '04 : Proceedings of the 11th Working Conference on Reverse Engineering*, pages 70–79, 2004.

[74] Thomas Panas, Rebecca Berrigan et John Grundy. A 3D metaphor for software production visualization. Dans *IV'03 : Proceedings International Conference on Information Visualization*, pages 314–319, 2003.

[75] Daniel Patel, Christopher Giertsen, John Thurmond, John Gjelberg et Eduard Grller. The seismic analyzer : Interpreting and illustrating 2d seismic data. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1571 –1578, nov.-dec. 2008.

[76] Martin Pinzger, Harald Gall, Michael Fischer et Michele Lanza. Visualizing multiple evolution metrics. Dans *SoftVis '05 : Proceedings of the 2005 ACM symposium on Software Visualization*, pages 67–75, 2005.

[77] V.S. Ramachandran et William Hirstein. The science of art : a neurological theory of aesthetic experience. *Journal of Consciousness Studies*, 6:15–51(37), 1999.

[78] Ronald A Rensink. Change detection. *Annual Review of Psychology*, 53:245–277, 2002.

[79] Ronald A. Rensink. The dynamic representation of scenes. *Visual Cognition*, 7: 17–42(26), January 2000.

[80] Ronald A. Rensink, J. Kevin O'Regan et James J. Clark. To See or not to See : The Need for Attention to Perceive Changes in Scenes. *Psychological Science*, 8(5): 368–373, 1997.

[81] Alexander Repenning, Andri Ioannidou et John Zola. Agentsheets : End-user programmable simulations. *Journal of Artificial Societies and Social Simulation*, 3(3), 2000.

[82] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman et Yasmin Kafai. Scratch : programming for all. *Commun. ACM*, 52: 60–67, November 2009.

[83] Juergen Rilling et S.P. Mudur. 3d visualization techniques to support slicing-based program comprehension. *Computers & Graphics*, 29(3):311–329, 2005.

[84] George Robertson, Roland Fernandez, Danyel Fisher, Bongshin Lee et John Stasko. Effectiveness of animation in trend visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1325–1332, 2008.

[85] Martin P. Robillard et Gail C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16, February 2007.

[86] Pierre N. Robillard. Opportunistic problem solving in software engineering. *IEEE Software*, 22:60–67, November 2005.

[87] D. Rothlisberger, M. Harry, A. Villazon, D. Ansaloni, W. Binder, O. Nierstrasz et P. Moret. Augmenting static source views in ides with dynamic metrics. Dans *ICSM '09 : IEEE International Conference on Software Maintenance*, pages 253 –262, 2009.

[88] Mariam Sensalire, Patrick Ogao et Alexandru Telea. Evaluation of software visualization tools : Lessons learned. Dans *VISSOFT '09 : 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 19 –26, 2009.

[89] Maruthappan Shanmugasundaram, Pourang Irani et Carl Gutwin. Can smooth view transitions facilitate perceptual constancy in node-link diagrams ? Dans *GI '07 : Proceedings of Graphics Interface 2007*, pages 71–78, 2007.

[90] Roger C. Tam, Christopher G. Healey, Borys Flak et Peter Cahoon. Volume rendering of abdominal aortic aneurysms. Dans *VIS '97 : Proceedings of the 8th conference on Visualization '97*, page 43, 1997.

[91] Maurice Termeer, Christian F. J. Lange, Alexandru Telea et Michel R. V. Chaudron. Visual exploration of combined architectural and metric information. Dans *VISSOFT '05 : Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, page 11, 2005.

[92] Melanie Tory, Colin Swindells et Rebecca Dreezer. Comparing dot and landscape spatializations for visual memory differences. *IEEE Transactions on Visualization and Computer Graphics*, 15:1033–1040, November 2009.

[93] Edward R. Tufte. *Visual explanations : images and quantities, evidence and narrative*. Graphics Press, Cheshire, CT, USA, 1997.

[94] Colin Ware. Quantitative texton sequences for legible bivariate maps. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1523–1530, 2009.

[95] Richard Wettel et Michele Lanza. Program comprehension through software habitability. Dans *ICPC '07 : Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 231–240, 2007.

[96] Richard Wettel et Michele Lanza. Visual exploration of large-scale system evolution. Dans *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 219–228, 2008.

[97] Jingwei Wu, Richard C. Holt et Ahmed E. Hassan. Exploring software evolution using spectrographs. Dans *WCRE '04 : Proceedings of the 11th Working Conference on Reverse Engineering*, pages 80–89. IEEE Computer Society, 2004.

[98] Jeff Zacks, Ellen Levy, Barbara Tversky et Diane J. Schiano. Reading bar graphs : Effects of extraneous depth cues and graphical context. *Journal of Experimental Psychology : Applied*, 4(2):119–138, 1998.

Annexe I

Document d'expérimentation

Équipe : ____

Partie 0 – Tutoriel sur l'outil de visualisation intégré à *Eclipse : Verso*.

Partie A – Manipulation de l'outil.

1. Répondez aux questions d'analyse suivantes :
   a. Quel est le package le plus important? Pourquoi?

      ____

      ____

      ____

   b. Il y a un package obsolète dans le système. Lequel? Pourquoi?

      ____

      ____

      ____

   c. Sur quelle classe il y a eu le plus de travail selon vous? Pourquoi?

      ____

      ____

      ____

   d. Si vous aviez la possibilité de faire le *refactoring* de deux ou trois classes seulement. lesquelles choisiriez-vous? Pourquoi?

      ____

      ____

      ____

2. Faites la correction d'un bug
    a. Les résultats pour CBO donnent toujours des valeurs correctes ou supérieures à ce qui est attendu. Le problème apparaît lors de la création des métriques. Dans ce cas précis, on considère que CBO est le nombre de classes différentes appelées par une classe donnée.
        i. Localisez ce bug.
        ii. Corrigez ce bug.
        iii. Faites une nouvelle version après votre modification (Mark As Version)
        iv. Jugez-vous que votre modification a amélioré, détérioré ou gardé stable la qualité du code (du point de vue de la maintenance)?

        ____

        ____

        ____

    b. Lors de la lecture des bugs depuis un fichier, le système ne trouve systématiquement pas les Éléments (classes) et les packages. La liste des entités reliées reste donc toujours vide. Aucun bug n'apparaît durant la visualisation non plus.
        i. Localisez ce bug.
        ii. Corrigez ce bug.
        iii. Faites une nouvelle version après votre modification (Mark As Version)
        iv. Jugez-vous que votre modification a amélioré, détérioré ou gardé stable la qualité du code (du point de vue de la maintenance)?

        ____

        ____

        ____

3. Ajout de fonctionnalité
    a. Dans la vue sur les métriques de contrôle (SVN), on voit qu'il existe la métrique *MainAuthor* qui nous permet de connaître la personne qui travaille le plus sur une classe, un package ou une méthode. Nous aimerions cette fois avoir la métrique *LastAuthor* qui nous indique la dernière personne à avoir travaillé sur une entité. (Vous pouvez uniquement implanter la méthode pour les Éléments(Classes))
        i. Localisez les endroits où faire les modifications.
        ii. Faites les modifications.
        iii. Faites une nouvelle version après votre modification (Mark As Version)
        iv. Jugez-vous que votre modification a amélioré, détérioré ou gardé stable la qualité du code (du point de vue de la maintenance)?

____

____

____

4. *Refactoring*
    a. Les Packages et les Systèmes se comportent souvent de la même façon. Dans les deux cas, ils peuvent contenir une hiérarchie de packages et d'éléments. Étudiez la possibilité de créer un parent commun (ou une hiérarchie commune aux deux classes et commencez votre implantation.)
        i. Localisez les endroits où faire les modifications
        ii. Faites les modifications
        iii. Faites le commit de vos modifications.
        iv. Jugez-vous que votre modification a amélioré, détérioré ou gardé stable la qualité du code (du point de vue de la maintenance)?

____

____

____

Partie B – Questionnaire

1. Pour chacune des questions présentées dans la partie A, cotez l'utilité de Verso pour répondre aux exigences de la question (1 – pas utile, 10 – très utile). Encerclez votre réponse
   - Q1a 1 2 3 4 5 6 7 8 9 10
   - Q1b 1 2 3 4 5 6 7 8 9 10
   - Q1c 1 2 3 4 5 6 7 8 9 10
   - Q1d 1 2 3 4 5 6 7 8 9 10
   - Q2a 1 2 3 4 5 6 7 8 9 10
   - Q2b 1 2 3 4 5 6 7 8 9 10
   - Q3a 1 2 3 4 5 6 7 8 9 10
   - Q4a 1 2 3 4 5 6 7 8 9 10

2. Pour chacune des questions présentées dans la partie A, déterminez le pourcentage approximatif d'utilisation de verso face aux fonctionnalités déjà présentes dans *Eclipse*. (Exemple : *Verso* : X%, *Eclipse* : (100-X)%)
   - Q1a *Verso* : ______ *Eclipse* : ______
   - Q1b *Verso* : ______ *Eclipse* : ______
   - Q1c *Verso* : ______ *Eclipse* : ______
   - Q1d *Verso* : ______ *Eclipse* : ______
   - Q2a *Verso* : ______ *Eclipse* : ______
   - Q2b *Verso* : ______ *Eclipse* : ______
   - Q3a *Verso* : ______ *Eclipse* : ______
   - Q4a *Verso* : ______ *Eclipse* : ______

3. Quelles fonctionnalités (tous outils confondus) ont été les plus utiles lors de l'expérience?

    ____

4. Quelles fonctionnalités de *Verso* ont été les plus utiles lors de l'expérience.

    ____

5. Cotez l'utilité générale de *Verso*. (1 – pas utile, 10 – très utile)

    1 2 3 4 5 6 7 8 9 10

6. Y-a-t-il des fonctionnalités absentes de Verso que vous auriez aimé utiliser lors de l'expérience.

    ____

    ____

    ____

7. Commentaires et Suggestions sur Verso

    ____

    ____

    ____

Annexe II

Questionnaire de l'expérience préalable au projet

Expérience préalable au projet

Nom, Prénom : ______________________________ login Diro : ____________

Pour chacune des questions, choisissez la réponse qui correspond le mieux à votre situation. Ceci n'est pas une évaluation.

1. Quel est votre niveau d'expérience en programmation?
   a. Pas d'expérience / Aucun travail / 0 ligne de code
   b. Peu d'expérience / 1 ou 2 petits travaux sous-gradués / < 1000 lignes de code
   c. Expérience significative / plusieurs petits travaux sous-gradués ou 1 à 2 gros projets / < 5000 lignes de code
   d. Expert / l'équivalent de plus d'une année d'expérience / > 5000 lignes de code
2. Quel est votre niveau d'expérience en Java?
   a. Pas d'expérience / Aucun travail / 0 ligne de code
   b. Peu d'expérience / 1 ou 2 petits travaux sous-gradués / < 1000 lignes de code
   c. Expérience significative / plusieurs petits travaux sous-gradués ou 1 à 2 gros projets / < 5000 lignes de code
   d. Expert / l'équivalent de plus d'une année d'expérience / > 5000 lignes de code
3. Quel est votre niveau d'expérience avec l'IDE Eclipse?
   a. Pas d'expérience / Aucun travail / 0 ligne de code
   b. Peu d'expérience / 1 ou 2 petits travaux sous-gradués / < 1000 lignes de code
   c. Expérience significative / plusieurs petits travaux sous-gradués ou 1 à 2 gros projets / < 5000 lignes de code
   d. Expert / l'équivalent de plus d'une année d'expérience / > 5000 lignes de code
4. Quel est votre niveau d'expérience avec le travail en équipe à travers les logiciels de contrôle de version?
   a. Pas d'expérience
   b. Quelques projets
   c. L'équivalent de plus d'une année d'expérience
5. Quel est votre niveau d'expérience avec les réseaux de Petri?
   a. Aucune
   b. Vous avez entendu parler de la théorie il y a plus ou moins longtemps
   c. Vous connaissez les réseaux de Petri et avez appliqué la théorie dans plusieurs exercices ou dans un projet
   d. Vous êtes un expert des réseaux de Petri

Annexe III

Projet (première partie)

Introduction

Les réseaux de Petri sont des modèles mathématiques servant à représenter des processus utilisant des ressources et qui servent à leur tour à générer ou non des ressources. Les réseaux de Petri sont utilisés pour vérifier certaines propriétés d'un système comme l'absence de blocage ou encore la vivacité. Ils aident à prouver formellement qu'un système fonctionne sans avoir recours à des tests qui, même exhaustifs, restent une façon approximative de vérifier un programme.

Il existe plusieurs définitions pour les réseaux de Petri temporisés. Dans le contexte de ce projet, nous allons tout simplement appliquer un nombre d'unités de temps à chacune des transitions. Le temps avancera donc unité par unité pour représenter le temps qui passe et permettra d'animer le graphe. Il serait possible de représenter le temps à l'aide de réseaux de Petri traditionnels au coût d'une complexité (extrêmement) accrue. De plus, un système de priorisation des transitions sera utilisé. Ceci retire le principe de non-déterminisme des réseaux de Petri classiques, mais permet de faire des animations plus fluides en réduisant ou en éliminant l'intervention de l'utilisateur.

Le but du projet sera de travailler en équipe à la confection d'un éditeur graphique de réseaux de Petri temporisés. Pour ce faire, vous allez utiliser le logiciel-cadre JHotdraw qui sert à développer des éditeurs graphiques. Vous devrez donc comprendre le logiciel-cadre, faire la conception de l'éditeur, faire son développement, le tester et le déboguer. Le résultat fera l'objet d'un rapport et d'une présentation en classe. Vous devrez aussi porter une attention particulière à la qualité du produit logiciel comme discuté dans les parties théoriques du cours.

Conditions et contexte

Deux équipes seront formées au sein des étudiants du cours. Les deux équipes effectueront le même travail et le résultat final devra être semblable. Il sera par contre possible de répartir le travail de la façon que vous préférez à l'intérieur de l'équipe. Vous serez évalué pour ce projet et une présentation aura lieu à la suite de la première partie et de la deuxième partie. Les tâches à effectuer pour la deuxième partie de ce projet seront présentées ultérieurement.

Pour la première partie, une des deux équipes devra utiliser l'environnement de développement intégré (IDE) Eclipse. L'autre équipe utilisera Eclipse en plus du plugiciel (plug-in) développé dans le cadre de la thèse de doctorat de Guillaume Langelier : Verso. Il s'agit d'un plugiciel permettant la visualisation de la qualité des logiciels à l'aide de représentation 3D, de navigation et de multiples vues présentes dans plusieurs niveaux de granularité. De plus amples informations ainsi qu'une démonstration sera faite à l'équipe qui utilisera l'outil. Lors de la deuxième phase de l'expérience, ce sera au tour de la seconde équipe d'utiliser Verso tandis que la première utilisera l'IDE Eclipse uniquement.

Les deux équipes devront utiliser un système de carte de temps (TimeSheets Next Generation) pour comptabiliser leur temps. Les utilisateurs devront consigner le temps passé à travailler sur le logiciel en ajoutant des commentaires précis sur les activités effectuées, les outils utilisés et les difficultés rencontrées. Ces entrées dans les feuilles de temps sont importantes dans le but de récolter des informations sur l'utilisation des environnements de développement. Par contre, ces entrées ne seront pas utilisées ou prises en compte lors de l'évaluation du projet. Il est donc primordial de les remplir le plus précisément et le plus honnêtement possible. Le logiciel de feuille de temps est très simple, mais une courte formation sera donnée aux deux équipes pour l'utiliser correctement.

Réalisation et contraintes

1. Éditeur graphique de réseaux de Petri.

2. Simulation et animation

   - Création d'un canevas représentant l'espace de travail. L'espace doit être potentiellement infini pour les deux axes.
   - Vous devez créer les objets graphiques représentant les places, les transitions et les arcs. Pour les places, on doit voir leur nom et leur nombre de jetons. Les noms, la priorité et le nombre d'unités de temps sont requis pour les transitions. Pour les arcs, vous devez indiquer le nombre de jetons nécessaires entre une place et une transition pour tirer la transition.
   - Ces items doivent posséder un outil correspondant dans une barre de tâche. Une fois l'outil sélectionné, il est possible de cliquer dans le canevas pour ajouter de nouveaux éléments avec un nom par défaut.
   - L'outil *arc* agit différemment et place le système dans un contexte différent où les places et les transitions peuvent être sélectionnées pour dessiner les liens (pas d'ajout d'arc sans qu'il y ait un départ et une arrivée). Une fois le noeud de départ sélectionné, le système montre seulement les noeuds d'arrivée valides (pas d'arcs place-vers-place ou encore transition-vers-transition).
   - Création d'un outil sélection permettant de déplacer les différents objets à l'aide du glisser-déplacer et de déformer les arcs. L'outil permet aussi de modifier les noms et le nombre de jetons des éléments.
   - Création d'un outil de sauvegarde conservant le modèle interne ainsi que les emplacements exacts des éléments de la représentation graphique.

3. Simulation et animation

   - Vous devez utiliser le modèle créé pour faire l'animation du réseau de Petri.
   - Pour le marquage courant, vous devez être en mesure de déterminer les transitions qui peuvent être tirées.
   - Dans le mode automatique, appliquez la transition avec la plus haute priorité en résolvant les égalités de façon aléatoire. Continuez tant que le réseau

n'est pas bloqué ou encore jusqu'à ce que l'utilisateur arrête manuellement l'animation.

- Dans le mode manuel, effectuez une transition automatiquement quand une seule est possible ou bloquez pour laisser l'utilisateur choisir la transition désirée. L'utilisateur doit pouvoir choisir la transition en cliquant dessus.
- Dans un autre mode, vous devez animer une série de transitions donnée par l'utilisateur avant de commencer l'animation. Votre programme doit indiquer si la série de transitions est invalide au besoin. Ajout de la possibilité de faire marche arrière pour l'utilisateur.
- Enregistrement et possibilité de sauvegarde de la série de transitions effectuée lors d'une séance d'animation. Conserver la durée pour chacun des marquages atteints lors de l'enregistrement.
- L'unité de temps pour les transitions correspondra à des secondes durant l'animation. L'animation doit présenter le ou les jetons voyageant sur les arcs pour passer de son point de départ à son point d'arrivée. La moitié du temps de la transition est consacrée à la section avant la transition alors que l'autre moitié est consacrée à la section après la transition. La représentation de la transition devra être animée pour voir rapidement les transitions en cours d'exécution.
- Des outils de navigation à l'intérieur de l'animation doivent être ajoutés à la barre. Les outils démarrer, arrêter, pause et recommencer doivent être présents.

Annexe IV

Projet (deuxième partie)

Introduction

Le but de cette deuxième partie du projet sera de travailler en équipe pour la modification de l'éditeur et simulateur de réseaux de Petri. Pour ce faire, vous allez utiliser le programme produit dans la première partie du projet. Vous devez terminer et corriger les fonctionnalités existantes et ajouter l'aspect de vérification. Le résultat fera l'objet d'un rapport et d'une présentation en classe. Vous devrez aussi porter une attention particulière à la qualité du produit logiciel comme discuté dans les parties théoriques du cours.

Conditions et contexte

Les deux équipes garderont la même composition. Les deux équipes effectueront le même travail et le résultat final devra être semblable. Il sera par contre possible de répartir le travail de la façon que vous préférez à l'intérieur de l'équipe.

Dans cette deuxième partie, les deux équipes changeront d'environnement (Eclipse et Eclipse+Verso). Les deux équipes continueront d'utiliser le système de carte de temps (TimeSheets Next Generation) pour comptabiliser leur temps.

Réalisation et contraintes

1. Correction des fonctionnalités d'édition et de simulation

2. Graphe de marquage et analyse

   - Définir un marquage initial pour un réseau.

   - Création d'un algorithme permettant de générer le graphe des marquages accessibles (sans tenir compte du temps). Création du graphe de couverture

si nécessaire.

- Analyse des propriétés de blocage, de vivacité et déterminer si le réseau est K-borné. Identifiez le marquage correspondant à la situation d'un blocage ou d'un réseau qui n'est pas K-Borné. Dans le cas où un réseau n'est pas vivant, donnez la liste des transitions mortes. Vous devez utiliser la définition L1-live donnée sur le site suivant : http ://en.wikipedia.org/wiki/Petri_net#Liveness.
- Représentation graphique du graphe des marquages accessibles ou du graphe de couverture dans un canevas à part. Le placement initial des différents marquages devra se faire de façon hiérarchique. Le marquage initial se trouve sur la première rangée et les marquages accessibles depuis le marquage initial se trouvent sur la deuxième rangée et ainsi de suite. Vous devez donc calculer le plus court chemin pour se rendre à un marquage depuis le marquage initial pour déterminer sa rangée. Les marquages sur une rangée donnée sont dans un ordre quelconque. Les marquages peuvent être déplacés par l'utilisateur une fois la version initiale du placement créée.
- Sauvegarde et récupération de ce graphe.

Annexe V

Questionnaire projet IFT6251

Nom : ____ Groupe : ___________________

Encerclez la réponse qui convient à votre choix ou rédigez un court texte pour expliquer votre réponse.

Utilisation générale

1- Selon quelle fréquence avez-vous utilisé le logiciel de contrôle de versions?

   1-jamais 2-un peu 3-souvent 4-toujours

2- Selon quelle fréquence avez-vous rempli les feuilles de temps?

   1-jamais 2-un peu 3-souvent 4-toujours

3- Comment avez-vous utilisé Verso?

   1-jamais 2-pour vérifier le système à la fin 3-pour vérifier après une séance

   4-souvent 5-toujours ouvert

4- Combien avez-vous entré de bogues dans le système?

   ____(approximativement)

5- Avez-vous utilisé la fonctionnalité permettant de créer des versions?

   1-Oui 2-Non

Utilisation des vues

1- Avez-vous utilisé la vue de la qualité?

   1-Oui 2-Non

2- Avez-vous utilisé la vue des logiciels de contrôle de versions?

   1-Oui 2-Non

3- Avez-vous utilisé la vue des bogues?

   1-Oui 2-Non

4- Quelle vue a été le plus la utile pour vous?

   1-qualité 2-contrôle de versions 3-bogues 4-aucune

5- Avez-vous utilisé le niveau de granularité sur les lignes de code?

1-Oui 2-Non

6- Avez-vous utilisé le niveau de granularité des méthodes?

1-Oui 2-Non

7- Avez-vous utilisé le niveau de granularité des classes?

1-Oui 2-Non

8- Avez-vous utilisé le niveau de granularité des paquetages?

1-Oui 2-Non

9- Quel niveau de granularité a été le plus utile pour vous?

1-lignes de code 2-méthodes 3-classes 4-paquetage 5-aucun

10- Avez-vous utilisé les outils de navigation entre les vues?

1-Oui 2-Non

11- Jugez-vous pertinent d'avoir plusieurs vues dans un même outil plutôt qu'avoir accès à plusieurs outils?

1-pas pertinent
2-peu pertinent
3-moyennement pertinent
4-pertinent
5-très pertinent

12- De quelle façon avez-vous principalement utilisé Verso? Décrire un scénario typique.

____

____

____

____

____

____

____

Qualité et performance

1. En temps normal, quel est votre niveau d'intérêt pour la qualité reliée au code et sa maintenance?

   1-pas important
   2-rarement important
   3-parfois important
   4-souvent important
   5-toujours important

2. Pour ce projet, quel a été votre niveau d'intérêt pour la qualité reliée au code et à sa maintenance?
   1-pas important
   2-rarement important
   3-parfois important
   4-souvent important
   5-toujours important

3. De quelle façon Verso a modifié votre niveau d'intérêt pour la qualité reliée au code et à sa maintenance?

   1-diminué mon intérêt
   2-gardé le même intérêt
   3-augmenté mon intérêt

4. De quelle façon Verso a modifié votre performance quand vous développiez le logiciel? Comparez les deux périodes : celle où vous aviez accès à Verso et celle où vous utilisiez seulement *Eclipse*.

   1-diminué
   2-gardé stable
   3-augmenté

5. De quelle façon Verso a modifié la qualité du programme développé? Comparez les deux périodes : celle où vous aviez accès à Verso et celle où vous utilisiez seulement *Eclipse*.

   1-diminué
   2-gardé stable
   3-augmenté

Verso en général

1. Est-ce que selon vous Verso était juste et précis dans sa représentation de la qualité et des informations sur le code?

   1-jamais juste et précis
   2-rarement juste et précis
   3-parfois juste et précis
   4-souvent juste et précis
   5-toujours juste et précis

2. Quelle était votre fonctionnalité favorite dans Verso?

   ___

3. Cotez l'utilité générale de Verso (1-pas utile, 9-très utile)

   1 2 3 4 5 6 7 8 9

4. Après la période de développement et l'apprentissage de Verso complété. Cotez l'utilité générale de Verso (1-pas utile, 9-très utile)

   1 2 3 4 5 6 7 8 9

5. Est-ce que l'état de prototype de Verso a pu nuire à votre expérience?

   1-Oui 2-Non

6. Si une version complète et finale d'un logiciel comme Verso était disponible gratuitement avec votre éditeur de programme favori, est-ce que vous seriez tenté de l'utiliser?

   1-Oui 2-Non

7. Y a-t-il des fonctionnalités manquantes de Verso que vous auriez aimé utiliser?

   ___
   ___
   ___
   ___
   ___
   ___

8. Avez-vous des commentaires ou des suggestions pour l’amélioration de Verso?

____

____

____

____

____

____

____

____

____

____

9. Avez-vous des commentaires généraux en tant que participant à l’expérience?

____

____

____

____

____

____

____

____

____

____