Université de Montréal

Visualisation de la qualité des logiciels de grandes tailles

par
Guillaume Langelier

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

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

décembre, 2006

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Visualisation de la qualité des logiciels de grandes tailles

présenté par:

Guillaume Langelier

a été évalué par un jury composé des personnes suivantes:

Philippe Langlais,	président-rapporteur
Houari Sahraoui,	directeur de recherche
Pierre Poulin,	codirecteur
Stefan Monnier,	membre du jury

Mémoire accepté le: . . . . . .

RÉSUMÉ

Les logiciels sont de plus en plus complexes et les équipes œuvrant sur ces logiciels peuvent être fort imposantes. On a donc besoin de logiciels de qualité pour que la maintenance se fasse facilement. Cependant, la qualité du logiciel dépend d'un ensemble de facteurs encore mal compris. Pour améliorer cette compréhension il faut être en mesure d'analyser plusieurs logiciels dans le but de maîtriser la qualité. Dans ce mémoire, nous proposons une approche d'analyse de logiciels basée sur la visualisation. Notre approche comporte deux parties pour analyser rapidement un grand nombre de codes sources orientés objets. La première partie consiste à représenter des versions de logiciels uniques. Les classes sont affichées comme des boîtes en trois dimensions où les caractéristiques graphiques (couleur, hauteur et rotation) représentent des métriques. Ces classes sont disposées sur un plan selon l'architecture en paquetages en utilisant des algorithmes de placement. Notre étude comparative montre que ces placements sont plus performants qu'un placement naïf et que notre approche est plus efficace que l'inspection manuelle du code. La deuxième partie de notre approche présente l'évolution du logiciel. Nous utilisons l'animation pour représenter la cohérence entre les versions d'un logiciel. Les transitions entre les versions se font à l'aide d'images intermédiaires. Notre approche permet de visualiser plusieurs milliers de classes simultanément à travers plusieurs versions. Elle est basée sur les forces et les faiblesses du système visuel humain. Elle permet aussi la recherche de plusieurs phénomènes associés à la qualité.

Mots clés: visualisation de logiciels, qualité, métrique, détection de phénomènes, évolution du logiciel, animation, expérience.

ABSTRACT

Software applications are becoming more and more complex and teams working on those applications are growing too. We need good quality software in order to facilitate the maintenance process for developers. However, software quality depends on several factors that are still misunderstood. In order to better understand software quality, it is important to analyse several applications quickly. In this M.Sc. thesis, we propose a software analysis approach based on visualization. Our approach consists of two parts to quickly analyse numerous object oriented source codes. The first part explains how we represent software for single versions. Classes are associated to 3D boxes where graphical characteristics (color, height, rotation) represent metrics. Classes are distributed on a plane according to their architectural hierarchies with the use of space-filling algorithms. Our comparative study shows that these algorithms give better results than a naïve layout and that our approach is faster than manual inspection of the code. The second part of our approach studies the representation of software evolution. We are using animation to represent coherence between software versions. The transitions between versions are shown with the help of in-between frames. Our approach allows the visualization of several thousands of classes simultaneously over many versions. It is based on the strengths and weaknesses of the human visual system. It also allows the identification of many quality related phenomena.

Keywords: software visualization, quality, metrics, phenomena detection, software evolution, animation, experiment.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

LISTE DES SIGLES

UML	Unified Modeling Language
RGB	Red Green Blue
XML	Extensible Markup Language
VERSO	Visualisation pour l'Évaluation et la rétro-ingénierie du *SOftware*
GEODES	*Group of Open and Distributed Systems, Empirical Software Engineering*
POM	*Primitive Operators Metrics*
CRSNG	Conseil de Recherches en Sciences Naturelles et en Génie du Canada
FQRNT	Fond Québécois de la Recherche sur la Nature et les Technologies
ACM	*Association for Computing Machinery*
JASE	*Journal of Automated Software Engineering*
CBO	*Coupling Between Objects*
WMC	*Weighted Methods per Class*
LCOM5	*Lack of COhesion in Method Version 5*
DIT	*Depth in Inheritance Tree*
ISO	oraganisation internationale de normalisation
JDK	Java Development Kit
RSS	*Really Simple Syndication* ou *Rich Site Summury*

à mes parents.

REMERCIEMENTS

Les sujets ayant participé à l'expérience méritent des remerciements particuliers pour avoir pris de leur temps bénévolement pour faire avancer ce projet.

Je tiens à remercier toutes les personnes qui ont participé à la réussite du projet VERSO. Des remerciements particuliers vont à Karim Dhambri et Salima Hassaine pour leur apport au projet et pour le temps qu'ils y ont investi. J'aimerais les remercier ainsi que Duc-Loc Huynh pour les idées et les débats qu'ils ont suscités lors de mes recherches.

Je tiens aussi 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 également 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 ce mémoire.

Je remercie également les organismes subventionnaires du CRSNG et du FQRNT pour m'avoir épaulé dans mes études supérieures à différents moments. La subvention du CRSNG au premier cycle m'a permis de m'intéresser à la recherche et d'obtenir des résultats préliminaires alors que le financement du FQRNT m'a permis de poursuivre ces études avec succès.

Finalement, j'aimerais remercier mes parents pour leur support de tous les instants tant au niveau académique que moral. Leurs encouragements, leur support financier ainsi que leurs judicieux conseils ont été des facteurs indispensables à la réussite de cette maîtrise. Je veux aussi remercier mon frère, Simon Langelier, et mes amis, Martin Richard, Sébastien Vézina, Dominique Borduas et Marianne Voyer, pour leurs encouragements soutenus durant ma maîtrise.

CHAPITRE 1

INTRODUCTION

Ce mémoire décrit les recherches qui ont mené à l'élaboration d'une approche de visualisation permettant de représenter efficacement les logiciels de grandes tailles. Nous représentons aussi leur évolution. Cette visualisation tient principalement compte des aspects de qualité du logiciel à travers l'utilisation de métriques. Des expériences viennent aussi valider les observations et la pertinence d'une telle approche.

1.1 Contexte

Le développement de logiciels a beaucoup changé au cours des années. Dans un premier temps, les logiciels deviennent de plus en plus gros et s'adressent à un public de plus en plus large. Ces logiciels demandent donc des ressources sans cesse croissantes tant en termes d'argent, de temps, que de main-d'œuvre. Ces nouvelles réalités incitent les développeurs de logiciels à se soucier un peu plus de la qualité de leurs applications. Un logiciel de qualité est non seulement un logiciel qui offre un service de façon stable, mais est aussi un logiciel qui sera appelé à évoluer facilement et sans engendrer des coûts importants pour ceux qui le maintiennent.

Auparavant, de petites équipes travaillaient sur des logiciels simples répondant à des besoins spécifiques d'une communauté d'initiés. En effet, l'élargissement de la clientèle et l'apparition des ordinateurs personnels amènent les développeurs à changer leur façon de concevoir les problèmes reliés à leurs logiciels. L'important auparavant était d'optimiser la mémoire, la charge sur le processeur et même le nombre d'accès au disque. Aujourd'hui, avec les ordinateurs de plus en plus puissants et surtout les logiciels de plus en plus complexes, cette philosophie d'optimisation des ressources matérielles s'est estompée dans la plupart des situations et la communauté s'est tournée vers une philosophie d'optimisation des ressources financières. Cette optimisation passe principalement par un souhait de réduction du temps de développement et de maintenance. Puisque se-

lon de nombreux auteurs, le temps de maintenance représente entre 60% et 80% [49, 81] de l'effort dans la création de logiciels, des énergies supplémentaires doivent être mises en place pour réduire le temps consacré à cette activité. Chaque avancement dans ce domaine devient alors plus facilement rentable comparativement à des améliorations apportées au processus de développement du logiciel. Il faut aussi empêcher que les modifications apportées n'amènent des défauts supplémentaires à l'intérieur du logiciel.

À la lumière de ces faits, il est évident que la qualité du logiciel peut être évaluée sous plusieurs facettes. En effet, contrairement à la croyance populaire, il ne suffit pas qu'un logiciel ne « bogue » pas pour qu'il soit considéré de bonne qualité. Parmi les autres facettes de la qualité, on trouve d'ailleurs la maintenabilité qui occupe une place très importante. C'est surtout sur cet aspect de la qualité que portera le présent mémoire.

On s'est intéressé très tôt à rendre le code plus lisible, par exemple en élimiant des instructions plus difficilement compréhensibles tel le *goto*, ou en insistant pour que les programmeurs ajoutent des commentaires à leur code et respectent des règles syntaxiques et sémantiques en écrivant les programmes. Plus récemment, il s'est développé une étude plus complète qui tente de mesurer la qualité à l'intérieur des logiciels. Des calculs plus sophistiqués étudient l'intérieur des classes et leurs connexions. C'est ainsi que plusieurs métriques ont été développées pour répondre à différents besoins concernant le logiciel, notamment en éliminant des anomalies. Ces indicateurs doivent servir à repérer des problèmes de haut niveau en terme de qualité.

Un des freins majeurs à l'élaboration de modèles fiables de qualité est l'absence de théorie. Malgré l'intérêt porté à la qualité du logiciel, il n'existe pas de modèles prédictifs appuyés par des lois acceptées par tous. Le domaine en est donc toujours au stade expérimental [19]. Pour être en mesure de progresser vers des modèles plus précis, les chercheurs ont besoin d'analyser une grande quantité de logiciels. En comparant les caractéristiques des logiciels et leurs retombées sur la qualité de ces logiciels, il sera possible de trouver des modèles théoriques plus fiables.

L'avènement et surtout la prolifération de l'Internet, combiné à la grande popularité des logiciels *open source*, ont grandement contribué à rendre une multitude de données facilement accessibles à tous. Ces données peuvent ainsi être analysées et comprises pour

en extraire des règles complexes qui pourront ensuite être validées sur d'autres logiciels. L'interprétation de ces données est primordiale dans un domaine sans cesse en croissance et en changement. Par contre, ces données sont contenues dans des codes sources et donc sous une forme brute. Il faut avoir recours à des outils pour extraire et surtout pour interpréter efficacement ces données. Vue la quantité énorme d'information contenue dans un code source, il peut s'avérer judicieux de faire un tri dans ces données et de se concentrer sur les informations les plus pertinentes de façon à pouvoir les analyser dans un temps fini.

L'étude de l'évolution du logiciel est aussi importante dans un contexte où on ne peut plus se contenter d'analyser les problèmes d'un logiciel. On doit aussi les régler et les éviter lors des prochains projets. En effet, l'observation d'un logiciel à une version donnée peut nous montrer les symptômes de ce qui est arrivé plus tôt sans l'expliquer. D'un autre côté, si plusieurs versions sont visibles à la fois, il est possible de suivre le cheminement qui a mené vers une erreur. On peut aussi constater si une erreur est le fruit d'un ajout instantané ou le résultat de petits manques successifs ayant entraîné une anomalie majeure. De plus, l'évolution permet de travailler autant sur le processus que sur le produit, puisque chaque étape est représentée. Finalement, sans chercher explicitement des anomalies ou des défauts, l'évolution peut aussi nous aider à comprendre le processus de création d'un logiciel qui au-delà de quelques principes généraux, n'est pas documenté du tout. Puisque l'évolution fait exploser le nombre de données à analyser, elle requiert, elle aussi, des outils permettant d'extraire et d'analyser correctement les données provenant de sources brutes trouvées dans l'Internet ou dans les projets logiciels à grand déploiement.

1.2 Problématique

Comme stipulé plus haut, beaucoup d'information est disponible, mais il faut maintenant l'extraire et la représenter de façon efficace. C'est surtout sur la deuxième partie que se concentre le présent mémoire. Il faut savoir doser la quantité d'information présentée à l'utilisateur. Un trop grand nombre de données l'empêche de comprendre certains

principes à cause de la saturation de ses capacités cognitives, mais une trop faible quantité de données empêche toutes conclusions à cause d'éléments manquants. Il faut aussi pouvoir interpréter cette information de façon juste, c'est-à-dire sans interférence des autres informations et sans biais dans la lecture des données. De plus, il faut que ce soit fait efficacement afin de pouvoir explorer le plus grand nombre de données possible et que les résultats soient les plus pertinents possibles. Il faut compresser les données sans perdre les informations importantes à l'étude en cours.

Le défi est encore plus grand en ce qui a trait à l'évolution du logiciel parce que la quantité de données est multipliée par le nombre de versions qui intéressent l'analyste. De plus, il faut aussi représenter de façon adéquate le cheminement chronologique des versions et aider l'utilisateur à identifier les modifications entre ces dernières. En plus de l'addition des données présentes dans toutes les versions d'un logiciel, d'autres informations supplémentaires viennent se greffer dans les transitions. Le caractère itératif du processus de création d'un logiciel fait en sorte que les informations pertinentes peuvent être extraites de ses versions si elles sont considérées dans l'ordre chronologique. Nous verrons plus tard que ceci se traduit dans une cohérence entre les versions subséquentes d'un logiciel.

1.2.1 Problèmes de l'approche manuelle

Une première solution pour l'analyse de ces données est tout simplement de le faire manuellement. Il y a au moins deux façons de le faire. On peut tout d'abord prendre le code lui-même et l'analyser ligne par ligne en le parcourant tout simplement. Cette méthode très exhaustive permet d'analyser le style de programmation en profondeur. Par contre, elle ne fonctionne réellement que pour les systèmes de très petites tailles, car le processus est long et fastidieux [14]. De plus, en observant les lignes de code une par une, on peut passer à côté d'une vision plus globale du système. La présentation d'un programme sous forme de texte réparti dans différents fichiers est aussi très peu efficace et est loin d'exploiter les qualités sensorielles du système perceptuel humain à leur pleine capacité. Le processus est toutefois inévitable pour confirmer des résultats recueillis par voie automatique.

Une deuxième façon de faire l'inspection manuelle consiste en l'analyse par un expert de grandes tables contenant des informations extraites par des outils automatiques. En fonction des indicateurs recueillis, cette technique peut s'avérer plus efficace en ce qui a trait à la vue globale du système. Une panoplie d'information non pertinente est donc éliminée et un plus grand nombre d'entités peuvent être étudiées à la fois. Par contre, de grandes tables remplies de chiffres sont aussi très difficiles à interpréter par l'humain [57]. Le processus demande beaucoup de concentration et de temps. Puisqu'il est facile de perdre le fil entre les différentes entrées dans les tables. Encore une fois, il s'agit de textes et de chiffres qui sont loin d'employer les capacités perceptuelles de l'humain.

1.2.2 Problèmes de l'approche automatique

Une autre façon d'interpréter la grande quantité de données qui proviennent des dépôts de logiciels *open source* est de laisser la machine le faire à notre place. La plupart du temps, il s'agit d'un système de règles ou encore d'un système basé sur l'apprentissage statistique. Les techniques automatiques sont en général très efficaces et peuvent absorber des quantités énormes de données en très peu de temps. Par contre, les systèmes à base de règles doivent être conçus par des experts en fonction de leurs connaissances. Puisque la théorie reste pauvre dans le domaine de la qualité du logiciel et de ses indicateurs, les bornes numériques indispensables aux règles sont ardues à trouver [70]. La qualité d'un logiciel dépend souvent du contexte dans lequel il est pris : son utilité générale, sa taille ou tout autre facteur pouvant influencer un choix de conception. L'ordinateur peut seulement trancher de façon nette dans son analyse du logiciel. Par contre, la recherche de problème demande plutôt un certain flou sur la voie à suivre à chacune des étapes de l'analyse. Cette prise de décision demande aussi une bonne connaissance de l'environnement externe du logiciel et des autres composants avec lesquels il communique.

En ce qui a trait à l'apprentissage automatique, il demande de faire des hypothèses importantes sur les données telle la normalité et le choix des facteurs d'influences, difficilement vérifiable sur les informations extraites d'un logiciel. De plus, cette forme d'in-

telligence requiert d'entraîner les systèmes avec des quantités importantes de données pour s'assurer de leur bon fonctionnement. Or, il faut que ces données soient vérifiées et le processus doit très souvent être fait à la main. Vue la lenteur de ces systèmes à converger, les efforts humains impliqués sont trop grands. Ces systèmes ont aussi de la difficulté à trancher de manière floue.

Un autre désavantage commun aux deux solutions automatiques est l'absence de causes dans les résultats. En effet, la machine donne en sortie des problèmes et des anomalies, et on peut difficilement remonter à la cause de ceux-ci. C'est surtout un problème dans le cas de l'apprentissage. La machine est donc moins en mesure de donner les causes complexes d'un problème et c'est donc plus difficile de le régler. N'ayant aucune connaissance du contexte, la machine peut difficilement donner des conseils sur comment obtenir de meilleurs résultats dans de futurs projets.

1.3 Solution proposée

Puisqu'à la fois la solution complètement manuelle et la solution complètement automatique posent problème, la solution proposée dans ce mémoire utilisera une technique à mi-chemin entre les deux. L'approche est concrétisée dans un projet nommé VERSO pour «Visualisation pour l'Évaluation et la Rétro-ingénierie du *SOftware*». En effet, la visualisation des données logicielles présentées dans ce mémoire se veut une solution semi-automatique mitoyenne entre les algorithmes automatiques trop déterministes et l'approche manuelle trop fastidieuse. À l'aide de la visualisation, les données reçoivent une forme de prétraitement pour à la fois les épurer et leur donner une forme plus aisément compréhensible. Cette forme est donc mieux adaptée au système visuel humain. Elle peut ensuite être présentée à l'écran et étudiée par un expert qui prendra des décisions éclairées par son jugement.

L'approche utilisée et décrite tout au long de ce mémoire est plus précisément la visualisation basée entre autres sur les métriques des logiciels orientés objets. Ces métriques sont transformées en attributs graphiques utilisés lors de la visualisation. Puisque l'objet d'étude reste les programmes orientés objets, chaque élément de la visualisation

représente une classe ou une interface dans un logiciel. Les éléments graphiques sont des boîtes affichées en trois dimensions. Les attributs de ces éléments graphiques sont principalement la hauteur de la boîte, sa couleur et sa rotation. Ces boîtes sont placées sur un plan en deux dimensions en fonction de l'architecture du logiciel. Pour ce faire, une version modifiée de l'algorithme du *Treemap*[35] est utilisée. Ceci permet de bien occuper l'espace. L'approche est interactive, offre aussi des capacités de navigation dans l'univers en trois dimensions et permet à l'utilisateur de choisir complètement librement les métriques qu'il désire associer avec les caractéristiques graphiques. On verra comment il est possible d'utiliser cette approche sur des cas concrets de qualité du logiciel.

En ce qui a trait à l'évolution, au lieu de représenter toutes les versions sur une même image, notre approche utilise l'animation pour faire la transition d'une version à une autre. En utilisant le temps comme analogie pour représenter les versions qui changent, il est possible à la fois d'aider l'utilisateur à comprendre les modifications à l'aide de cette analogie et aussi de réduire énormément l'espace nécessaire durant la visualisation, ce qui permet de disposer plus d'éléments. Il existe plusieurs types de transitions selon la tâche à accomplir. Il est possible de garder les éléments fixes et de voir évoluer leurs caractéristiques ou encore de voir tout le système en mouvement. La cohérence logique entre les versions d'un logiciel est transformée en cohérence visuelle pour que l'utilisateur comprenne mieux les changements qui surviennent au cours du développement ou de la maintenance d'un logiciel. Encore une fois, cette approche est illustrée par des tâches concernant l'évaluation de la qualité d'un logiciel.

1.4 Pièges de la visualisation traditionnelle

Bien que la visualisation soit une solution hybride entre le manuel et l'automatique, ce n'est pas une solution miracle, et il existe quelques pièges à éviter pour obtenir de bons résultats. Dans un premier temps, pour qu'une approche de visualisation soit efficace dans le contexte de l'évalution des classes d'un logiciel, il faut qu'elle soit en mesure de représenter un grand nombre d'éléments. Si seulement quelques éléments à la fois peuvent être affichés ou si l'ajout d'éléments supplémentaires empêche de voir

les précédents, la visualisation n'apportera pas beaucoup d'amélioration par rapport à l'approche manuelle. Le nombre d'éléments visualisés est donc un défi important dans une visualisation adéquate.

Ensuite, le nombre de dimensions montrées par entité est aussi important. Il faut trouver le juste milieu entre présenter un grand nombre de caractéristiques pour chaque élément et la lisibilité de leurs caractéristiques. La prise de décision en rapport aux éléments logiciels requiert souvent de l'information multidimensionnelle et les types de visualisation n'offrant pas cette capacité ou l'offrant par l'intermédiaire de différentes vues sont moins efficaces. Par contre, tout comme quand on ajoute trop d'éléments, l'ajout de trop de caractéristiques peut finir par altérer la perception des diverses caractéristiques. La représentation d'un grand nombre de caractéristiques est donc un deuxième défi pour une visualisation adéquate.

Dans un troisième temps, les représentations classiques du logiciel, surtout par l'entremise de *UML*, ont forcé les chercheurs à interpréter le logiciel comme un graphe. Par contre, sans faire de jeux de mots, les graphes sont difficilement représentables graphiquement quand le nombre de nœuds dépasse la centaine. Ce sont surtout les liens entre les nœuds qui s'entremêlent et deviennent rapidement difficiles à lire. De plus, l'espace occupé par ces liens empêche de placer de nouveaux nœuds à cet endroit. Cette vision du logiciel en tant que graphe a été ancrée si profondément qu'elle a fait dire à Brooks [36] qu'il serait impensable de visualiser le logiciel à cause de sa complexité. Il est donc important d'éviter de succomber à la tentation d'utiliser à nouveau des graphes pour le logiciel et de renouveler la façon dont ses caractéristiques sont visualisées.

Par la suite, comme mentionné plus haut en ce qui a trait à la saturation d'informations causée par le nombre d'éléments ou par le nombre de dimensions représentées, l'ajout aveugle d'informations est à proscrire. En effet, certaines caractéristiques graphiques peuvent nuire à la lecture d'autres caractéristiques ou être tout simplement inadéquates pour être interprétées par l'œil humain. On pense entre autres à la largeur et à la longueur d'une boîte ; à longeur égale les boîtes plus larges paraîtront en général moins grandes car le ratio entre les deux dimensions est plus petit. De plus, certaines textures ou certaines caractéristiques mélangées à l'utilisation de la perspective peuvent

créer des illusions d'optique indésirables à l'intérieur d'une visualisation. On pense entre autres à la différence de luminosité d'une couleur. Elle est difficile à distinguer sans un étalon à portée de la main. Le mélange de couleur peut aussi être contre intuitif pour les non-initiés.

Un dernier piège à éviter dans la construction d'une visualisation adéquate est de vouloir tout montrer ou plutôt de pouvoir tout montrer. En effet, certains outils sont très généraux et sont en mesure de montrer toutes sortes de données. La spécialisation aide à mieux répondre à des problèmes concrets qui sont rencontrés dans des visualisations particulières. Le fait de pouvoir modifier les paramètres d'une visualisation est bien, mais pas au détriment d'une détection efficace de problèmes précis. De plus, l'utilisateur moyen ne veut pas consacrer beaucoup de temps à trouver les réglages optimaux pour un outil générique. Il n'a probablement pas la capacité de le faire non plus vu la complexité du domaine de la perception visuelle humaine.

1.5 Contributions

L'approche de visualisation présentée dans ce mémoire est innovatrice à plusieurs égards. Dans un premier temps, cette façon de visualiser les données logicielles tient compte des forces et des faiblesses du système visuel humain et donc des forces et des faiblesses de l'analyste qui observera les données. La plupart des outils de visualisation de logiciels se contentent de représenter les données de façon naturelle sans observer la théorie du système visuel humain. Notre approche de visualisation est capable d'afficher plusieurs milliers d'éléments à la fois sans qu'il y ait des problèmes de performance ou que l'utilisateur soit saturé par les informations. Le plus grand nombre d'éléments visualisés à l'aide de notre approche tourne autour de 20 000 éléments en analysant l'environnement de développement d'*Eclipse* et plusieurs de ses librairies, ce qui correspond à une taille de deux 2 millions de lignes de code. Il est évident que les capacités de navigation sont nécessaires pour un aussi grand nombre d'éléments, mais une vue d'ensemble du contexte donne toujours une bonne idée de la composition du logiciel même en le visualisant dans son entièreté à une distance raisonnable. Surtout pour les

outils en trois dimensions montrant autant de caractéristiques pour chaque élément, ce type de visualisation se classe parmi celles qui sont capables d'afficher le plus grand nombre d'éléments à la fois.

De plus, la visualisation est une approche mixte qui rallie les qualités des approches manuelle et automatique, mais surtout qui réduit leurs défauts. Ce type d'analyse de la qualité du logiciel remet l'expert au centre de l'étude du logiciel. En effet, il est plus apte que la machine à interpréter ce qu'il voit, car il a conscience du contexte et du logiciel en entier. Il peut aussi utiliser des expériences d'analyses antérieures pour mieux comprendre le logiciel en cours de visualisation et pour reconnaître certains patrons récurrents. Ces patrons peuvent être des règles théoriques et stables insoupçonnées sur la conception du logiciel. Ce qui amène à l'autre avantage de la non-automatisation de l'approche, c'est-à-dire la capacité de l'approche à laisser l'expert explorer le logiciel sans qu'il ait de buts précis. Il peut ainsi s'attarder sur des éléments captant son attention à l'extérieur des balises préfixées par l'analyse. Ce genre d'exploration est très difficile, voire impossible en utilisant des algorithmes automatiques.

Pour finir, notre approche de visualisation est axée sur la détection de phénomènes liés à la qualité du logiciel. Ceci la démarque de plusieurs approches qui se concentrent sur la cueillette d'informations techniques sur les logiciels pour prendre des décisions d'un autre ordre. De plus, notre approche a montré qu'elle était en mesure d'aider à découvrir facilement plusieurs anomalies et phénomènes relatifs à la qualité du logiciel. Le fait qu'elle puisse être utilisée tant par les chercheurs que par les développeurs de logiciels en fait un outil puissant. Son efficacité a aussi été démontrée sur plusieurs logiciels réels de plusieurs milliers de classes et ne se limite pas à des cas jouets ou à des cas bien précis.

Les différents travaux rapportés dans ce mémoire ont donné lieu à un certain nombre de publications dans différents forums [40–44] en plus d'avoir été l'objet de diverses démonstrations. La publication à la conférence *Automated Software Engineering*[41] s'est vue décerner le prix de l'article s'étant distingué (*ACM Distinguished Paper Award*) et nous a valu une invitation à publier dans le journal JASE(*Journal of Automated Software Engineering*). De plus, des résultats obtenus récemment font l'objet de projets de

publications dans des forums de haut niveau.

1.6 Survol du mémoire

La suite de ce mémoire décrit en détail l'approche de visualisation découlant des recherches et est divisée comme suit. Le chapitre 2 est une mise en situation présentant à la fois de l'information sur le système visuel humain et sur l'aspect quantitatif des logiciels. Ce chapitre se termine par une discussion sur les travaux précédents concernant la visualisation de logiciels. Le chapitre 3 présente en détail comment le logiciel et les métriques sont visualisés. On peut aussi y voir quelques applications possibles pour l'approche développée. Le chapitre 4 donne les résultats d'une expérience menée sur l'approche de visualisation. Elle se concentre principalement sur l'efficacité des différents algorithmes de placement. Le chapitre 5 reprend le même schéma que le chapitre concernant la visualisation d'une version unique et présente la visualisation de l'évolution. Finalement, le chapitre 6 est une conclusion récapitulant les idées principales du mémoire et donnant une vision sur les travaux qui auront lieu suite à cette recherche.

CHAPITRE 2

CONTEXTE ET TRAVAUX CONNEXES

Avant d'en arriver à l'exposé des techniques utilisées pour la représentation des logiciels, il est intéressant de comprendre sur quoi est construite la visualisation. Pour obtenir une bonne visualisation il faut tout d'abord avoir quelque chose à représenter. Puisque le point d'intérêt de l'équipe de VERSO se situe au niveau de la qualité du logiciel, les métriques sont un choix intéressant. La section 2.1 explique ce qu'est une métrique et comment elles servent en contrôle de la qualité. La section explique aussi comment on peut décupler l'impact des métriques en utilisant la visualisation. Ensuite, pour qu'une visualisation soit efficace, elle requiert une compréhension approfondie du système visuel humain. La section 2.2 discute de certains enjeux sur l'évolution du logiciel. La section 2.3 présente les forces, les faiblesses et les pièges à éviter lors d'une telle interaction avec le système visuel humain. Finalement, la section 2.4 discute les travaux précédents en énumérant leurs caractéristiques et donne une critique en les comparant avec notre approche présentée dans ce mémoire.

2.1 Aspects quantitatifs du logiciel

Les codes des programmes sont complexes et ont une sémantique comprise par les artisans du logiciels seulement. Les machines sont capables de les interpréter, mais elles ne font que traduire les instructions en calculs sans comprendre ce qu'elles font. Une preuve de ceci est que lors de la rétro-ingénierie, il n'est pas facile d'extraire les informations permettant de déterminer la fonction du logiciel et son contexte. Devant cette abstaction, les chercheurs ont décidé d'utiliser des métriques. Les métriques sont des indicateurs pour les caractéristiques de qualité définies à un niveau supérieur.

Les métriques sont en fait des mesures [20] qu'on prend sur le code ou d'autres artefacts résultant du processus de développement du logiciel. Ce type de mesure est semblable à celles qui peuvent être prises sur des objets physiques. Au lieu de prendre

les mesures à l'aide d'un appareil comme une règle ou un thermomètre, elles sont prises en comptant différents éléments du logiciel. Des opérateurs arithmétiques ainsi que des facteurs d'atténuation peuvent aussi être utilisés sur ces décomptes. Quelques exemples de mesures sont le nombre de lignes de code, d'attributs ou encore de méthodes dans une classe. On peut aussi mesurer des informations sur les arguments des méthodes ou encore sur les appels internes ou externes d'une classe.

Il existe deux grands types de métriques : les métriques statiques et les métriques dynamiques [34]. Les premières sont extraites du code source d'un programme ou de son exécutable, alors que les deuxièmes requièrent que le programme s'exécute pour être comptabilisées. Les deux ont des avantages et des inconvénients. Les métriques dynamiques ne sont valables que pour une seule exécution, alors que les métriques statiques sont valables pour toutes les exécutions. Par contre, les métriques statiques ne tiennent pas compte des valeurs décidées dynamiquement dans le programme alors que les métriques dynamiques représentent le contexte d'exécution typique d'un programme. Selon Jackson et Rinard [34], le futur des métriques se trouve plutôt du côté statique. C'est d'ailleurs sur ce type particulier que se concentreront les techniques de visualisation décrites dans les prochains chapitres. Veuillez vous référer à l'annexe III pour des exemples de métriques statiques prises sur le code.

Les métriques ont un gros avantage sur le code représenté par du texte. Ce sont des nombres significatifs sur lesquels les opérations sont faciles à effectuer. À titre d'exemple, il est facile de prendre une série de métriques et d'en extraire la moyenne, le mode et la médiane. Il est possible de prendre ces nombres et de les modifier à l'aide de transformations plus ou moins complexes sans perdre ou altérer les informations contenues dans les valeurs. Ceci est un sérieux avantage puisque la visualisation oblige déjà la transformation des données vers une forme graphique.

L'extraction de ces métriques demande une infrastructure importante et n'est pas aussi simple qu'il n'y paraît. Il faut tout d'abord créer une représentation manipulable du programme et appliquer des algorithmes sur les graphes générés dans ce modèle. Pour extraire les métriques utilisées dans le présent mémoire, nous avons utilisé un outil appelé *POM* [26] développé par des membres de l'équipe du *GEODES* de l'Université

de Montréal. Cet outil permet de calculer des métriques sur de très grands logiciels dans des temps raisonnables (environ 1 heure pour un logiciel de 5000 classes pour une machine standard au moment de l'écriture de ce mémoire). Les métriques générées par cet outil sont ensuite écrites dans un fichier XML contenant chaque classe et ses attributs accompagnés d'informations structurelles telles les relations d'héritage ou les relations d'association également extraites par un outil interne [27]. Ce fichier peut être facilement transformé si les besoins de la visualisation changent. Il est aussi possible d'utiliser des données provenant d'autres sources en les traduisant vers le format XML accepté par notre prototype de visualisation présenté dans les chapitres 3 et 5.

2.1.1 Métrique en tant qu'indicateur de qualité

Les métriques sont le reflet des besoins des utilisateurs et des chercheurs dans leur tâche d'évaluation de la qualité d'un logiciel. Ceci leur confère un avantage certain par rapport aux autres techniques, parce qu'elles sont spécialisées sur la qualité du logiciel. Par exemple, les métriques sur la taille partent du principe qu'il est préférable d'avoir des classes de petites tailles dans les logiciels, quitte à en avoir plus. Plus les éléments individuels sont petits (tout en étant complets), plus il est facile en général de les comprendre et de les modifier pour les futurs programmeurs [10]. De la même façon, les mesures de couplage testent le potentiel de répercution d'une erreur ou de changements pouvant intervenir sur une portion de code. Si une classe est couplée à plusieurs autres, elle peut potentiellement influencer toutes les classes avec laquelle elle est couplée. Cette classe nécessitera donc plus de tests de façon à s'assurer que les fonctionnalités du reste du logiciel n'ont pas été brisées quand des changements surviennent dans cette classe. C'est pourquoi un faible couplage donne un bon indice de la qualité d'un logiciel [10].

La cohésion est aussi une qualité recherchée par les développeurs soucieux de permettre une maintenance plus facile de leurs logiciels. Tout comme une petite taille aide à cerner ce qui est pertinent à l'intérieur d'une classe, un composant logiciel faisant une tâche unique et précise aide à retrouver les endroits où le code doit être modifié pour amener les changements requis. La cohésion étant la tendance d'une classe à n'effectuer qu'une seule fonction, elle est relativement abstraite et est donc plus difficile à décrire

sous la forme d'une formule mathématique. Les chercheurs l'estiment de différentes manières avec un succès relatif jusqu'à présent [6]. Pour ce qui est des métriques de structures telles la profondeur dans l'arbre d'héritage, le nombre d'enfants, le nombre d'ancêtres et ainsi de suite, elles sont très pertinentes pour déterminer si une classe peut être facilement réutilisée ou non. Si une classe possède plusieurs enfants, elle a tendance à être plus abstraite. Par contre, un arbre d'héritage très plat témoigne d'un manquement au niveau de la réutilisation disponible à travers l'héritage et aussi d'une sous-utilisation des outils des langages orientés objets, tel le polymorphisme [50]. Toutes ces métriques apparaissent comme de bons indicateurs de la qualité d'un logiciel.

2.1.2 Choix des métriques

VERSO exploite toutes les métriques décrites dans l'annexe III et plusieurs autres. Il suffit que ces métriques soient transformables sous forme de nombres réels. Les métriques les plus utilisées sont présentées au tableau 2.1. CBO (*Coupling Between Objects*) représente le couplage en comptant le nombre de classes différentes rattachées à une classe précise autant en entrée qu'en sortie. WMC (*Weigthed Methods per Class*) est une mesure à la fois de taille et de complexité puisqu'elle additionne le nombre de méthodes d'une classe pondéré par un facteur de complexité quelconque. Dans notre cas, ce facteur est le nombre de méthodes invoquées à l'intérieur d'une méthode. LCOM5 (*Lack of Cohesion in Methods Version 5*) représente le manque de cohésion dans une classe. Son calcul est fait à partir des paires de méthodes agissant sur les mêmes attributs versus celles agissant sur des attributs différents. Une grande valeur pour cette métrique représente un manque de cohésion dans la classe. Finalement DIT (*Depth in Inheritance Tree*) sert de mesure structurelle sur la hiérarchie des classes du système. Pour une classe donnée, DIT représente la longueur du chemin pour se rentre de la racine à cette classe en passant par les liens d'héritage.

Deux raisons principales justifient le choix d'utiliser fréquemment ces métriques. La première est que chaque métrique est le représentant le plus commun de chacune de quatre catégories, c'est-à-dire la taille, le couplage, la cohésion et les métriques de structure. La deuxième raison est qu'elles ont tendance à bien représenter les attributs de

Acronyme	Nom	Catégorie
CBO	*Coupling Between Objetcs*	Couplage
DIT	*Depth in Inheritance Tree*	Structure
LCOM 5	*Lack of Cohesion in Methods*	Cohésion
WMC	*Weighted Method per class*	Taille

Tableau 2.1 – Métriques principalement utilisées avec notre approche.

qualité reliés à la maintenance du logiciel. Leur pertinence a été reconnue dans diverses recherches (voir section 2.1.4). Ceci dit, l'utilisateur de VERSO est invité à expérimenter avec les métriques de son choix.

2.1.3 Correspondance entre métriques et code

Les métriques sont aussi utilisées parce qu'elles représentent bien le code. C'est-à-dire que les métriques vont suivre les changements apportés au code. Par exemple, Géhéneuc *et al.* [26] utilisent des vecteurs de métriques pour retrouver des patrons de conception à l'intérieur du code sans le consulter en tant que tel. De la même façon, il est pertinent d'utiliser un vecteur de métriques pour retrouver des classes qui ne correspondent pas à certains critères de qualité. Si le vecteur de métriques a plusieurs dimensions, il est même possible de différentier chacune des classes en étudiant seulement ce vecteur.

Le fait de transformer du code ayant une sémantique complexe et une structure difficilement interprétable par l'ordinateur en un simple vecteur de nombres rend l'utilisation des métriques attrayante. En effet, la visualisation demande des données simples pour pouvoir les transformer en un nombre raisonnable de caractéristiques graphiques.

2.1.4 Quelques travaux sur les métriques

Il existe des études qui prouvent que les métriques sont efficaces et de bons indicateurs de certains axes de la qualité d'un logiciel. Basili *et al.* [3] montrent qu'il y a une corrélation entre les métriques introduites par Chidamber et Kemerer [10], et le nombre de fautes trouvées dans les classes. Toutes les métriques ont donné des résultats

significatifs sauf pour la métrique LCOM (*Lack of COhesion in Method*) qui a été améliorée depuis le temps. Les hypothèses évidentes ont toutes été confirmées. Par exemple, un grand couplage d'une classe a tendance à augmenter son nombre de fautes et il en est de même pour une faible cohésion. Dans un deuxième temps, Li et Henri [48] ont aussi démontré que la même suite de métriques forment de bons indicateurs de l'effort de maintenance déployés sur une classe. Les conclusions vont dans le même sens que la recherche avec le nombre de fautes par Basili *et al.* [3]. Mao *et al.* [53] ont utilisé différentes métriques à l'intérieur d'un système d'apprentissage machine pour prédire la réutilisabilité d'un logiciel. Il s'avère que les métriques, combinées à certains seuils prédisent efficacement le caractère réutilisable d'une classe. Aussi, Grosser *et al.* [25] montrent qu'il est possible de prédire la stabilité à l'aide de différentes métriques logicielles. Ils utilisent cette fois le raisonnement basé sur les cas pour trouver la façon d'utiliser ces métriques. Finalement Briand et Wüst [7] font une revue des mesures utilisées dans le logiciel orienté objet. Leur revue comprend aussi des validations et les métriques pouvant servir à prédire les différents indicateurs de qualité.

2.2 Évolution du logiciel

L'évolution du logiciel est devenue de plus en plus importante depuis quelques années. Bien qu'on s'intéresse depuis longtemps au processus de création du logiciel à travers différentes normes et procédés tels la norme ISO 9126, il n'en reste pas moins que l'évolution du logiciel en tant que produit est une préoccupation relativement nouvelle. Il s'agit d'une optique très différente puisque les normes s'intéressent plus aux façons de faire les choses et à encadrer le milieu dans lequel les logiciels sont construits. Par contre, inspecter l'évolution de la qualité du produit logiciel aide à mieux comprendre et prédire les coûts rattachés aux erreurs de conception après qu'elles aient été faites. L'intérêt a tout d'abord commencé par la qualité et les anomalies présentes dans un logiciel sur des versions fixes. Par la suite, il a été réalisé qu'il est pertinent d'évaluer la qualité des versions précédentes pour mieux comprendre les problèmes rencontrés dans le cheminement du logiciel.

Dans un premier temps, l'évolution du logiciel permet de dater l'introduction d'un problème ou n'importe quel événement à l'intérieur de l'évolution d'un logiciel. De cette façon, il est possible d'identifier le développeur associé à un phénomène observé dans la version finale. Par contre, si c'est la seule raison de regarder l'évolution, ça n'apporte que peu d'informations pour la recherche. Avec la datation d'un phénomène, il est aussi possible de faire des corrélations entre son apparition et l'âge du logiciel, le contexte du logiciel lors de son implantation, d'autres événements ayant été introduits simultanément et le contexte entourant l'équipe à ce moment.

Ensuite, on peut voir comment un problème est introduit dans le logiciel. Ainsi, il est possible de distinguer entre une introduction lente et progressive d'un phénomène ou son apparition instantanée. Les deux types d'événements ont des causes différentes et donc des solutions différentes. Par exemple, le premier cas n'est pas apparent dès son introduction et ne constitue d'ailleurs pas une situation particulière lors de son implantation. C'est plutôt le reste du logiciel qui influence la stratégie adoptée pour qu'elle dégénère en quelque chose d'inadéquat. Il s'agit donc d'une erreur dans l'évaluation du développement à long terme d'un logiciel. La solution à adopter dans ce cas est probablement de revoir la stratégie initiale dans le cadre d'un réusinage important. Il faut aussi tirer des leçons d'une telle situation parce que ce sont des phénomènes difficiles à éviter. Ils se forment graduellement et demandent plus de travail pour la correction une fois diagnostiqués.

Le deuxième type d'événements est souvent causé par une seule modification du logiciel, donc souvent dû à un seul auteur. Le problème est moins sournois cette fois-ci et peut être diagnostiqué dans toutes les versions subséquentes à son implantation. Pour la correction, il s'agit seulement de modifier le code comme si on avait choisi une meilleure solution au moment de l'implanter.

Un dernier point utile à la représentation du logiciel est la cohérence entre les versions d'un logiciel. Par cohérence logicielle, on entend que les versions consécutives d'un logiciel ont tendance à se ressembler beaucoup plus que des versions qui sont éloignées dans le temps ou encore les versions d'un autre logiciel. Ce principe est similaire aux animations en infographie, les images successives d'une séquence vidéo ont ten-

dance à se ressembler plus que les images éloignées dans le temps. La plupart des algorithmes de compression vidéo sont d'ailleurs basés sur ce principe. Ceci paraît évident, mais il sera montré plus tard que ce ne sont pas tous les outils de visualisation montrant l'évolution qui exploitent cette facette du logiciel. Cette cohérence est due au fait que les versions sont toujours bâties en partant de versions déjà existantes, sauf peut-être pour la toute première. Les développeurs font toujours des modifications basées sur l'état de la version précédente. On le fait soit pour corriger une anomalie dans la version précédente, soit pour lui ajouter une fonctionnalité. Il en résulte que la plupart des changements entre les versions[1] sont petits et ciblés facilement.

2.2.1 Quelques travaux sur l'évolution du logiciel

D'importantes recherches sur l'évolution du logiciel ont été effectées par Lehman *et al.* [47]. Lors de leurs recherches où ils ont analysé l'évolution des logiciels, ils ont été en mesure de faire ressortir plusieurs lois qui s'appliquent de façon générale. Pour résumer en quelques mots ces lois, le logiciel doit être en constante évolution et les efforts pour le maintenir doivent être soutenus pour le garder en vie. Les huit lois sont les suivantes : *Continuing Change, Increasing Complexity, Self Regulation, Conservation of Organisational Stability, Conservation of Familiarity, Continuing Growth, Declining Quality, Feedback System*. La publication de ces différentes lois s'échelonne sur trois décennies, des années 70 aux années 90. Malheureusement, beaucoup de ces lois sont plutôt reliées au processus logiciel qu'au produit logiciel. Dans un article de Ramil [65], on recherche des métriques intéressantes pour qualifier l'évolution du logiciel. Ces métriques servent principalement à compter le nombre d'éléments ajoutés. Pour leur part, Kemerer et Slaughter [37] décrivent une approche sur la façon de faire des études empiriques sur l'évolution du logiciel, malgré les défis que cela représente. Ils ne font que présenter une méthode rigoureuse, mais ne tirent pas de conclusions ou de lois précises sur l'évolution du logiciel.

Dans ses recherches, Capiluppi [9] confirme certaines des lois de Lehman *et al.* en

[1] Par «version» on entend quelques validations (*commits*) tout au plus et non pas les versions livrées au grand public.

étudiant des programmes *open source*. Il semble alors que le nombre d'intervenants dans un programme augmente à mesure que le programme grossit. Il a aussi découvert que bien que la taille des modules change beaucoup entre les différents programmes étudiés, elles semblent tous converger vers une valeur stable à un moment ou à un autre. Même s'il semble exister très peu de recherches qui s'intéressent directement à l'étude de l'évolution, elle est toujours indirectement évaluée à l'aide des caractéristiques de qualité. En effet, la plupart des travaux sur les métriques et la qualité sont directement liés à la facilité de maintenance en tant que caractéristique de qualité externe. Le phénomène en lui-même reste par contre toujours mal compris et peu de gens se sont intéressés à trouver des métriques particulières reliées à l'évolution. Plusieurs travaux donnant des outils pour mieux comprendre l'évolution sont présentés dans l'état de l'art de la visualisation de l'évolution à la section 2.4.3.

2.3 Système visuel humain

L'humain a une grande capacité d'analyse lui permettant de donner une sémantique au monde qui l'entoure. L'humain acquiert les informations lui servant à résoudre les problèmes à travers son système sensoriel dont fait partie entre autres le système visuel. Ce système s'adapte très facilement à toutes sortes d'informations de différentes natures qui peuvent être floues, imprécises, incomplètes et même totalement inédites. Cette polyvalence est intéressante, mais empêche la spécialisation dans certaines sphères telle la lecture de données sous forme de tableau où l'ordinateur, lui par contre, excelle. Il est donc primordial de comprendre le système visuel humain pour être en mesure de lui fournir l'information de la façon qui lui convient le mieux. Une approche de visualisation, en plus de montrer l'information de façon pertinente et juste, doit aussi montrer l'information pour qu'elle soit le plus facilement interprétable possible.

Le système visuel humain comporte des forces et des faiblesses et il est important de les prendre en considération dans un système de visualisation. Cette section fait donc la revue des caractéristiques de du système visuel humain.

2.3.1 Forces

Une des principales forces du système visuel humain réside dans le haut taux de parallélisme du cerveau. En effet, l'humain traite l'information contenue dans les images de façon simultanée et très rapide. Ceci lui donne l'occasion d'avoir une vision générale d'une scène et ainsi d'éliminer les parties inintéressantes et de mettre l'accent sur ce qui est important. Les éléments identifiés peuvent ensuite être inspectés plus en profondeur par l'observateur.

Certaines caractéristiques sont du domaine de la perception instantanée [2]. Un tel phénomène est perçu par l'observateur extrêmement rapidement (environ 0,3 seconde, ce qui correspond au temps nécessaire pour que le système nerveux avertisse le cerveau). Ce type de la perception relève donc du réflexe et doit être exploité le plus possible durant la visualisation. Par contre, quand le stimulus visuel requiert que l'observateur parcourt tous les éléments un à la suite de l'autre, même si c'est fait rapidement, on perd les avantages de la perception instantanée et le temps de recherche devient ensuite dépendant du nombre d'éléments présentés à l'image. Comme on le voit dans la figure 2.1, la couleur peut être perçue très rapidement alors que la forme est beaucoup plus difficile à percevoir quand elle est influencée par la couleur.

De plus, le cerveau humain a de la facilité à compléter l'information sur la profondeur, même quand elle est simulée sur un support en deux dimensions. Le cerveau fonctionne aussi beaucoup par association d'images connues. Il reconnaît ainsi instantanément certains patrons. Ceci est très efficace pour retrouver des occurrences de phénomènes récurrents, mais aussi pour retrouver des éléments et qui apparaissent anormaux.

2.3.2 Faiblesses

Comme mentionné plus haut, le système visuel humain est très polyvalent, mais manque de spécialisation dans certains domaines. Dans un premier temps, si l'œil doit parcourir tous les éléments d'une liste, il sera lent à accomplir cette tâche puisqu'il doit s'attarder un minimum de temps sur chaque élément en laissant le temps à l'information

[2] L'expression perception instantanée est une traduction libre de *preattemptive perception* mentionnée par Healey et Enns [31].

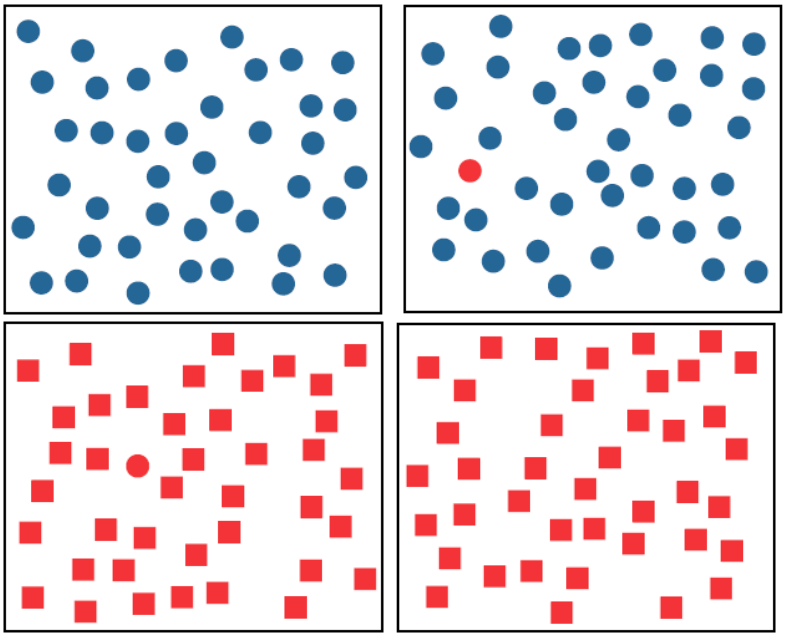


Figure 2.1 – Exemple de perception instantanée. Dans cet exemple [31], la couleur présentée dans les images du haut est observable instantanément alors que la forme est plus difficle à détecter dans les images du bas.

de voyager dans le système nerveux. Le système visuel humain est donc beaucoup plus lent pour ce genre de tâche que l'ordinateur qui peut parcourir des tableaux de données à une vitesse phénoménale.

Dans un deuxième temps, le système visuel humain est facilement influencé par les biais. On sait que le cerveau fonctionne à l'aide d'associations d'images qu'il a déjà vues. Ce phénomène peut lui jouer des tours, car certaines images sont analysées trop rapidement parce qu'elle sont associées à quelque chose de connu. Par contre, l'image perçue n'est pas toujours conforme à la réalité. Ceci introduit des biais dans la perception qui se traduisent par une mauvaise interprétation des données. Par exemple, les rectangles plus larges nous paraissent toujours un peu plus courts parce que le système visuel humain interprète rapidement le ratio entre les côtés de cette forme connue. Pour les mêmes raisons, une couleur paraît différente selon la couleur du fond sur lequelle elle est représentée. Résultant d'une autre ambiguïté, les représentations en trois dimensions portent à confusion puisqu'il faut que l'observateur comprenne si la position de l'objet

est plus éloignée ou s'il est tout simplement plus petit. Pour empêcher la confusion et assurer une lecture des données qui est juste et rapide, il faut réduire au maximum les biais présents dans une visualisation. La figure 2.2 montre des exemples de biais.

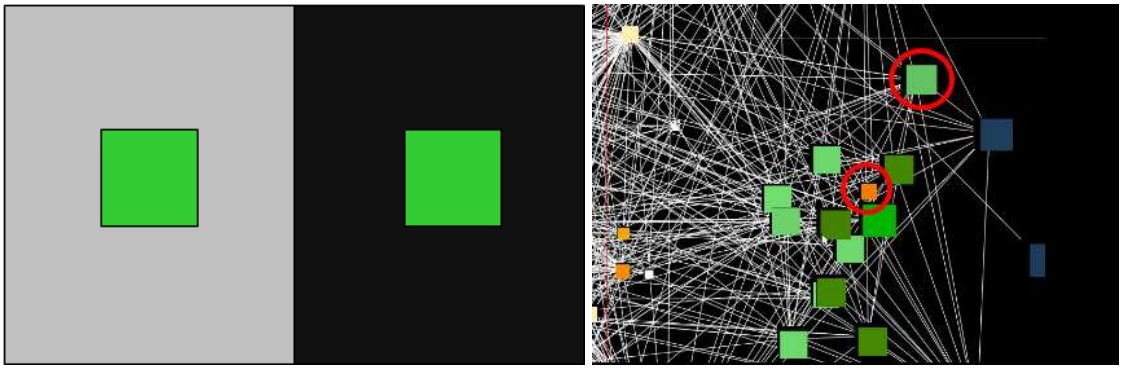


Figure 2.2 – Exemples de biais possibles. L'image de gauche montre comment l'œil est influencé par la couleur. Le carré vert situé sur la gauche semble plus foncé que celui de droite à cause du fond plus pâle. En réalité, les deux carrés verts ont exactement la même teinte. Dans l'image de droite [69], on voit comment la pronfondeur peut influencer la perception de la grosseur d'un objet. Sans indice visuel, il est impossible de savoir si le cube est éloigné ou petit. Dans la réalité tous les cubes ont la même grosseur dans cette représentation.

Pour finir, le système visuel humain est aussi facilement saturé. La surabondance d'information, si elle est mal classée ou moindrement complexe, est fatale à l'interprétation des données. Curieusement, à partir d'une certaine limite, plus il y a d'informations de présentées, moins il y aura d'informations absorbées par l'observateur. De la même façon, des représentations se chevauchant amènent parfois une certaine portion d'information à être incompréhensible parce que cachée. C'est malheureusement le cas des données représentées sous forme de graphes où les liens s'entremêlent pour devenir un *spaghetti*. Il est alors impossible de retrouver l'origine et la destination de chaque lien. C'est aussi le cas pour les représentations de type nuage de points qui, selon la corrélation des variables, ont une fâcheuse tendance à empiler les éléments les uns sur les autres. La figure 2.3 montre un exemple où les données deviennent difficiles à interpréter à cause d'une trop grande quantité d'informations organisées de façon maladroite.

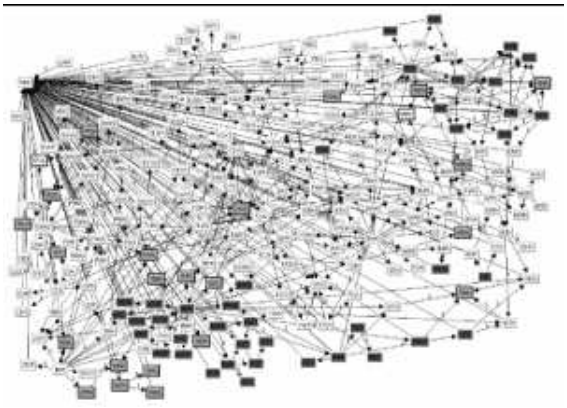


Figure 2.3 – Exemple de saturation [38]. Cette image montre que trop d'information peut rendre la lecture des données difficile et ainsi rendre la visualisation inefficace. Les représentations sous forme de graphes sont particulièrement pernicieuses à cause du grand nombre d'arcs qui se croisent et se superposent innévitablement.

2.3.3 Principes de la *Gestalt*

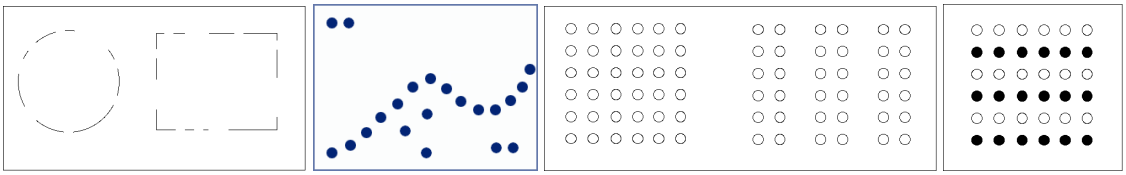
Les principes de la *Gestalt* consistent en une série d'énoncés en psychologie, dont une partie est directement liée à la perception. Entre autres, les lois les plus intéressantes et les plus simples de cette théorie sont les lois de la *Prägnanz*. Selon ces lois, l'humain groupe les éléments qui selon lui forme un tout. Ces groupements sont efficaces pour catégoriser les éléments et pour les différentier lors de la visualisation d'information.

Les lois de la *Prägnanz* sont les suivantes :

- la loi de fermeture groupe les objets composant des formes.
- la loi de similarité groupe les objets ayant les mêmes caractéristiques (couleur, forme, intensité, etc).
- la loi de proximité groupe les éléments étant à une plus courte distance.
- la loi de la symétrie groupe les éléments symétriques.
- la loi de la continuité groupent les éléments dessinant un patron quelconque.
- la loi du sens commun groupe les éléments se déplaçant dans une même direction.

La figure 2.4 illustre certaines de ces lois.

Dans ces lois, les plus importantes concernant notre approche de visualisation sont les lois de similarité, de proximité et du sens commun. Dans un premier temps, le re-



(A) (B) (C) (D)

Figure 2.4 – Principes de la *Gestalt*. A) La fermeture : le cerveau complète les lignes pour reconnaître les formes. B) La continuité : les objets d'une même forme sont groupés. C) La proximité : les objets près les uns des autres sont groupés. D) La similarité : Les objets avec des caractéristiques similiaires sont groupés.

groupement des éléments similaires est important pour retrouver les éléments ayant les mêmes caractéristiques visuelles. Dans le cas des classes, ces caractéristiques visuelles correspondent à des métriques et donc à des caractéristiques de qualité de la classe. Le groupement des classes ayant les mêmes caractéristiques est important pour observer un même comportement sur des classes similaires. Dans un deuxième temps, il faut, dans la mesure du possible, donner une sémantique à la proximité pour ne pas que l'utilisateur soit confus. Dans VERSO, nous avons développé un amalgame entre le principe de fermeture et le principe de proximité. En fait, les différentes entités représentant l'architecture du logiciel sont encadrées, puis positionnées à proximité en fonction de leur lien de parenté avec les autres entités. La loi du sens commun est utilisée pour l'évolution où les items sont appelés à bouger. On peut repérer facilement les objets qui ont été modifiés de ceux qui n'ont pas été modifiés car les uns se transforment et les autres non.

2.3.4 Perception de la cohérence

En animation, la cohérence entre les images successives est primordiale pour bien comprendre les changements d'une image à une autre. En effet, garder une continuité entre les images aide l'observateur à mieux se situer et à se concentrer sur les changements. Le logiciel a une certaine part de cohérence intrinsèque. Les changements d'une version à l'autre sont souvent ciblés et en petits nombres (voir section 2.2). Il faut par contre traduire cette cohérence logicielle en cohérence visuelle.

Comme dans le cas d'une séquence vidéo, il faut s'assurer que les images changent

peu d'une à l'autre pour que le système visuel de l'observateur sente la continuité qui existe entre elles. Dans la visualisation d'information, la problématique n'est plus esthétique mais bien cognitive. Si seulement quelques objets bougent lors d'un changement d'image et que le reste est présenté de la même façon, l'observateur aura l'impression qu'il regarde la même image où seulement quelques éléments ont été effacés et redessinés. L'œil est automatiquement attiré vers ces changements. L'observateur peut alors se concentrer très facilement sur ces éléments et oublier le reste, car c'est déjà imprimé dans sa mémoire visuelle.

Une telle cohérence est facilement atteignable à l'aide de l'animation. Par contre, si des images sont placées côte à côte, on perd cette notion de cohérence et de continuité. L'observateur est contraint de déplacer son œil d'une image à l'autre pour retrouver les différences. De plus, si l'animation est utilisée, mais que l'image est complètement différente quand elle change, la cohérence n'est pas conservée non plus. À ce moment, l'observateur devra faire un effort cognitif supplémentaire pour retrouver tous les éléments qui se trouvaient dans la scène de départ dans le but de trouver des points de repère. Ensuite, il pourra s'attarder à chercher les modifications. Elles ne seront pas soulignées par l'effet du changement d'images à ce moment et beaucoup plus difficile à trouver. Nous verrons à la section 5.3 comment ce principe est intégré à VERSO.

2.3.5 Quelques travaux sur la perception

Les travaux ayant influencés le plus les recherches décrites dans le présent mémoire sont ceux de Healey et Enns [31]. Healey a entre autres fait des travaux sur les couleurs les plus efficaces en terme de perception [28]. La couleur exprimée dans le format traditionnel (RGB) ne forme pas un espace uniforme en terme de perception par l'être humain. Il propose donc une technique pour discrétiser un certain nombre de couleurs pour qu'elles soient efficacement différenciées par un observateur. Dans un autre article, Healey *et al.* discutent de différents attributs graphiques et de la facilité des détecter par l'utilisateur [29]. C'est dans cet article qu'il introduit la notion de perception instantanée et qu'il discute de l'influence de certains attributs sur d'autres. À l'aide de collègues différents, il a mis ces principes en application dans une visualisation scientifique en trois

dimensions [30] et a plus récemment discuté du mouvement et de sa perception [33].

Ramachandran [64] pour sa part s'est intéressé à la perception de l'esthétisme dans l'art. Ses recherches sont en lien avec celles présentées dans ce mémoire dans le sens où l'appréciation et l'intérêt de l'observateur sont critiques pour qu'une visualisation fonctionne bien. De plus, il est possible de jouer avec certaines règles propres à l'art pour les appliquer à l'intérieur de la visualisation. Par exemple, le *peak shift* ou contraste peut influencer la perception d'un phénomène à travers les émotions de l'utilisateur. Parfois, il est souhaitable pour mettre l'accent sur un phénomène, mais parfois ça peut induire l'observateur en erreur si le phénomène est dû à la construction de la visualisation ou au hasard plutôt qu'aux données représentées. Ramachandran discute aussi des groupements possibles et comment ils peuvent influencer la perception à travers les émotions.

Tufte a aussi écrit sur la perception et la façon de représenter les données. Dans son livre [77], il insiste d'abord sur le fait que la manière de présenter l'information est aussi importante que les données elles-mêmes. La présentation claire des données fait d'ailleurs souvent partie de la solution. Il discute de la métaphore et de son influence dans la création d'un effet percutant sur l'observateur. Il discute aussi de l'importance du réalisme et du fait qu'il ne faut pas aller contre les éléments de perception ancrés dans l'imaginaire collectif des gens. On sait par exemple que les couleurs bleues sont considérées froides et que les couleurs rougeâtres sont chaudes. Inverser ce schéma de pensée peut déstabiliser l'observateur. Il discute aussi de la couleur et du fait qu'il faut apprendre à la décortiquer en différents éléments en s'assurant que chacun des éléments ne se contredit pas. Par exemple, il est possible que la luminosité ne suive pas la même échelle que la saturation de la couleur. Ceci dérangera l'observateur de manière inconsciente.

2.4 État de l'art en visualisation

Dans les sections qui suivent nous survoleront les travaux principaux reliés à trois thèmes, soient la visualisation du code du logiciel, la visualisation des métriques et de la qualité et finalement la visualisation de l'évolution du logiciel.

2.4.1 Visualisation du logiciel et son code

Reiss *et al.* [66] ont développé un outil pour représenter différents artefacts du code. Un aspect important de leur outil repose sur une interface graphique permettant à l'utilisateur de choisir et de gérer les items qu'il veut visualiser ainsi que les liens qui les unissent. L'outil choisit ensuite parmi une gamme de sept visualisations différentes (*Box trees, Tree maps, File maps, Layouts, Point maps, Connected point maps, Spirals*) la plus appropriée en fonction de la tâche et des items à observer. Dans d'autres travaux [67], Reiss observe le comportement des programmes en exécution. Il utilise des rectangles en montrant trois dimensions soit la couleur, la hauteur et la largeur pour afficher le déroulement du programme. Son outil essaie de ralentir le programme observé de façon minimale pour ne pas perturber la lecture des résultats.

Parmi les premières recherches s'intéressant à la visualisation du code lui-même, on compte l'outil SEESOFT développé par Eick *et al.* [16]. Cet outil a par la suite été l'inspiration de toute une famille d'outils permettant de représenter les lignes de code à l'aide de segments de couleur. Dans ce cas, les fichiers sont représentés côte à côte et chaque fichier contient un certain nombre de segments représentant les vraies lignes du fichier. Les segments ont un pixel de hauteur et la largeur en pixels correspond au nombre de caractères de la ligne de code. La couleur de chaque segment donne des statistiques extraites de la ligne, soient l'âge de la ligne, son auteur, son type, etc. Leur outil permet d'afficher jusqu'à 50 000 lignes de code dans des conditions idéales. La figure 2.5 montre un exemple de l'utilisation de l'outil SEESOFT.

Similairement, Orso *et al.* [60] ont développé un outil comportant trois vues différentes sur les variables dans un programme. Une première vue concerne un très bas niveau, c'est-à-dire la présentation du texte lui-même. La différence majeure est qu'un code de couleur est utilisé pour représenter différentes statistiques sur les expressions dans le code. La deuxième vue consiste en la représentation des fichiers d'une façon très similaire à celle utilisée dans l'outil de Eick *et al.* [16]. La troisième vue représente tous les fichiers au niveau du programme avec un *Treemap*[3] dans lequel ils utilisent la couleur

[3]Le *Treemap* sera discuté plus en détail dans le chapitre 3.

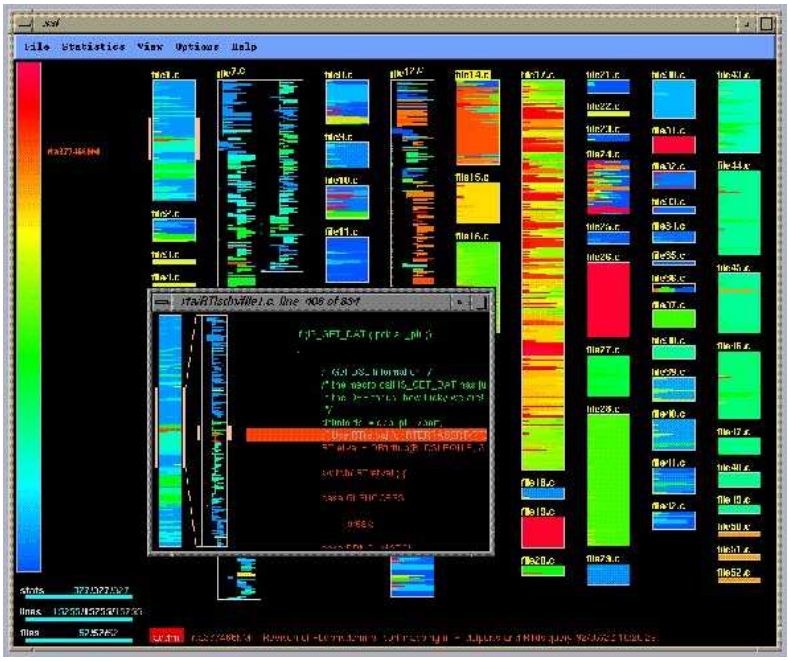


Figure 2.5 – SEESOFT [16]. Cette image montre un exemple de l'utilisation de l'outil SEESOFT où les lignes de code sont représentées comme des segments de différentes couleurs.

et la luminosité pour représenter deux informations (par exemple sur les exceptions ou le profiling) à la fois. La grosseur des différents fichiers est directement reliée à la grosseur des rectangles générés par l'algorithme. Leurs représentations peuvent aussi être mises à jour en fonction de l'exécution du programme pour suivre les modifications apportées aux différentes caractéristiques étudiées.

Lommerse *et al.* [51] pour leur part, ont représenté de l'information similaire dans des vues semblables à celles de Orso *et al.* [60]. Par contre, au lieu de colorier les expressions individuelles, les auteurs utilisent plutôt des coussins pour englober les expressions et les différents niveaux de portée dans le code. Les coussins ne sont en fait que des rectangles sur lesquels on applique un éclairage particulier pour donner une illusion de trois dimensions. De cette façon, les rectangles semblent bombés et il est plus facile de déterminer la région occupée par celui-ci. Une amélioration de cette technique appelée les coussins à plateau (*plateau cushions*) consiste à aplatir une bonne partie du coussin pour

obtenir un éclairage uniforme sur cette région. Ceci permet une lecture plus facile du texte qui se trouve dans la portée sous le coussin. Ces coussins peuvent aussi être réduits à un point tel qu'on arrive à une représentation par ligne de code où l'on perçoit différents niveaux d'imbrication à l'aide de la superposition de coussins et où le texte devient illisible et sans importance. Les auteurs utilisent le même principe de coussins à l'intérieur du *Treemap* pour représenter tous les symboles d'un programme. Ces symboles incluent les différents paquetages ou espaces de nom jusqu'aux noms des différentes variables. Par contre, l'information contenue dans le corps des fonctions n'est pas considérée. La figure 2.6 montre un exemple de leurs travaux contenant quelques coussins.

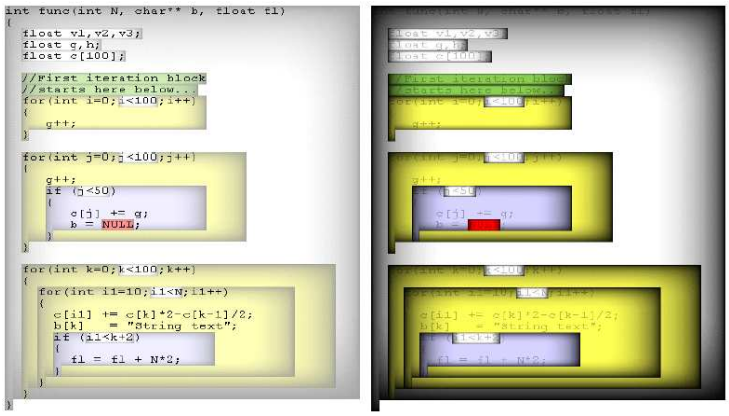


Figure 2.6 – *Visual Code Navigator* [51]. Cette image montre un exemple de l'utilisation de l'outil *Visual Code Navigator*. Cet exemple concerne l'affiche au plus bas niveau de visualisation où le texte est visible. Les coussins sont présentés avec différents niveaux de transparence.

Knight et Munro [39] ont fait une première tentative avec la métaphore de la ville en établissant les bases sur l'utilisation d'une métaphore pour représenter le logiciel. Ils ont surtout insisté sur l'importance de la métaphore dans la représentation des logiciels. Ces auteurs représentent les méthodes et les classes en fonction de leur taille et de leurs caractéristiques. Un exemple de ces caractéristiques est le niveau d'accès des méthodes. Leur approche est plus centrée sur la navigation et la réalité virtuelle. La perception et la sensation de l'utilisateur sont alors primordiales pour l'aider dans sa compréhension du logiciel. Du même souffle, les auteurs critiquent les représentations de type graphes

avec des liens et des nœuds parce que ces dernières ne permettent pas la représentation de plusieurs centaines d'éléments.

Panas *et al.* [62] utilisent aussi une métaphore de la ville pour représenter le logiciel. Leurs graphismes sont très réalistes et très attrayants pour l'utilisateur. Les édifices et les rues réalistes sont agrémentés de détails tels que des arbres et des lampadaires. La fonction première de ce logiciel est de faire le suivi des classes dans une perspective de gestion de projet. Par exemple, on peut voir un édifice en feu quand il y a un bogue dans une classe ou voir apparaître des noms de programmeurs sur les édifices pour distribuer le travail. Le but est donc d'avoir une interface plus conviviale pour comprendre et gérer des projets de grande envergure. Une capture d'écran de leur outil est présentée à la figure 2.7.

Figure 2.7 – Visualisation à l'aide de villes [62]. Dans leur système de visualisation, Panas *et al.* ont choisi d'utiliser la métaphore de la ville associée à des graphismes esthétiques et réalistes.

Finalement, dans la catégorie des visualisations affichant les informations de bas niveau sur le code ou les lignes de code elles-mêmes, on peut compter les travaux de Marcus *et al.* [54]. Dans ces travaux, on montre toujours les lignes les unes à la suite des autres à la manière de SEESOFT [16]. Par contre, les auteurs utilisent le 3D pour représenter plus de caractéristiques pour chaque élément. Des informations de base comme des statistiques sur les lignes de code sont associées à des barres en trois dimensions sortant du plan. Le plan peut être pivoté à la guise des utilisateurs. La longueur des barres et leurs couleurs sont utilisées pour représenter les différentes caractéristiques. Les auteurs

utilisent autant le dessus du plan que le dessous du plan pour représenter les éléments. Ceci leur permet de multiplier les caractéristiques visibles. Ils utilisent différents angles de vue selon les tâches et exploitent la transparence pour accentuer certaines parties de la représentation.

Toutes ces contributions sont très intéressantes pour vérifier l'information de bas niveau dans le logiciel. La grande majorité font preuve d'ingéniosité pour afficher le plus d'éléments possibles tout en montrant le plus de dimensions possibles pour chaque élément. Par contre, bien que ces types de visualisation soient efficaces, la plupart d'entre elles ne considèrent pas le système visuel humain assez en profondeur. De plus, ces représentations restent vagues sur les applications possibles et ce qui peut être découvert à l'aide de leur outil. Elles ne s'intéressent donc pas directement aux métriques et à la qualité du logiciel comme notre approche. De plus, la tendance à vouloir visualiser plusieurs éléments de bas niveau rend impossible la visualisation de logiciels entiers ayant une taille normale de façon efficace. Certaines approches pallient à ce problème en affichant plusieurs vues d'un même logiciel. Un problème de cette approche est que l'utilisateur peut avoir de la difficulté à faire le lien entre les différentes vues si ces dernières ne sont pas bien intégrées. Comme récemment stipulé [68], il ne faut pas qu'il y ait un saut trop important entre les différentes vues pour que le saut dans le cerveau des utilisateurs de l'outil soit lui aussi petit.

2.4.2 Visualisation de métriques et de la qualité

Il existe une autre famille d'outils qui représentent les métriques du logiciel plutôt que de représenter les éléments de bas niveau retrouvés dans le logiciel. Ces travaux s'orientent souvent plus vers la qualité du logiciel et la détection de problèmes dans ce dernier. Lanza et Ducasse [46] utilisent des rectangles pour représenter différents éléments du logiciel. Les métriques pour chacun de ces éléments sont représentées par la hauteur, la largeur et la couleur des classes. À quelques occasions, les entités sont montrées comme un nuage de points et leur position est significative, mais malheureusement, les auteurs ne montrent pas d'exemples où plus de trois métriques sont visibles à la fois. Les autres vues affichent des hiérarchies de classes et de méthodes reliées par des lignes

ou encore un alignement des différentes entités ordonnées selon une métrique donnée. Cette méthodologie est appliquée pour répondre aux besoins des logiciels industriels.

Balzer et Deussen [2] ont développé une nouvelle façon de représenter les hiérarchies inspirées de l'algorithme du *Treemap*. Le *Voronoï Treemap* utilise les tessellations de *Voronoï* pour donner une forme autre que rectangulaire aux nœuds. Ainsi ils peuvent insérer plus de formes avec des ratios hauteur/largeur plus balancés que ceux que l'on retrouve pour l'algorithme original du *Treemap*. Aux feuilles de leur arborescence, la couleur représente les métriques logicielles. Les auteurs ne s'arrêtent pas au niveau des classes, mais visualisent aussi les méthodes et les attributs à l'aide de leur structure hiérarchique. Bien que la forme de départ rectangulaire accommode bien les écrans normaux, la technique peut partir de n'importe quelle forme et générer l'arbre résultant d'une hiérarchie de classes. Le temps de calcul pour la relaxation de l'espace réservé à chacun des nœuds prend toutefois plus de temps que le *Treemap*. De plus, pour éviter la confusion entre les différents niveaux de la hiérarchie, il est nécessaire d'élargir les séparateurs en fonction du niveau. Les directions de séparation devenant infinies, la hiérarchie est plus difficile à déterminer visuellement. La figure 2.8 donne un exemple de leur représentation hiérarchique.

Termeer *et al.* [76] ont développé un outil pour visualiser plusieurs métriques à la fois avec le diagramme de classes *UML* comme support initial. Ils ajoutent des éléments soit en deux dimensions ou en trois dimensions à l'intérieur des rectangles représentant les classes. Ils peuvent ajouter plusieurs artefacts dans l'espace alloué des boîtes pour représenter plusieurs métriques à la fois. Il est facile de retrouver les artefacts correspondants aux mêmes métriques dans les autres classes en comparant leur forme et leur couleur. Ils utilisent aussi des régulateurs de transparence pour permettre à l'utilisateur de tantôt voir les artefacts ou tantôt voir le diagramme lui-même. Leur outil laisse la liberté à l'utilisateur de choisir les agencements de métriques dont il a besoin. L'information structurelle via les diagrammes *UML* est ainsi préservée et des informations supplémentaires sur les métriques sont aussi disponibles. La figure 2.9 montre un exemple de leur approche.

Graham *et al.* [23] ont pour le part exploité la métaphore du système solaire pour afficher les éléments d'un programme *Java.* Les étoiles représentent les paquetages autour

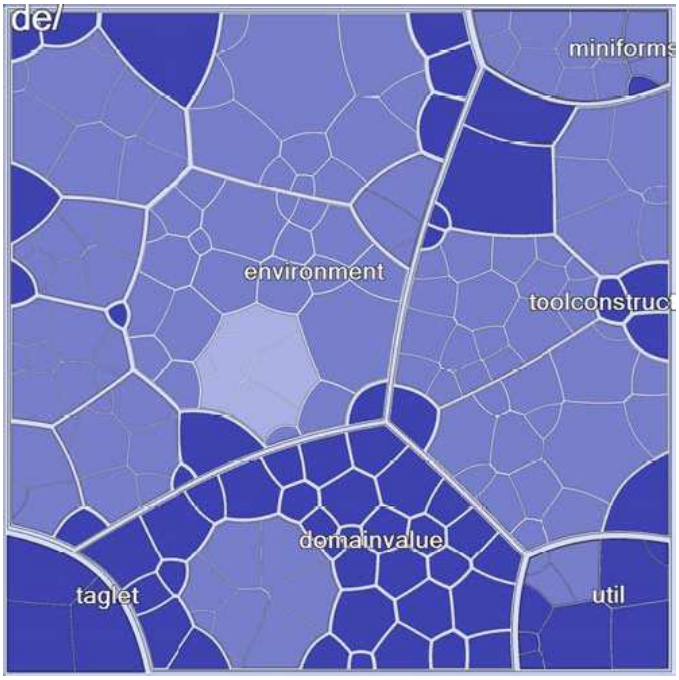


Figure 2.8 – *Voronoï Treemap* [2]. Le *Voronoï Treemap* est une façon nouvelle de représenter les hiérarchies. Comparativement au *Treemap,* les nœuds sont modélisés par des polygones arbitraires.

desquels gravitent d'autres étoiles (paquetages). Les planètes représentent les classes ou les interfaces selon leur couleur, et la taille de ces planètes représente la taille réelle des classes en nombre de lignes de code. La taille des planètes peut être associée à une autre métrique au besoin. L'orbite des planètes indique la profondeur des classes dans l'arbre d'héritage. Des liens affichés sous forme de ligne peuvent être activés pour représenter les liens d'héritage ou encore les liens de couplage entre les éléments. Des valeurs limites peuvent être utilisées pour filtrer l'information perçue et permettre à l'utilisateur de retrouver des nuances plus subtiles dans des éléments correspondant à des critères qui l'intéressent. De plus, plusieurs métriques sont disponibles en sélectionnant une entité en particulier.

Holten *et al.* [32] utilisent la couleur, la teinte, la texture et le relief pour montrer des métriques réparties dans une représentation traditionnelle du *Treemap* en deux dimensions. Par contre, les auteurs ne montrent que deux métriques à la fois. La luminosité

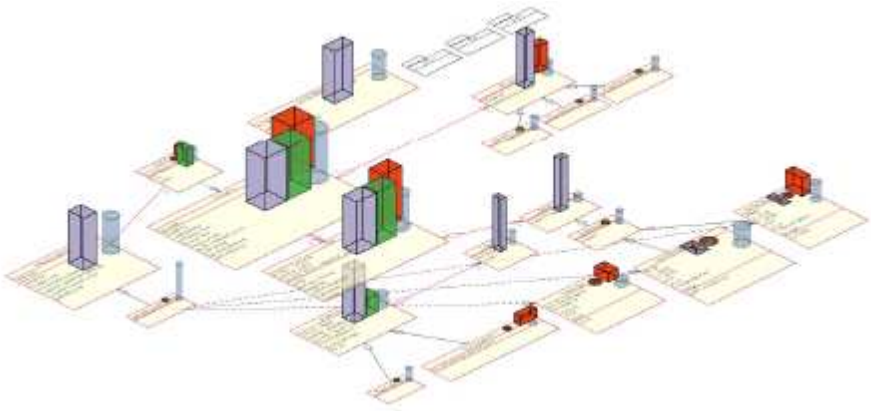


Figure 2.9 – Visualisation à l'aide d'UML. Cette approche utilise la représentation UML par dessus laquelle des métriques sont ajoutées. Chaque artefact donne une valeur pour une métrique.

est utilisée pour accentuer les contours à l'aide de la technique du coussin [79] mentionnée plus haut. Le relief, implanté à l'aide de la technique infographique bien connue du *bump mapping*, sert à accentuer l'effet des textures pour que leur lecture soit plus facile. Ces auteurs vont aussi jusqu'à afficher les métriques des méthodes.

Le problème majeur des solutions mentionnées ci-haut est leur difficulté à gérer la grande quantité de données. Tantôt, le nombre de dimensions fait défaut, tantôt le nombre d'entités affichées fait défaut. Bien que certaines approches soient efficaces dans un des secteurs ou les deux en théorie, la quantité d'informations réelle perçue par l'utilisateur reste en deçà de ce que VERSO propose. Par exemple, on ne peut pas demander à l'utilisateur de reconnaître la granularité d'une texture complexe quand l'espace disponible pour représenter une information s'avère être une bande de 2 par 10 pixels. De plus, il faut souligner que ces recherches n'ont pas toutes été conçues en s'appuyant sur les forces et les faiblesses du système visuel humain. Notre approche est à la fois axée sur la présentation de métriques et de liens architecturaux entre les éléments, et permet de voir et comprendre une grande quantité d'informations en une fraction de seconde.

2.4.3 Visualisation de l'évolution du logiciel

Quelques chercheurs ont aussi décidé de s'intéresser à la visualisation de l'évolution du logiciel, mais ce ne sont pas toutes leurs approches qui s'attardent à représenter la qualité du logiciel.

Plusieurs approches utilisent des représentations statiques où le logiciel et son évolution sont montrés comme une matrice. En général, l'axe des X représente le temps qui s'écoule donc les différentes versions, tandis que l'axe des Y représente les différents éléments dans le logiciel (fichiers, classes, méthodes, lignes de code, etc.). C'est le cas entre autres de Lommerse *et al.* [51] qui proposent un outil de visualisation où toutes les lignes d'un fichier sont présentées sur l'axe des Y et où le temps est présenté sur l'axe des X. Des pixels représentent des lignes de code à un instant donné. La couleur du pixel représente soit le type de la ligne de code, soit l'auteur de la ligne de code. Dans les deux cas, la profondeur de la portée de la ligne de code est représentée par la luminosité du pixel. Lommerse *et al.* peuvent combiner les images provenant de différents fichiers et les placer dans une matrice à deux colonnes, selon les types et les auteurs. En alignant les axes du temps, il est possible de voir si une prériode précise a influencé les fichiers de la même façon. Cette approche est très similaire à celle de Wu *et al.* [80] qui utilise une technique par remplissage de pixels, mais où cette fois-ci, chacun des pixels représente un fichier à un temps donné.

Lanza et Ducasse [45] disposent les classes dans une matrice où les rangées représentent une classe précise et où les colonnes sont les différentes versions d'un logiciel. Cette fois-ci, les classes ne sont pas présentées comme des pixels, mais comme des rectangles dont la largeur et la hauteur sont associées à des métriques. Dans cet article, les auteurs dépassent la simple définition d'un modèle de visualisation et ont aussi découvert des anomalies en basant la nomenclature sur une métaphore de l'astronomie. Par exemple, une classe dont les métriques grandissent subitement est appelée une super nova. Ils étudient différentes anomalies de ce genre tout en indiquant la signification de ces problèmes et la marche à suivre pour les régler éventuellement. Ils présentent finalement des analyses sur des programmes réels de tailles variant de petite à moyenne. Ils

observent alors des particularités dans l'évolution comme des phases de stabilisation ou encore des sauts. Un exemple de leur approche est montré sur la figure 2.10.

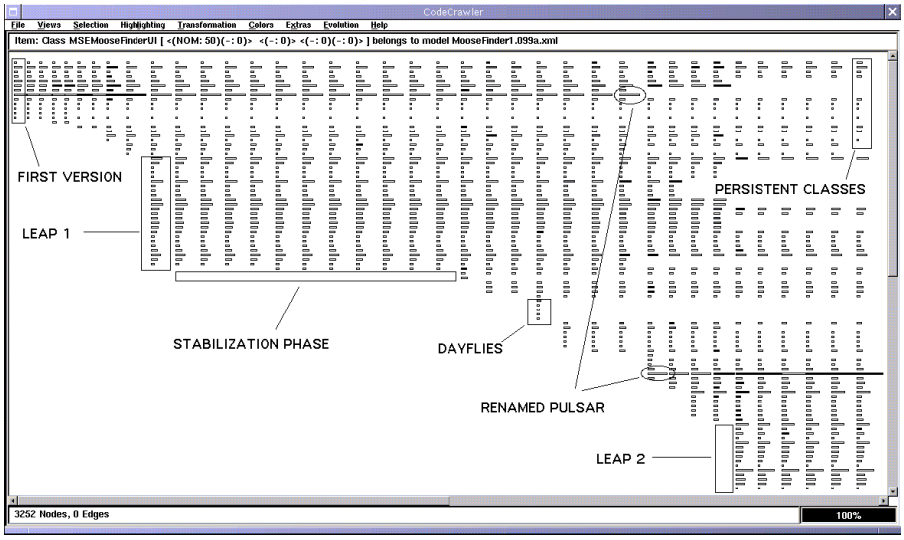


Figure 2.10 – Matrice d'évolution [45]. Dans cet exemple, le temps est représenté sur l'axe des X, les différentes classes sont représentées sur l'axe des Y et les métriques sont encodées dans les caractéristiques de chaque petit rectangle. Certaines observations concernant des sauts et des phases de stabilisation ont été ajoutées à l'image.

Voinea et Telea [78] représentent aussi l'évolution en une matrice avec les fichiers en rangées et les différentes versions en colonnes. Par contre, leur version est plus intéressante esthétiquement puisque tant qu'un fichier précis n'est pas modifié, il sera affiché comme une bande. On observe alors des coupures sur les lignes seulement quand une modification de fichier est effectuée. Ceci crée une impression de continuité et évite à l'utilisateur de revérifier s'il n'y a pas de différences subtiles entre deux versions pour chacun des fichiers. Les fichiers peuvent être triés selon différents critères, par contre, le plus efficace selon les auteurs est le tri par similarité. Les groupements formés permettent de retrouver facilement un fichier qui a eu le même processus d'évolution qu'un autre. Ensuite, les auteurs utilisent les marges gauches et supérieures de leur matrice pour afficher des métriques relatives aux fichiers et au temps respectivement. Des petits histogrammes indiquent les fluctuations des métriques. Pour la marge de gauche, l'histogramme est tout simplement transformé en diagramme à barres.

Le principe de la matrice a aussi été repris par Mesnage et Lanza [56], par contre cette fois, la matrice est en trois dimensions et placée à l'intérieur d'un cube imaginaire. La troisième dimension sert pour l'auteur d'une modification. Chaque élément visualisé est affiché comme un cube. La couleur du cube à un emplacement particulier représente le nombre de changements effectués dans le fichier. Une autre vue montre des informations sur les auteurs. Cette fois-ci, il s'agit d'une représentation semblable au nuage de points en trois dimensions. La position du point sur chacun des axes représente une information, la couleur représente une information et les trois dimensions du cube (hauteur, largeur, profondeur) représentent aussi une information. On choisit souvent de représenter la même information à l'aide de différentes caractéristiques pour éviter la surcharge. Les informations à visualiser sont toutes en lien avec des paramètres extraits des systèmes de contrôle de fichiers. Par exemple, on retrouve la première date à laquelle un auteur a soumis des changements, le nombre de changements, le nombre de fichiers changés à travers toutes les modifications, le nombre de lignes de code, etc. Il existe aussi une autre vue qu'ils appellent l'activité des fichiers. Cette vue est très semblable à celle des auteurs sauf que les cubes faisant office de points dans le nuage de points représentent en fait des fichiers. La figure 2.11 montre un exemple de cette approche.

Toujours dans le domaine de la visualisation de données extraites des logiciels de contrôle de versions, mais dans une autre optique, D'Ambros *et al.* [13] utilisent une vue à base de fractales pour représenter l'influence des développeurs sur les différents fichiers. Une technique semblable à celle du *Treemap* montre la quantité de travail de chaque développeur. Chaque développeur a une couleur différente. On commence avec un carré vide. Ensuite on prend le plus influent et on lui accorde une bande verticale à l'intérieur du carré. L'aire de cette bande est proportionnelle à l'influence du développeur. Ensuite on prend le deuxième plus influent développeur et on trace une bande horizontale proportionnelle à son influence dans l'aire restée vacante du carré initial. On continue tant qu'il n'y a plus de développeurs impliqués en alternant entre des rectangles verticaux et horizontaux. Ensuite, les auteurs calculent un indice à l'aide des fractales générées. Ces indices sont corrélés au nombre de fautes trouvées dans les modules pour connaître quel type de travail d'équipe est le plus efficace. Il s'est avéré que

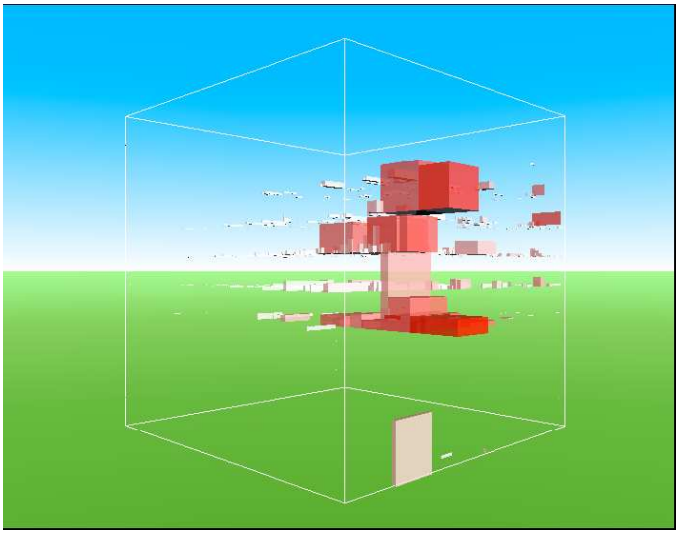


Figure 2.11 – *White Coats* [56]. Cette image montre la vue de l'activité des fichiers à l'intérieur de l'outil *White Coats*. Il s'agit d'une visualisation où des informations sont encodées à la fois dans la position des cubes et dans leurs caractéristiques.

plus le nombre de participants à un fichier est grand, plus il est susceptible de contenir des fautes.

Collberg *et al.* [11] présentent une approche basée sur les graphes pour afficher les informations extraites du code et des systèmes de contrôle de versions. Les types de graphes représentés sont le graphe hiérarchique, le graphe de flot de contrôle et le graphe des appels. Le graphe est animé au fur et à mesure que le code change à travers les différentes versions. Pour adoucir les transitions d'un graphe vers un autre, on fusionne toutes les versions du graphe pour connaître la proportion du temps où chaque nœud est présent. On fait la même chose pour les arêtes entre les nœuds. Les nœuds se repoussent entre eux en fonction de leur poids et les arêtes les retiennent ensemble aussi en fonction de leur poids. De plus, les nœuds sont rattachés à leur homologue des versions voisines à l'aide d'arêtes virtuelles. Cette technique empêche les représentations de changer drastiquement lors les différentes transitions. Ces dernières sont d'ailleurs animées avec des images intermédiaires pour permettre à l'utilisateur de mieux suivre les transformations. Dans toutes les représentations, la principale information montrée est la date de modification du code. Plus un nœud a été modifié récemment, plus sa couleur est près de celle

de son auteur, sinon, les nœuds tendent vers la couleur bleue.

Dans une récente publication D'Ambros et Lanza [12] utilisent le cercle pour observer l'évolution du couplage logique entre fichiers et modules. Le couplage logique est mesuré par la propension de deux items à être modifiés en même temps. Un cercle représentant un module choisi par l'utilisateur est placé au milieu d'un plus grand cercle. On accorde un certain angle à chaque module en fonction du nombre de fichiers présents et on répartit chaque fichier autour du cercle à l'intérieur de leur module à une distance angulaire égale. Ces fichiers sont aussi représentés par des cercles et leur couleur et grosseur peuvent recevoir n'importe quelle métrique fournie par l'utilisateur. Plus un fichier s'approche du centre du cercle, plus il est couplé avec le module central. L'utilisateur peut choisir différentes tranches de temps et calculer les représentations correspondantes. Il n'y a pas d'animation par contre dans ce cas-ci. Pour visionner un autre module, il s'agit de le sélectionner et il deviendra le module central. Cette forme d'interaction peut aider l'utilisateur à découvrir quels fichiers ont des liens forts dans deux modules différents. Si un fichier s'approche trop du centre, il faut peut-être penser à l'introduire dans le module central à cause de son fort couplage avec celui-ci. Cette approche est intéressante parce qu'elle s'attarde au niveau de granularité du module (ce qui est rare) et permet des réusinages concrets. Un exemple de visualisation possible avec cet outil est montré à la figure 2.12.

Pour finir, Pinzger *et al.* [63] représentent l'évolution de plusieurs métriques à la fois. Pour ce faire, ils utilisent des diagrammes de *Kiviat* où plusieurs métriques sont placées à des angles différents autour d'un cercle. Sur le rayon représentant une métrique, on place un point dont la distance détermine la valeur de la métrique. On fait de même avec toutes les métriques autour du cercle. Ensuite, pour aider l'utilisateur à mieux identifier les valeurs, on relie les points en un polygone qu'on remplit d'une couleur. Cette approche s'intéresse encore une fois aux modules et non aux classes particulières. Les auteurs montrent les liens entre les différents modules en reliant plusieurs diagrammes à l'aide de lignes. L'évolution entre en jeu au moment où les diagrammes possèdent plusieurs couches qui sont superposées. On utilise alors différentes couleurs pour dessiner les polygones associés à différentes versions. Il est ainsi possible de voir si les métriques

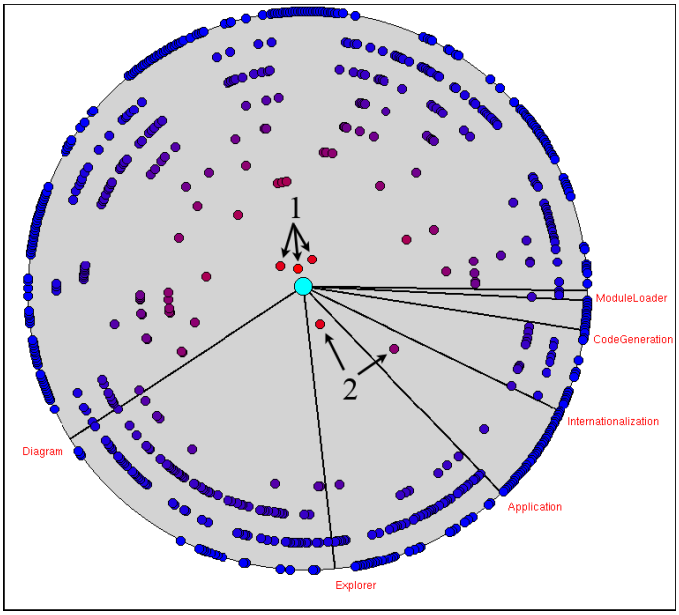


Figure 2.12 – Visualisation du couplage logique [12]. Dans ce type de visualisation, un module est choisi comme central et les classes des autres modules sont réparties autour de celui-ci. Plus les cercles représentant les classes s'approchent du cercle central, plus elles sont couplées à ce module en question.

augmentent, diminuent ou restent stables entre les différentes versions. Si une couleur domine largement, c'est que le passage à cette version a généré une grand chamboulement dans les métriques étudiées.

Les techniques décrites plus haut sont intéressantes dans la mesure où elles donnent des informations sur l'évolution du logiciel et qu'il est possible de faire des analyses poussées à l'aide de ces visualisations. Par contre, peu de ces approches s'intéressent directement à la qualité du logiciel. Dans certains cas, on montre des informations de bas niveau qui sont difficilement utilisables pour la détection de phénomènes liés à la qualité. Dans d'autres cas, on s'intéresse à un problème très précis de la qualité comme le couplage logique et la vision globale du système est laissée de côté. Quelques approches utilisent pour leur part les mêmes métriques dans les mêmes desseins que VERSO. Par contre, elles présentent l'information sur une seule image en deux dimensions. Ceci oblige à faire des compromis sur le nombre de dimensions à afficher par entité ou encore

sur le nombre d'entités visibles simultanément. De plus, ce genre d'approche, bien que donnant une bonne idée de la globalité de l'évolution d'un logiciel, ne met pas l'emphase sur les transitions suspectes. Très peu d'approches ont utilisé l'animation pour représenter les transitions (à l'exeption de [11]). Notre approche est donc innovatrice en considérant la nature de la visualisation présentée dans VERSO.

Bien que peu d'approches se soient intéressées à l'animation et particulièrement à l'animation du placement dans le génie logiciel, quelques auteurs ont par contre étudié la question dans le contexte général de la visualisation. Par exemple, Nguyen et Huang [58] ont développé une technique qui permet de passer d'une représentation plus large d'un arbre vers une représentation où un de ses descendants (sous-arbre) est représenté. Ceci est fait par une succession de glissements où le nœud choisi remonte la hiérarchie en remplaçant son parent à chaque fois. Des images intermédiaires aident l'utilisateur à suivre les changements apportés à la représentation. Pour leur part, Bladh *et al.* [5] proposent une animation semblable pour passer d'une représentation d'un *Treemap* général à une image représentant un sous-arbre de celui-ci. Par contre, il ne s'agit que d'un agrandissement de la portion en question et il n'y a aucun réagencement effectué à l'intérieur de la sous-portion pour la rendre plus carrée. Finalement, Fekete et Plaisant [18] ont aussi travaillé sur l'animation des *Treemaps*. Bien que leurs nœuds ne soient pas réagencés, ils changent toutefois de forme à travers l'animation. En fait, la mesure donnant la taille de chaque nœud est modifiée et il en résulte des transformations importantes dans la visualisation. Leur approche propose de déplacer les éléments dans un premier temps et de les redimensionner dans une deuxième phase de l'animation. Le déroulement simultané de ces deux opérations aurait pour conséquence de confondre l'utilisateur.

CHAPITRE 3

VISUALISATION DE LOGICIELS

Dans le chapitre précédent, les métriques et la perception visuelle humaine ont été étudiées. Les métriques fournissent un vecteur de données qu'il faut représenter à l'écran pour chaque classe du logiciel, alors que le système visuel humain impose des balises et des critères de performance pour une visualisation efficace. Dans ce chapitre, il est question de la manière de représenter le logiciel dans VERSO. Ce chapitre se divise en trois grandes sections : la section 3.1 explique la représentation graphique de chaque classe individuellement ; la section 3.2 discute de l'agencement de ces classes à la fois pour représenter le plus grand nombre possible d'entités et aussi obtenir de l'information plus globale sur le logiciel ; alors que la section 3.3 présente une évaluation de notre approche de visualisation. On se concentre tout d'abord sur une version en particulier d'un logiciel, alors que le chapitre 5 traitera de la visualisation de plusieurs versions d'un même système.

3.1 Représentation des classes

La classe est l'élément de granularité le plus précis auquel nous nous sommes intéressés durant nos recherches. Elle permet la compréhension d'un logiciel à un niveau de granularité élevé tout en représentant un élément très concret sur lequel il est possible de prendre des mesures. Ces mesures peuvent par la suite être présentées à l'écran plus simplement que l'entité elle-même. Il sera tout de même possible de pousser l'investigation à un niveau de granularité plus bas dans de futures expérimentations avec la visualisation du logiciel.

3.1.1 Représenter l'intangible

Il existe plusieurs domaines où la visualisation est facilitée par la nature des éléments représentés. En effet, dans plusieurs cas, on représente des éléments concrets qu'il est

possible d'afficher en restant fidèle à leur forme réelle. C'est le cas par exemple de la visualisation biomédicale où les organes peuvent être dépeints de façon fidèle à leur pendant réel [75]. C'est le cas aussi pour la simulation en physique [17] (moteur de voiture, simulation de collisions entre différents objets dans différents milieux, observation du mouvement des astres, etc.). Les phénomènes ou les prédictions météorologiques peuvent aussi être représentés en étant fidèles à la forme des manifestations observables dans la nature [77].

Par contre, certaines données tels les indicateurs socio-économiques sont plus abstraites. Dans ce cas, les scientifiques cherchant à visualiser leurs données peuvent se fier sur des cartes [31, 52] par-dessus lesquelles des données sont affichées. Ceci leur permet de donner une information supplémentaire, soit la variable géographique, tout en donnant un support intuitif à l'utilisateur pour qu'il retrouve et compare facilement l'information. Dans d'autres domaines encore, des modèles acceptés par la communauté comme représentatifs d'une réalité abstraite existent déjà et peuvent être utilisés pour représenter les données. On pense entre autres au modèle de l'atome et aux représentations des molécules en chimie. Par contre, certains types de données ne peuvent pas être représentées naturellement. La situation financière d'une entreprise en fonction du temps, le volume de matière première utilisée en fonction de chacun des produits fabriqués par une entreprise en sont des exemples. Pour ces formes de données, on utilise souvent des tableaux ou des graphiques plus traditionnels (histogrammes, diagrammes en pointe de tarte, graphes avec liens entre les entités, nuage de points). Ces représentations sont efficaces dans certaines situations, mais ont aussi plusieurs lacunes dues à leur trop grande généralité.

Le premier problème rencontré pour représenter un logiciel est l'absence de forme concrète de ce dernier. En effet selon Knight et Munroe [39], le code n'a pas de forme si ce n'est que le texte qui le représente. Ce fait rend la représentation du code très difficile. Le code a une sémantique qui existe dans le but unique d'être comprise par le programmeur et la machine et n'a pas de réalité en dehors de cet usage.

Étant donné cette problématique, nous sommes contraints de choisir des formes arbitraires pour représenter le code. Il est possible d'utiliser des formes de visualisation qui

existent déjà. Par exemple, une notation graphique définie dans le standard UML [59] sert déjà à représenter les programmes. Ce standard utilise des rectangles pour modéliser les classes. De plus, cette notation a du sens puisqu'il n'est pas rare de considérer du code comme des boîtes fermées qui traitent des entrées et qui génèrent des sorties. Il est aussi possible d'emprunter des techniques à d'autres domaines comme la cartographie pour créer un support artificiel pour disposer les classes à représenter.

3.1.2 Caractéristiques graphiques

Les caractéristiques visuelles des éléments composant une représentation graphique sont à la base de toute visualisation. Leur choix doit se faire à la fois dans un souci d'efficacité pour la compréhension de l'analyste, d'efficacité en terme d'affichage par la machine et aussi dans un souci d'esthétisme pour l'analyste. La justesse des caractéristiques graphiques fait la différence entre les bonnes et les mauvaises plateformes de visualisation.

3.1.2.1 Formes choisies

Comme stipulé plus haut, l'absence de formes concrètes du code oblige le choix de formes arbitraires. Dans le cadre de la visualisation de logiciels orientés objets, il est possible de distinguer deux groupes d'éléments : les classes et les interfaces. Il est intéressant de pouvoir distinguer les deux durant la visualisation d'un logiciel pour mieux comprendre les interactions des différentes parties du logiciel. Des formes différentes sont donc utilisées pour représenter les deux éléments. Le cube et le cylindre ont respectivement été choisis pour chacun des deux éléments. Des expérimentations ont aussi été faites avec les logiciels à aspects. Des pyramides représentent alors les aspects qui viennent se greffer au code objet traditionnel.

3.1.2.1.1 Boîte La boîte a principalement trois caractéristiques utilisables pour représenter de l'information : sa hauteur, sa couleur et son angle de rotation par rapport à l'axe des Y (pointant vers le haut dans les représentations). Sa base, deux fois plus large

que profonde, n'est pas carrée pour permettre une meilleure perception de la rotation. Sa face plane du dessus a sa normale parallèle à l'axe des Y, ce qui rend la comparaison des couleurs plus facile entre les différentes boîtes d'un logiciel (voir section 3.1.2.2). L'œil et le cerveau humain ont aussi beaucoup de facilité à identifier cette forme simple [61]. Ceci laisse plus de temps à l'analyste pour se concentrer sur les autres caractéristiques. Toutes les représentations possèdent la même aire sur le plan pour réduire les biais dans la perception.

3.1.2.1.2 Cylindre Le cylindre est semblable à la boîte sauf que sa base est circulaire. Il est donc impossible d'utiliser la rotation autour de l'axe des Y pour représenter une caractéristique. Ce n'est pas un problème puisque les interfaces n'ont pas de code à l'intérieur de leurs méthodes et toute une série de métriques deviennent donc non-significatives. On a donc besoin de moins de caractéristiques graphiques pour représenter les interfaces. Il est aussi relativement facile à afficher pour les cartes graphiques puisque l'approximation que nous en avons faite possède seulement 10 facettes rectangulaires sur le contour pour un grand total de 30 triangles (10 pour la face du dessus et 2 par côté ensuite). Une simple interpolation des normales sur chaque triangle permet de donner l'illusion, par réflexion, que le cylindre est bien rond. Le cylindre a aussi une face, celle du dessus, dont la normale unique pointe vers l'axe des Y, ceci élimine les problèmes d'influence de l'illumination sur la perception de la couleur comme dans le cas de la boîte (voir la section 3.1.2.2). Il est aussi une forme facilement identifiable par l'œil humain et surtout facilement différentiable de la boîte, ce qui permet de retrouver facilement les interfaces à l'intérieur d'un grand logiciel.

3.1.2.1.3 Pyramide La pyramide, pour sa part, a toutes les caractéristiques de la boîte et a aussi une base rectangulaire pour mieux représenter l'angle de rotation autour de l'axe des Y. Elle a par contre l'inconvénient de ne pas avoir de face perpendiculaire à l'axe des Y. Ceci peut causer de l'interférence entre la couleur et l'intensité lumineuse à cause des angles de la surface (voir la section 3.1.2.2). Encore une fois, il s'agit d'une forme facilement distinguable.

3.1.2.2 Illumination dans VERSO

L'illumination à l'intérieur de VERSO utilise une lumière directionnelle. La direction de la lumière correspond au vecteur (-1,-3,2) dans un système de coordonnée de la main gauche. Cette direction est arbitraire, mais répond bien aux besoins de la visualisation. La lumières vient de derrière la caméra quand elle est à son point de départ et elle est déplacée un peu à sa droite pour donner un effet plus naturel. Les objets sont donc dans la lumière pour ce premier apreçu du logiciel. Ce genre d'illumination est souvent choisie pour représenter des scènes simples. Bien que n'étant pas en mesure de simuler les particularités physiques complexes de la lumière nécessaire dans les images photo-réalistes, cette simplification est commune et donne un sentiment de lumière naturelle.

L'illumination aide l'analyste à percevoir la troisième dimension du polyèdre efficacement à cause des teintes de couleurs différentes présentes sur les polyèdres affichés. Les faces situées à l'opposé de la direction de la lumière devraient être noires. Cependant, ceci représenterait une perte d'information importante pour l'analyste lors de la navigation. Le problème est corrigé en donnant à ces faces la même couleur que celle de leur face opposée. Il ne s'agit pas d'une solution réaliste, mais d'un compromis visant à améliorer la lisibilité des données.

L'intensité de la lumière directionnelle est ajustée de manière telle que toutes les faces étant parallèles au plan XZ auront la couleur exacte de leur métrique. C'est-à-dire que si une classe possède une couleur bleue parfaite, celle-ci ne sera pas altérée par l'illumination lors de l'affichage pour la face parallèle au plan XZ, soit celle du dessus du polyèdre. La formule suivante donne la façon de calculer l'intensité de la lumière directionnelle pour obtenir cet effet.

$$I = \frac{1}{\vec{N} \cdot \vec{L}}$$

où $\vec{N}$ est le vecteur unité aligné à l'axe des Y et $\vec{L}$ est le vecteur unitaire représentant la direction de la lumière. Il est constant pour une visualisation donnée.

3.1.2.3 Autres caractéristiques graphiques

Plusieurs autres caractéristiques graphiques ont bien sûr été testées pour les différentes formes choisies. Par contre, ces caractéristiques étaient parfois très difficiles à interpréter ou parfois elles introduisaient un biais sur des caractéristiques déjà présentes. Par exemple, un seul axe de rotation est utilisé dans le but de représenter des caractéristiques. L'utilisation des autres axes de rotation ou encore de l'inclinaison (cisaillement) des différentes formes portaient à confusion et étaient difficilement distinguables depuis un point de vue où l'ensemble des classes étaient visibles. La décomposition de la couleur en RGB (Red, Green, Blue), c'est-à-dire isoler les composantes rouges, vertes et bleues de la couleur pour en faire trois caractéristiques n'a pas fonctionné non plus. En effet, le mélange n'est pas intuitif pour l'humain, car il n'arrive pas aisément à décomposer la couleur une fois observée [22]. Les autres couleurs introduisent un biais et la compréhension de chaque élément demande un effort cognitif très grand. Pour finir, quand on compare plusieurs centaines ou plusieurs milliers d'éléments entre eux, la largeur et la profondeur ont une influence certaine sur la perception de la hauteur [31]. En effet une classe plus large nous paraît moins haute et vice-versa. Le phénomène est le même pour la profondeur, qui de plus, demande que l'analyste cherche le point de vue adéquat pour comparer cette caractéristique.

Les trois caractéristiques principales choisies, soient la hauteur, un dégradé de couleur allant du bleu vers le rouge et la rotation par rapport à l'axe des Y sont intuitifs et réduisent le biais de façon satisfaisante pour nos besoins.

3.1.3 Association entre les métriques et les caractéristiques graphiques

La section 2.4 a répertorié certaines métriques intéressantes à l'étude du logiciel et la section 3.1.2 a justifié les caractéristiques graphiques choisies pour la visualisation. Il reste par contre à faire un lien judicieux entre ces deux composantes pour avoir une visualisation utile pour l'analyse des logiciels.

L'approche simple utilise une correspondance linéaire entre les métriques et les caractéristiques graphiques. Les valeurs dépassant les seuils fixés par l'analyste sont ra-

menées vers ce maximum ou ce minimum. Ces valeurs seuils sont considérées comme des maxima ou des minima pratiques. Cette correspondance est donnée par la formule suivante :

$$M'_v = \begin{cases} M_{min} & M_v < M_{min} \\ M_v & M_{min} \leq M_v \leq M_{max} \\ M_{max} & M_v > M_{max} \end{cases}$$

$$G_v = G_{min} + (G_{max} - G_{min}) \left(\frac{M'_v - M_{min}}{M_{max} - M_{min}} \right)$$

où M_v est la valeur de la métrique, G_v est la valeur graphique résultante, G_{min} et G_{max} sont respectivement les valeurs minimum et maximum pratiques de la caractéristique graphique, et M_{min} et M_{max} sont respectivement les minimum et maximum pratiques des métriques. La figure 3.1 montre la représentation de trois classes pour lesquelles les trois caractéristiques graphiques augmentent de la gauche vers la droite.

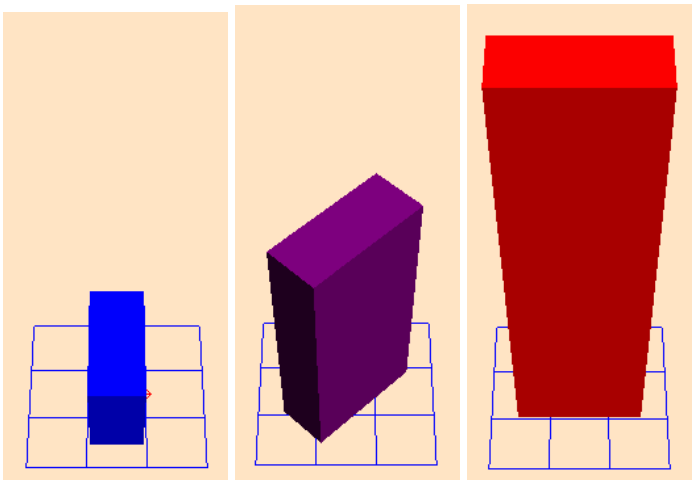


Figure 3.1 – Représentation des classes. Cette image montre des classes représentées à l'aide de VERSO. Les trois caractéristiques graphiques, hauteur, couleur et angle de rotation autour de l'axe des Y, vont en augmentant de la gauche vers la droite dans cet exemple.

L'utilisation de ces minima et maxima pratiques permet de représenter les éléments sans que la forme de certains d'entre eux ne dégénère. Ceci pourrait masquer les diffé-

rences qu'il y aurait entre les éléments se trouvant autour de la médiane. L'utilisation de ces limites pratiques est différente des valeurs seuils utilisées par les algorithmes automatiques. En effet, ramener les valeurs considérées comme extrêmement grandes ou extrêmement petites à des niveaux comparables permet toujours à l'analyste de comprendre que ces valeurs sont extrêmes et de prendre une décision en conséquence.

La hauteur a un minimum imposé pour ne pas voir la représentation graphique disparaître dans le plan ou être masquée trop facilement par les plus grandes classes. Le minimum perçu pour la couleur est un bleu pur (0,0,255) alors que le maximum est un rouge pur (255,0,0). L'interpolation entre les deux couleurs est linéaire, c'est-à-dire qu'à mesure que la métrique augmente, une valeur est soustraite au bleu alors qu'elle est ajoutée au rouge. L'angle de rotation varie entre 0° et 90° car dépassé ce stade, les angles peuvent être confondues avec l'angle correspondant dans le cadran opposé. Pour les métriques, celles étudiées ont toutes un minimum pratique fixé au même endroit que leur minimum théorique. La plupart de ces minima sont 0 et aucune métrique n'a un minimum théorique de moins l'infini. Il existe aussi des métriques possédant des maximums théoriques, mais pour les autres, l'analyste fournit la valeur maximale pratique. Cette valeur peut être calculée selon ses besoins ou estimée en fonction d'une plus grande distribution de données considérées représentatives du domaine.

3.1.3.1 Associations intuitives

Lors des différents essais avec notre approche, certaines associations se sont avérées meilleures que d'autres. En effet, le choix des associations peut influencer la perception de l'analyste des métriques et certaines associations sont plus intuitives que d'autres. Par exemple, il est bien connu que le couplage doit être évité le plus possible en génie logiciel. Une trop grande interaction entre les composants rend la maintenabilité plus complexes et augmente la propagation des problèmes d'un module à un autre. Le fait d'associer le dégradé de couleur du bleu vers le rouge à une métrique de couplage aide l'analyste à interpréter le logiciel. En effet, les zones où le rouge est prononcé indiquent les zones de danger. Le rouge étant associé au danger dans l'imaginaire collectif, ce choix de caractéristique s'avère judicieux. L'analyste peut alors faire un lien avec ce

qu'il connaît déjà pour interpréter les résultats. Ensuite, l'association entre la rotation et la cohésion est aussi un point de vue intéressant. La cohésion est le fait qu'une classe a une et une seule fonction. Lorsqu'un angle lui est affecté, on a l'impression que la classe dévie de son objectif ou de son but unique. Finalement, la taille des classes et la taille de leur représentation sont intuitivement reliées ensemble. C'est pourquoi faire correspondre des métriques de taille ou de complexité à la hauteur des boîtes est un meilleur choix.

3.1.3.2 Associations libres

Ceci étant dit, l'analyste peut toujours changer les associations à sa guise. En effet, toutes les métriques disponibles pour un logiciel sont emmagasinées dans un fichier XML. L'analyste peut donc associer n'importe quelle métrique se trouvant dans ce fichier avec n'importe quelle caractéristique graphique disponible. L'analyste peut donc se créer une série d'associations en fonction de ses besoins. La figure 3.2 montre bien le contrôle de l'analyste sur la visualisation. De plus, ce processus fait en sorte qu'il devient très facile d'introduire de nouvelles métriques, d'introduire de nouvelles caractéristiques de visualisation ou même de changer l'outil qui calcule les métriques sans affecter la sémantique de l'outil. Il est même possible d'utiliser ce type de visualisation avec des données qui ne proviennent pas de logiciels. Par exemple, on peut visualiser des caractéristiques de sports sans toucher à l'implantation de VERSO. La figure 3.3 en est un exemple.

3.2 Représentation des systèmes

Une fois que la représentation de chaque classe ou interface est bien définie, il reste à disposer ces éléments dans l'espace de façon cohérente. Le placement des entités aidera l'analyste à voir le programme aussi comme un tout et non comme un amas de classes sans lien entre elles.

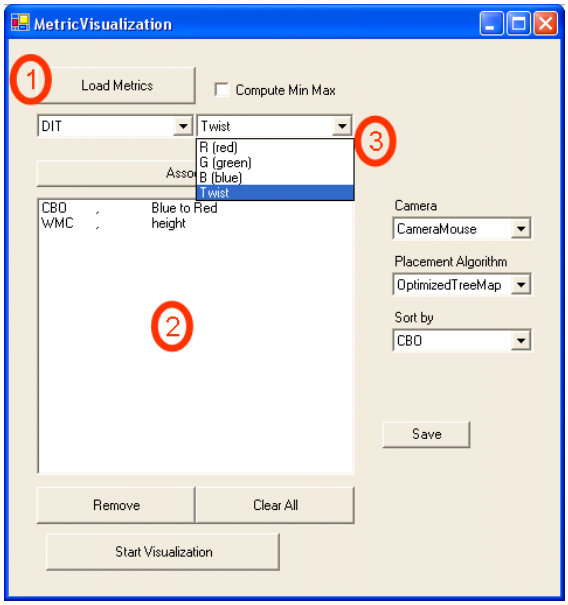


Figure 3.2 – Présentation de l'interface. 1) Chargement du fichier XML contenant les métriques. 2) Interface indiquant les associations en cours. 3) Interface de sélection des associations entre les métriques et les caractéristiques graphiques.

3.2.1 Principe

Dans le cadre de la visualisation de logiciel, le placement peut être utilisé de différentes façons. Certains ne font qu'aligner les éléments les uns à côté des autres [1, 46, 54], d'autres les placent de façon arbitraire pour que les classes soient bien visibles un peu à la manière d'UML [76], et d'autres encore représentent une information sur le logiciel à l'aide de la position dans l'espace [56, 69]. Le problème des deux premières techniques est que la position des éléments est perdu en tant qu'attribut graphique parce qu'elle ne représente aucune information. Pour ce qui est de la troisième technique, elle génère souvent des valeurs difficiles à lire et du chevauchement entre les éléments. Ce problème devient d'autant plus évident quand le placement est en trois dimensions.

La stratégie choisie dans le cadre de notre recherche est plutôt d'utiliser le placement pour représenter de l'information, mais pas des valeurs ou des métriques. Il est plus intuitif de voir le parallèle entre le placement et l'architecture du logiciel. Les classes appartenant aux mêmes paquetages sont donc regroupées pour former des amas distinguables.

Figure 3.3 – Résultats de Hockey. Cette figure représente les statistiques des joueurs pour la saison 2003-2004. Elle montre qu'il est possible de présenter toutes sortes d'informations tant qu'elles sont modélisées sous forme hiérarchique. Le temps de glace des joueurs est associé à la taille des boîtes, leur nombre de points est associé à la couleur des boîtes et finalement leur nombre de minutes de pénalité est associé à la rotation par rapport à l'axe des Y. Les équipes sont aussi séparées hiérarchiquement selon la conférence et la division.

De plus, VERSO n'utilise pas les trois dimensions entièrement. Il s'agit plutôt de 2D1/2 où les éléments représentés sont en trois dimensions, mais le plan qui les supporte est pour sa part en deux dimensions. Cette stratégie est un choix intéressant puisque le monde qui nous entoure est très près de cette situation alors que l'utilisation des trois dimensions ressemblerait plus à des objets flottant dans l'espace. Puisque notre support est planaire, on peut s'inspirer du domaine de la cartographie pour représenter l'architecture du logiciel. La géographie du logiciel étant inexistante, il n'y a pas de correspondance directe entre un logiciel et une carte. On peut par contre découper les paquetages et les sous-paquetages de façon telle que les séparations ressembleront à des frontières de régions, provinces, pays, continents etc., selon le niveau dans l'arbre hiérarchique. Les sections 3.2.2 et 3.2.3 présentent deux façons de diviser l'espace disponible.

3.2.2 *Treemap*

Le premier système utilisé pour représenter l'architecture des logiciels est le *Treemap* [35]. Cet algorithme, introduit par Johnson et Schneiderman en 1991, servait à l'époque principalement à la représentation des hiérarchies de fichiers. Dans ce temps, l'espace disque était plus dispendieux qu'aujourd'hui et on manquait souvent d'espace. L'analyse rapide des hiérarchies de fichiers était donc nécessaire. La principale force de cet algorithme est de pouvoir afficher des arbres en utilisant au maximum l'espace fourni par le support (souvent l'écran d'ordinateur). Les algorithmes traditionnels avec la représentation de la racine en haut et les enfants dessinés en largeur laisse beaucoup trop d'espace inutilisé et n'est donc pas efficace pour la représentation de hiérarchies en visualisation.

L'idée principale de l'algorithme est de prendre tout le rectangle de l'écran pour représenter la racine de la hiérarchie. Ensuite, pour représenter les enfants du premier niveau, le rectangle-racine est séparé en tranches verticales dont la largeur est proportionnelle à la taille de chaque enfant. Ensuite, chacun de ces enfants est séparé en tranches horizontales proportionnelles à la taille de ses propres enfants. Ce processus est appliqué récursivement en alternant la direction de la séparation entre horizontal et vertical jusqu'à ce qu'une feuille soit atteinte dans la hiérarchie. Le résultat est que tout l'espace est utilisé et que toute l'arborescence est représentée. On peut retrouver facilement les relations d'inclusion entre les éléments en observant la continuité des coupes. Par exemple, les coupures s'étalant d'un bout à l'autre de l'image sont celles du premier niveau alors que celles du deuxième niveau ont pour frontières les coupures à leur niveau parent et ainsi de suite. La figure 3.4 montre un exemple de l'algorithme traditionnel du *Treemap*.

Cet algorithme se prête bien à la représentation des logiciels, en particulier des logiciels *Java*, puisque des paquetages peuvent contenir d'autres paquetages ou des classes tout comme les répertoires peuvent contenir d'autres répertoires ou des fichiers. Par contre, l'algorithme de placement ne peut pas être utilisé comme tel. En effet, l'algorithme traditionnel du *Treemap* est capable de représenter des valeurs qui sont continues. L'algorithme doit donc être en mesure de couper les nœuds à n'importe quelle portion

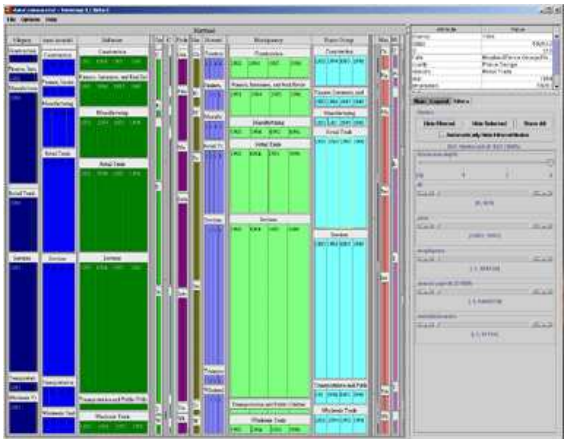


Figure 3.4 – Algorithme traditionnel du *Treemap* présenté à l'aide d'un outil commercial [72].

de l'image et pas nécessairement sur des portions entières. Il est impossible d'utiliser cette technique avec les classes de VERSO puisqu'elles doivent toutes occuper le même espace sur le plan pour éviter les biais dans la perception (voir la section 3.1.2.3). Il faut donc placer les éléments les uns à côté des autres dans les paquetages et faire la discrétisation des endroits où se trouvent les séparateurs.

La solution est d'arrondir l'espace nécessaire à l'entier supérieur pour permettre l'insertion de toutes les classes dans un paquetage. Bederson *et al.* [4] utilisent une technique similiaire pour représenter des items de tailles déterminées. Ils sont contraints dans l'espace de l'écran, mais peuvent par contre réajuster la dimension de leurs éléments à leur guise puisqu'ils travaillent avec des photos. Si la division n'est pas exacte, ce processus laissera de l'espace vide à l'intérieur des rectangles représentant les paquetages. Puisque de l'espace supplémentaire est ajouté presque à chaque fois, le rectangle représentant le parent du nœud doit lui aussi être augmenté. Ces petites augmentations de taille se répercutent jusqu'à la racine et accroissent donc la taille du rectangle initial. Ceci donne lieu à des rectangles finaux ayant des ratios différents selon le programme étudié. Par contre, les représentations restent assez carrées puisque la phase descendante de l'algorithme calcule l'espace nécessaire sans tenir compte des erreurs de discrétisation. Ces variations au niveau du rectangle initial sont sans conséquence puisque l'analyste se retrouve dans un univers en trois dimensions où il peut modifier son champ de vision.

Pour mieux comprendre la construction d'un *Treemap* à l'aide de l'algorithme modifié, la figure 3.5 présente la transformation d'un arbre représenté de la manière habituelle avec son équivalent représenté à l'aide de l'algorithme. Ensuite, la figure 3.6 présente un exemple des résultats obtenus à l'aide d'un vrai logiciel.

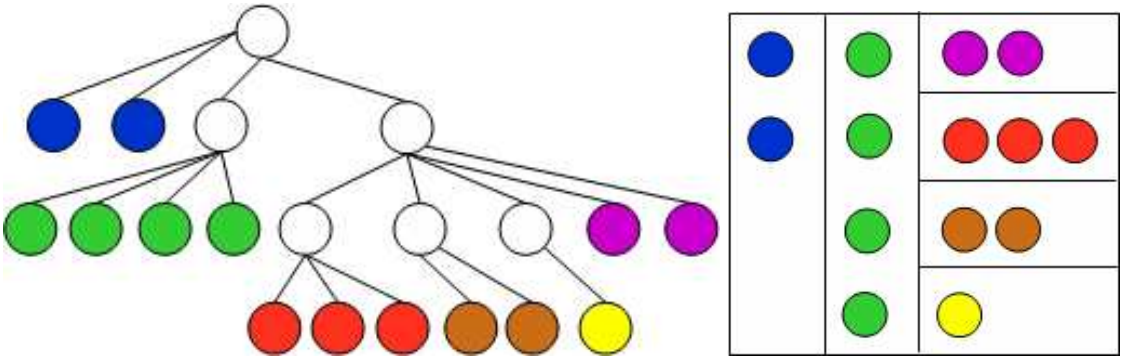


Figure 3.5 – Explication de l'algorithme modifié du *Treemap*. Sur la gauche, on voit un arbre représenté avec des nœuds et des arcs de la manière traditionnelle. Les feuilles sont en couleur et les autre nœuds sont blancs. Sur la droite, on voit l'équivalent de l'arbre de gauche aplani à l'aide de notre algorithme modifié du *Treemap*. Un paquetage virtuel est créé pour les classes quand leur parent contient aussi des paquetages.

3.2.3 *Sunburst*

La deuxième technique de placement étudiée est le *Sunburst* [74]. Tout comme le *Treemap*, l'algorithme original du *Sunburst* sert à représenter des hiérarchies de fichiers présents sur des disques. Cet algorithme est lui aussi très efficace pour représenter des programmes *Java* à cause des propriétés hiérarchiques.

Cette fois-ci, on exprime les hiérarchies à l'aide de dessins circulaires. On sépare les éléments à un même niveau en leur attribuant un angle proportionnel à leur taille. Par la suite, la distance depuis le centre du cercle est utilisée pour déterminer la profondeur dans l'arbre d'héritage. Au premier niveau, on sépare le cercle en attribuant un angle proportionnel à la taille de chaque enfant(sous-arbre) de la racine. Ensuite, on s'éloigne d'une unité du cercle, et chaque enfant de la racine distribue à ses propres enfants une partie proportionnelle à leur taille de l'angle qu'elle possède déjà. Le processus est récursif jusqu'à ce qu'une feuille soit atteinte. Il se peut que les feuilles soient à des niveaux

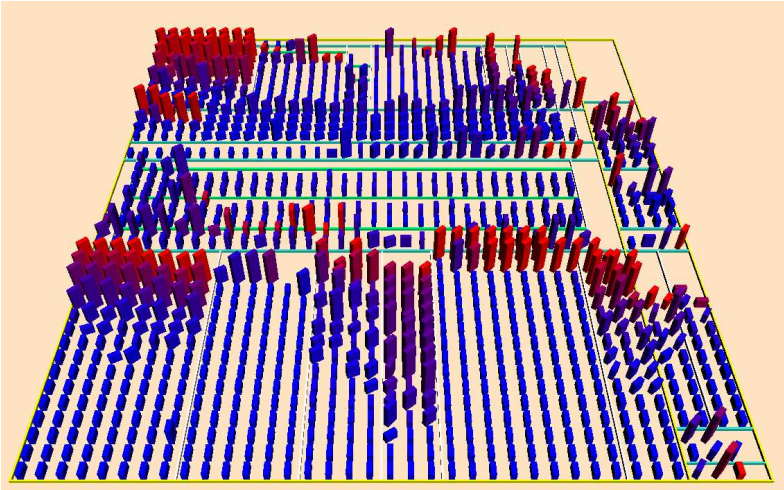


Figure 3.6 – Exemple de l'algorithme modifié du *Treemap*. Le logiciel *PCGen* servant à la création de personnages de jeux de rôle est représenté à l'aide de notre algorithme modifié du *Treemap*. Il compte 1129 classes. La couleur est ici associée au couplage, la hauteur est associée à la complexité de la classe et la cohésion est associée à l'orientation par rapport à l'axe des Y.

différents autour du cercle. La figure 3.7 montre un exemple de l'algorithme traditionnel du *Sunburst* appliqué sur un espace disque sous Unix.

Encore une fois, cet algorithme a été conçu pour représenter des valeurs continues qui peuvent être séparées de façon arbitraire. Bien entendu, ce n'est pas le cas des données que nous devons afficher. Dans ce cas, des portions d'arc sont réservées pour placer les représentations graphiques des éléments. Ces portions ont un arc intérieur égal à 1 et une épaisseur égale à 1 aussi. Lorsque l'espace manque sur une rangée, on passe tout simplement à la prochaine rangée jusqu'à ce qu'il n'y ait plus d'éléments à placer. Un séparateur en forme d'arc est ensuite placé et les éléments du prochain niveau sont traités. S'il y a de multiples enfants à traiter, l'angle du parent est séparé en parties proportionnelles et les classes sont ensuite placées. Il se peut que des rangées restent incomplètes et il se peut aussi que des rangées restent vides car leur angle ne permet pas d'accueillir aucun élément. Cette situation survient principalement très près du centre du cercle ou quand un angle très mince est attribué à un paquetage en raison de son faible nombre de classes.

Le cas où l'angle est trop petit engendre des problèmes dans certaines situations.

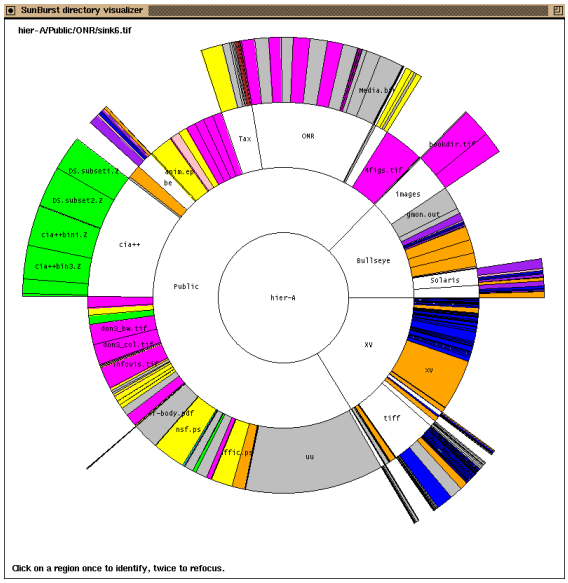


Figure 3.7 – Algorithme traditionnel du *Sunburst* [73]. Représente la hiérarchie de fichiers sur un disque sous Unix.

En effet, il arrive que certaines classes soient représentées très éloignées des autres à cause du petit angle. Ceci a pour effet d'empêcher l'analyste d'avoir une vue sur le programme où toutes les classes sont visibles. Pour palier à ce problème, les angles ne sont pas distribués équitablement entre les enfants à un niveau donné, mais on donne plutôt un angle minimal à chacun d'eux avant de distribuer la portion restante de façon proportionnelle. Cette action débalance quelque peu la représentation, mais évite certains cas extrêmes qui sont plus troublants pour l'analyste.

Pour mieux comprendre la construction d'un *Sunburst* à l'aide de l'algorithme modifié, la figure 3.8 présente la transformation d'un arbre représenté de la manière habituelle avec son équivalent représenté à l'aide de l'algorithme. Ensuite, la figure 3.9 montre un exemple de résultat obtenu à l'aide d'un vrai logiciel.

3.2.4 Trous dans les deux représentations

Comme il a été constaté dans les sections 3.2.2 et 3.2.3, les deux représentations affichent des valeurs qui sont discrètes et laissent donc de l'espace vide au niveau du placement. Le cas le plus simple des deux est le *Sunburst*. On ajoute des items dans la

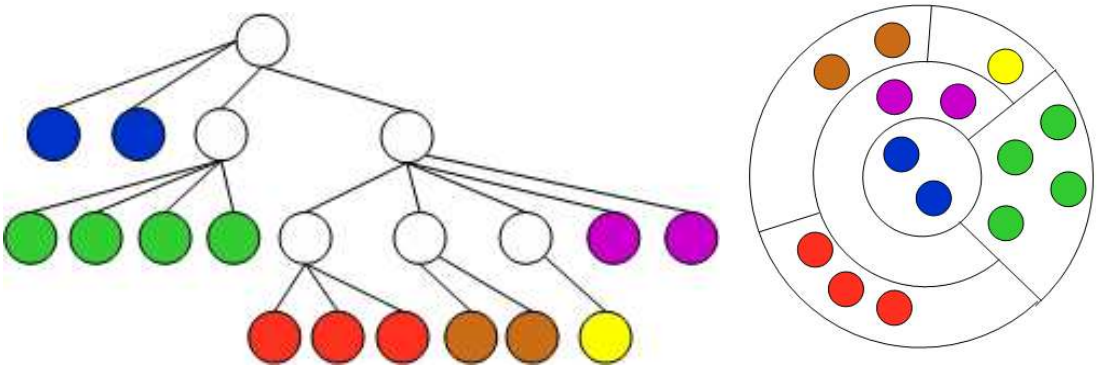


Figure 3.8 – Explication de l'algorithme modifié du *Sunburst*. Sur la gauche, on voit un arbre représenté avec des nœuds et des arcs de la manière traditionnelle. Les feuilles sont en couleur et les autre nœuds sont blancs. Sur la droite, on voit l'équivalent de l'arbre de gauche aplani à l'aide de notre algorithme modifié du *Sunburst*. L'éloignement par rapport au centre du cercle donne des indications sur la profondeur d'une classe dans l'arbre d'héritage.

portion de cercle qui nous est attribuée en augmentant la distance par rapport au centre du cercle à chaque fois. Il y a donc de l'espace négligeable qui est perdu au bout de chaque rangée parce que la division de l'arc de cercle n'arrive pas à l'entier près. Ensuite, il y a de l'espace perdu quand la dernière rangée n'est pas remplie. Ces trous sont difficilement évitables, mais sont souvent peu nombreux. Il serait possible de faire des zigzags avec les séparateurs de profondeur pour que le prochain niveau occupe l'espace laissé vacant par le niveau courant. Par contre, le jeu n'en vaut pas la chandelle, puisque l'esthétisme ainsi que la compréhension de l'analyste en seraient diminués. Il y a aussi les trous qui apparaissent en raison des arcs trop minces. La figure 3.10 montre comment sont formés les trous avec notre algorithme du *Sunburst* modifié.

L'apparition des trous à l'intérieur du *Treemap* est plus subtile, mais aussi plus sournoise. Le *Treemap* calcule l'espace nécessaire pour chacun des nœuds en allant de la racine jusqu'aux feuilles. On ne peut pas prédire le comportement des paquetages plus bas dans l'arbre architectural. Il se peut donc que l'espace nécessaire pour afficher les composants grandisse de façon arbitraire. Quand le nombre d'éléments à représenter n'a pas de facteur égal à l'espace alloué à ce dernier, la dernière rangée ou colonne ne sera pas remplie. Si plusieurs paquetages contenant des feuilles ont ce problème à un

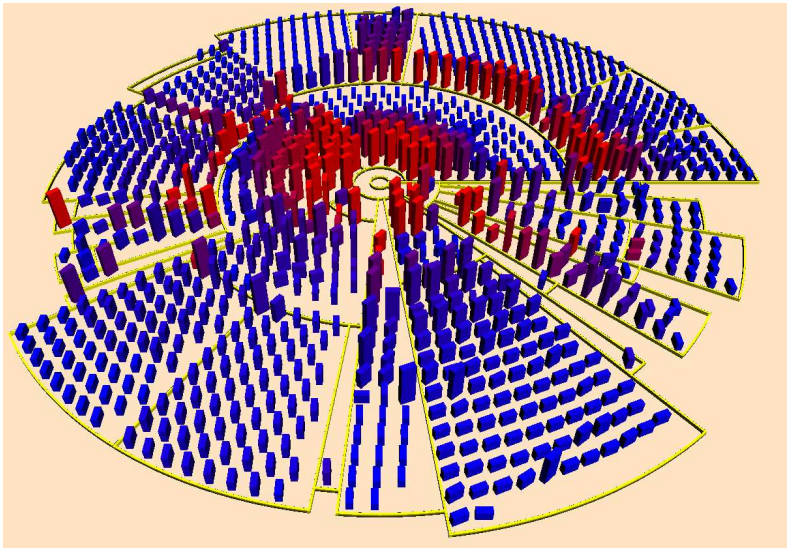


Figure 3.9 – Exemple de l'algorithme modifié du *Sunburst*. Le logiciel *PCGen* servant à la création de personnages de jeux de rôle est représenté à l'aide de notre algorithme modifié du *Sunburst*. Il compte 1129 classes. La couleur est ici associée au couplage, la hauteur est associée à la complexité de la classe et la cohésion est associée à l'orientation par rapport à l'axe des Y.

même niveau, on peut se retrouver avec plusieurs rangées ou colonnes supplémentaires. Ceci constitue la première façon dont les trous apparaissent. La deuxième façon survient quand pour deux paquetages au même niveau, le premier performe bien (un nombre d'éléments se divisant bien à chaque fois) et l'autre performe très mal (beaucoup de rangées incomplètes). Il y aura alors un grand trou formé au bout du paquetage se comportant bien, puisqu'il sera plus court que l'autre. Le parent ne peut que compléter l'espace avec le pire cas pour se juxtaposer au paquetage oncle (son voisin). Une troisième façon dont les trous peuvent être créés est quand deux paquetages placés à proximité les uns des autres, mais sont à des niveaux différents. Puisque les paquetages sont très grands, mais dans des directions différentes (horizontale par rapport à verticale), ils vont forcer de grands espaces vides dans des paquetages contenant peu de classes. La figure 3.10 montre la formation de trous à l'intérieur de notre algorithme du *Treemap* modifié alors que la figure 3.11 montre une des pires situations concernant les trous observée à l'aide de cet algorithme.

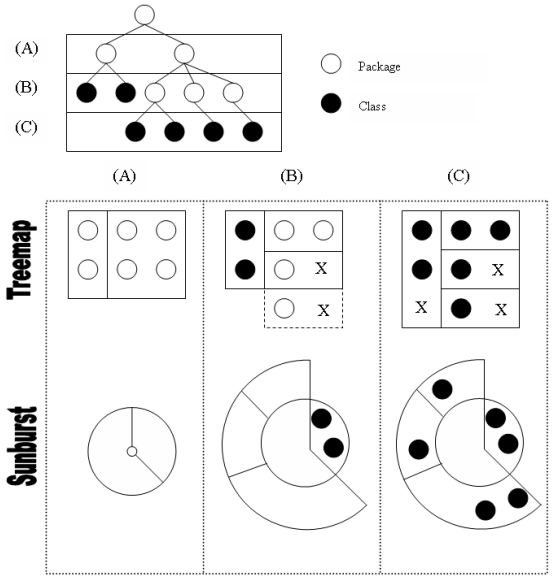


Figure 3.10 – Schéma représentant la création de trous. Ce schéma illustre comment les trous sont créés avec l'algorithme du *Treemap* (haut) et l'algorithme du *Sunburst* (bas). Le *Treemap* doit agrandir l'espace nécessaire en fonction des sépérations dans les niveaux inférieurs, ce qui amène la création de trous. Pour le *Sunburst*, c'est la non-complétude de rangée qui amène les trous.

Un algorithme a été développé afin de réduire le nombre de trous pour la représentation en *Treemap*. La technique, plutôt simple, essaie plusieurs possibilités et choisit celle qui génère le moins de trous. En descendant dans la hiérarchie, on impose une limite supérieure dans une seule direction à chaque fois. Si cette limite est impossible à atteindre, on l'augmente. Autrement, on sauvegarde le résultat et on réessaie avec une limite moins élevée. Ce processus se déroule en un temps exponentiel dans le pire des cas. Par contre, il trouve assurément les tailles des paquetages générant le moins de trous possible. Bien que l'algorithme semble lent, en sauvegardant des résultats redondants et en faisant une recherche dichotomique pour trouver la valeur qui résulte dans le moins de trous possible, le calcul peut se faire assez rapidement même pour les très grands systèmes (environ 7 secondes pour un système de plus de 5000 classes calculé sur une machine considérée standard au moment de l'écriture de ce mémoire). Il faut par contre noter que cet algorithme n'élimine pas les trous, mais réussit par contre à les réduire dans tous les systèmes. Il reste tout de même des cas où les trous sont toujours nom-

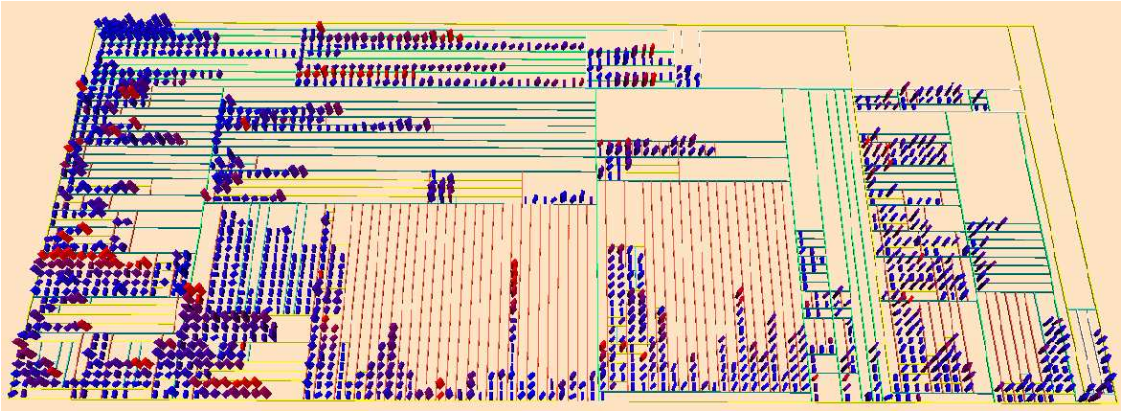


Figure 3.11 – Représentation où le nombre de trous est problématique. Dans le programme *Azureus* contenant 1422 classes, l'architecture fait que l'algorithme génère beaucoup de trous et de l'espace est ainsi perdu. Il s'agit d'un des pires cas observés concernant la prolifération de trous. Par contre la navigation et l'interprétation des données restent aisément pratiquables dans ce système.

breux, mais il est impossible de faire mieux avec des séparateurs suivant les normes du *Treemap*. Malgré tout, même si la réduction de trous est substantielle, l'utilisateur ne perçoit pas toujours de façon significative la différence entre les deux représentations. Il convient donc à l'analyste de déterminer si le temps de calcul supplémentaire vaut le coup pour ses observations.

3.2.5 Navigation

Notre approche permet aussi de naviguer de façon fluide dans l'espace en trois dimensions. Selon Healey et Enns [31], l'occlusion créée par les éléments en trois dimensions existe réellement, mais elle est facilement réglée en donnant accès à l'analyste à différents points de vue. La caméra peut être déplacée n'importe où sur la demi-sphère surplombant le plan. Le point d'intérêt (définissant la direction de vue) peut aussi être déplacé sur le plan XZ. Il permet en quelque sorte de déplacer le point central de la demi-sphère en permettant à l'analyste de se déplacer dans les 4 directions. Il ne sort jamais de ce plan pour éviter que l'analyste se perde dans l'univers en trois dimensions. Il est aussi possible de s'approcher et de s'éloigner du plan pour mieux voir certains détails ou encore avoir une meilleure vue d'ensemble. Tous les contrôles sont intuitifs et sont

activés à partir de la souris. Différents points de vue d'un même logiciel sont fournis à la figure 3.12.

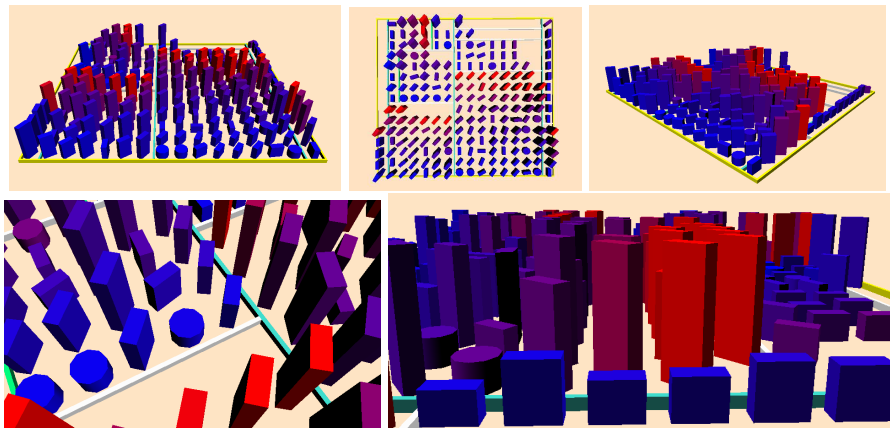


Figure 3.12 – Exemples de navigation. Différents points de vue d'un même logiciel illustrant les capacités de navigation de VERSO pour la visulisation des logiciels. La caméra est attachée à un point sur le plan que l'analyste peut modifier à sa guise. Elle peut se déplacer partout autour de la demi-sphère surplombant ce point.

3.3 Évaluation et discussion

Cette section explique comment VERSO se démarque en étant efficace et justifie sa raison d'être en donnant différentes utilisations possibles dans le domaine du génie logiciel.

3.3.1 Évaluation générale

Notre approche se distingue par plusieurs éléments. Dans un premier temps, elle permet la représentation de plusieurs milliers de classes simultanément. À titre d'exemple, la figure 3.13 représente un logiciel comportant plus de 5000 classes. Ceci est possible à la fois grâce à une bonne utilisation de l'espace, des représentations graphiques simples et peu encombrantes, ainsi qu'un mode de navigation efficace. Le point majeur contribuant à la représentation d'autant d'éléments est le renoncement à l'utilisation d'une forme de visualisation basée sur les graphes. En effet, selon Knight et Munro [39], cette

forme de représentation atteint rapidement ses limites à mesure que le nombre d'éléments à représenter grandit. Les liens entre les nœuds saturent trop rapidement l'espace disponible et cachent l'information pertinente. Une technique basée sur les filtres [41] a été développée dans un autre mémoire. Elle permet, sur demande, de visualiser certains liens entre les éléments sans que la vue des autres éléments ne soit obstruée.

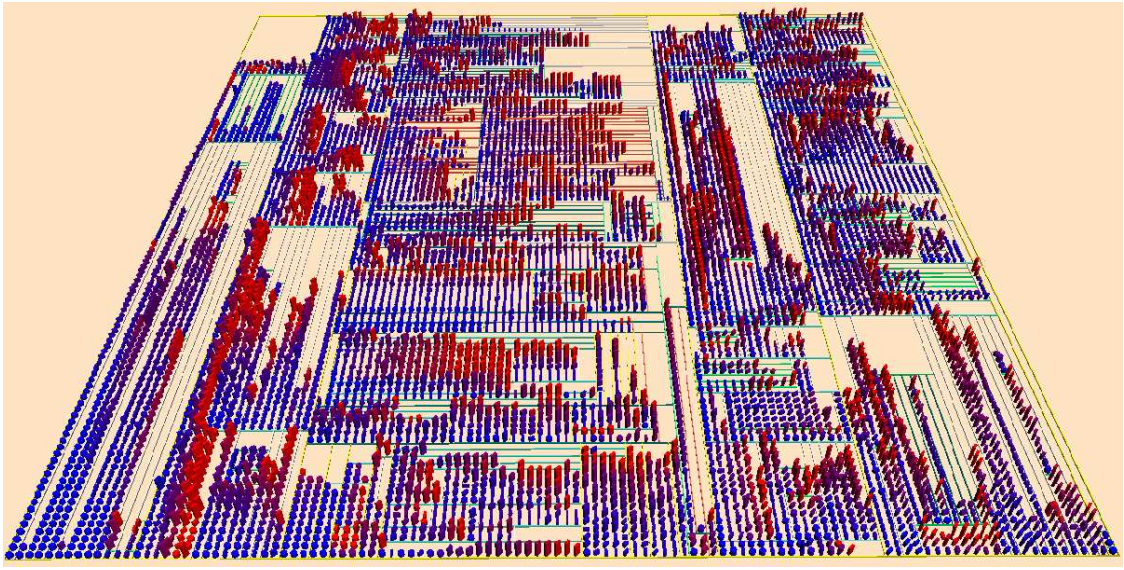


Figure 3.13 – *JDK1.5* représenté à l'aide du *Treemap*. Plus de 5000 classes sont affichées dans ce logiciel. À ce stade, la navigation reste fluide et il est toujours possible d'analyser les différentes parties du programme.

Ensuite, notre approche exploite bien le système visuel humain. Les caractéristiques utilisées sont facilement observables et peuvent être détectées depuis la vue d'ensemble d'un logiciel sans avoir à vérifier les classes une par une. La couleur a été démontrée comme étant une caractéristique très efficace dont la perception est instantanée dans les expérience de Healey [28]. La hauteur est aussi très efficace et le twist ne demande pas beaucoup plus d'efforts. Même avec une vue très éloignée des éléments, il est facile d'avoir une bonne idée de la composition d'un paquetage. Les formes choisies pour différentier les classes et les interfaces (boîtes et cylindres) permettent de bien classifier les deux représentations, mais ne viennent pas briser ou biaiser l'harmonie de la visualisation. L'homogénéité des deux éléments permet de regarder un programme sans subir de discontinuité cognitive pour analyser les deux entités. Ceci contribue à ne pas fausser la

perception plus générale d'un paquetage ou d'un système.

3.3.2 Applications possibles

Notre approche peut bénéficier à au moins deux grands groupes d'usagers : les développeurs et les chercheurs. Les développeurs peuvent l'utiliser pour évaluer leur code et le corriger au besoin. Ils peuvent aussi facilement vérifier la différence entre deux implantations pour savoir laquelle est meilleure en considérant la qualité. Les chercheurs pour leur part peuvent s'intéresser à notre approche pour comprendre et à terme expliquer certains phénomènes du génie logiciel. Avoir rapidement et facilement accès à une somme de données plus considérable leur permet de mieux comprendre le logiciel. L'objectif ultime est de trouver des lois régissant le logiciel un peu comme les lois régissant la physique. Sans vouloir aller aussi loin, il faut se rappeler que le génie logiciel est une science à caractère empirique. Tout outil permettant de défricher des concepts sur la qualité du logiciel sont donc les bienvenus.

Dans un premier temps, il est possible d'extraire très rapidement des principes généraux de qualité du logiciel. En effet, il existe des principes bien connus et acceptés par tous dans la communauté concernant la qualité et la maintenance du logiciel. Par exemple, il ne faut pas qu'un logiciel soit trop couplé, trop complexe ou manque trop de cohésion. Il s'agit alors seulement de montrer les métriques en question et de constater si certaines caractéristiques sont trop élevées.

Dans un deuxième temps, il est possible de détecter quelques antipatrons de conception [8] connus. Il s'agit de mauvaises solutions à des problèmes récurrents en informatique. L'équipe de VERSO a pu identifier plusieurs antipatrons et anomalies détectables à l'aide de notre outil de visualisation. Ils feront l'objet d'un mémoire futur. La liste comprend entre autres : *data classes/small classes, Blob/God class, Shotgun surgery, Misplaced class, God package, High coupling and low cohesion, Wide and shallow architecture*.

Finalement, il est possible de déterminer la nature ou la vocation des paquetages d'un logiciel seulement en ayant accès à leur représentation graphique. En effet, les paquetages ayant une vocation particulière incluent des classes avec des métriques caractéris-

tiques de leur condition. Par exemple, les paquetages très importants, dont le *«cœur»* du programme, seront souvent complexes et très couplés. Par contre, les paquetages contenant seulement des types de données seront très peu complexes et peu couplés. Entre les deux, il est possible de retrouver des paquetages dont le mandat est de donner des services pour aider les autres modules à accomplir certaines tâches. Les programmeurs nomment souvent ce genre de paquetages *util*. Les classes de ces paquetages ont habituellement une grande taille et un couplage moyen (elles apparaissent donc de couleur violette). En un clin d'œil, il est donc possible d'en savoir beaucoup sur l'architecture d'un programme, ou encore de savoir si l'architecture ne répond pas aux normes traditionnelles de construction des modules. Un exemple de différents paquetages ayant différentes utilités est présenté à la figure 3.14.

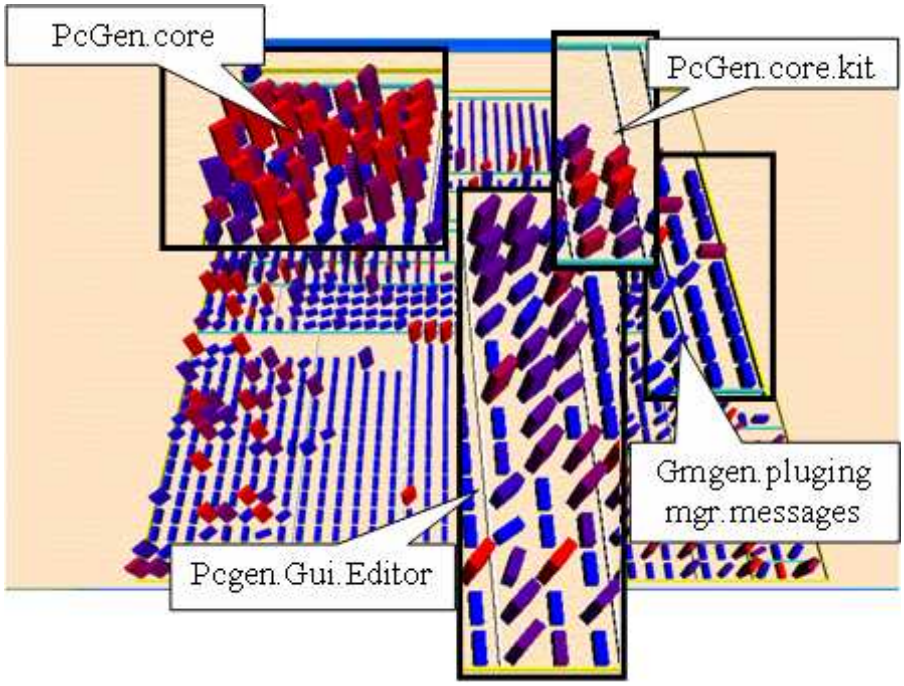


Figure 3.14 – Comprendre l'architecture. Seulement en utilisant la représentation graphique d'un programme, il est possible d'avoir une bonne idée sur la vocation de ses différents paquetages. Cette image est une représentation de *PCGEN*, un logiciel servant à la création de personnages pour des jeux de rôle. Les paquetages contenant plusieurs classes complexes et couplées ont tendance à être névralgiques dans le système comme c'est le cas de *PCGen.core*. D'autres paquetages sont destinés à la modélisation de données et contiennent des classes peu complexes comme c'est le cas pour *Gmgen.plugingmgr.messages*. Il existe aussi des paquetages de type utilitaire qui se retrouvent entre ces deux extrêmes.

CHAPITRE 4

ÉVALUATION DES ALGORITHMES DE PLACEMENT

Dans le but d'évaluer l'efficacité de notre approche par visualisation, nous avons mené une expérience à l'aide de sujets. Les observations et les données receuillies sont présentées ici pour valider les choix mentionnés dans le chapitre 3. Même si la technique de visualisation décrite est basée sur des principes rigoureux, la confirmation par des résultats expérimentaux montrent l'efficacité de notre approche dans des cas réels et concrets.

4.1 Objectif

L'expérience a pour but de comparer le *Treemap* et le *Sunburst* en tant qu'algorithmes de placement à l'intérieur de notre approche. On veut aussi savoir si les algorithmes de placement plus sophistiqués sont plus efficaces que les algorithmes de placement simples. Les résultats permettront d'identifier quelle stratégie de placement est la meilleure pour un contexte spécifique et permettront d'évaluer l'influence du placement sur la réalisation de certaines tâches reliées à l'étude de la qualité.

4.2 Hypothèses

Les différentes hypothèses de l'expérience sont les suivantes :

– *Hypothèse 1* :
  Des algorithmes plus sophistiqués diminuent le temps nécessaire pour effectuer des tâches d'analyse de la qualité.
– *Hypothèse 2* :
  L'algorithme du *Treemap* performe mieux pour effectuer des tâches d'analyse de la qualité.
– *Hypothèse 3* :
  L'algorithme du *Sunburst* performe mieux pour effectuer des tâches d'analyse de

la qualité.

– *Hypothèse 4* :
  L'algorithme du *Treemap* performe mieux pour des tâches d'analyse faisant appel aux informations quantitatives.
– *Hypothèse 5* :
  L'algorithme du *Sunburst* performe mieux pour des tâches d'analyse faisant appel à l'architecture (hiérarchie de paquetage).
– *Hypothèse 6* :
  Le temps nécessaire pour effectuer des tâches faisant appel à la qualité avec notre démarche est significativement inférieur au temps nécessaire pour effectuer ces mêmes tâches manuellement.

4.3 Protocole

L'expérience consiste en deux volets. Le premier est un questionnaire électronique alors que le second consiste en différentes questions subjectives. Ces deux volets seront décrits dans cette section.

4.3.1 Questionnaire électronique

Le questionnaire électronique comporte 20 tâches portant sur des logiciels différents. Les tâches sont toutes reliées à la qualité du logiciel. Il y a deux types de tâches : les tâches d'analyse impliquant les valeurs des métriques et les tâches d'analyse impliquant l'architecture des programmes. On peut trouver 5 questions du premier type, 5 questions du deuxième type et 10 questions touchant aux deux types. La totalité des questions est présentée dans l'annexe I.

4.3.1.1 Automatisation

La réponse à une question est toujours la même. Il s'agit de sélectionner (par un clic) la classe recherchée ou encore de sélectionner une des classes contenues dans le paquetage recherché. Si la réponse est juste, le programme quitte la visualisation courante et

passe automatiquement à la prochaine question. Si la réponse est fausse, le programme reste dans la visualisation courante jusqu'à ce qu'une réponse juste soit donnée. Toutes les tâches de chaque sujet sont chronométrées. Le chronomètre démarre quand le sujet a terminé de lire la question et qu'il appuie sur le bouton permettant de répondre à la question. La fenêtre de visualisation apparaît au même moment. Le chronomètre s'arrête uniquement quand la réponse juste est donnée. Les mauvaises réponses ne sont pas prises en compte. Le sujet peut revenir lire la question s'il le désire après avoir démarré la visualisation, mais le temps continue de s'écouler quand même. Chaque entrée de temps est conservée en mémoire à mesure que le sujet progresse dans l'expérience. La figure 4.1 montre l'interface permettant aux sujets de démarrer la question.

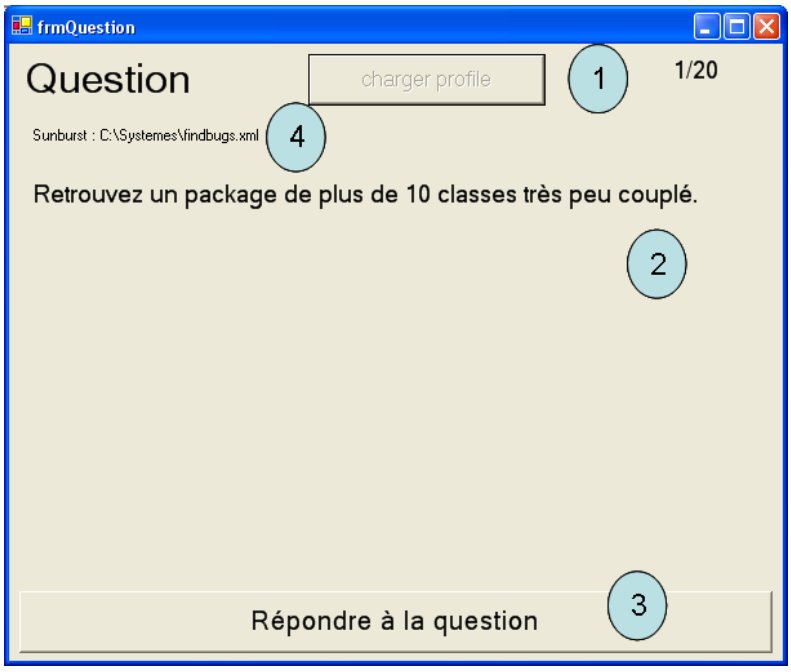


Figure 4.1 – Questionnaire électronique de l'expérience. Cette interface permet aux sujets de répondre aux différentes questions. 1) Chargement d'un des trois profiles. 2) Question. 3) Bouton démarrant la visualisation et le chronomètre. 4) Indication sur le mode de placement utilisé et sur le système visualisé.

4.3.1.2 Algorithmes de placement étudiés

En tout, il y a trois algorithmes : le *Treemap*, le *Sunburst* et le *Treeline*. Ce dernier est un algorithme de placement naïf qui est à la fois facile à implanter et facile à comprendre. Il s'agit d'afficher les classes les unes à la suite des autres sur une ligne. On change tout simplement de ligne quand il n'y a plus d'espace sur la ligne courante. On part du bas à gauche en plaçant les éléments de la gauche vers la droite sur une ligne, et on change de ligne en déplaçant vers le haut. On place tout d'abord les éléments qui sont les plus hauts dans la hiérarchie des paquetages et on descend dans l'arbre à l'aide de algorithme de fouille en profondeur. Une fois ces éléments placés, on utilise un séparateur pour montrer qu'un paquetage change de niveau. Ensuite, si le séparateur change de couleur, cela indique qu'on descend d'un niveau. Si le séparateur est de la même couleur que son prédécesseur, nous sommes en présence d'un paquetage au même niveau que le précédent. Si le séparateur revient à une couleur précédente, c'est qu'on remonte d'un niveau. Cet algorithme a l'avantage d'utiliser au maximum l'espace qui lui est disponible. Un exemple du placement de cet algorithme est disponible à la figure 4.2.

4.3.1.3 Attributions des tâches aux sujets

Les sujets ont été séparés en trois groupes de façon aléatoire. Chaque groupe se voyait attribué un algorithme de placement différent pour une même question. Évidemment, il est impossible de reposer la même question à un même sujet pour un algorithme différent. Le temps nécessaire pour trouver la réponse aurait été faussé par ses connaissances préalables. De plus, pour éviter les biais dus à la fatigue ou à l'apprentissage durant l'expérience, les questions sont présentées dans un ordre aléatoire, différent pour chacun des sujets.

4.3.2 Questions subjectives

Le deuxième volet de l'expérience consiste en deux questions subjectives demandées aux sujets. Les questions posées sont les suivantes :

– Quel système préférez-vous ?

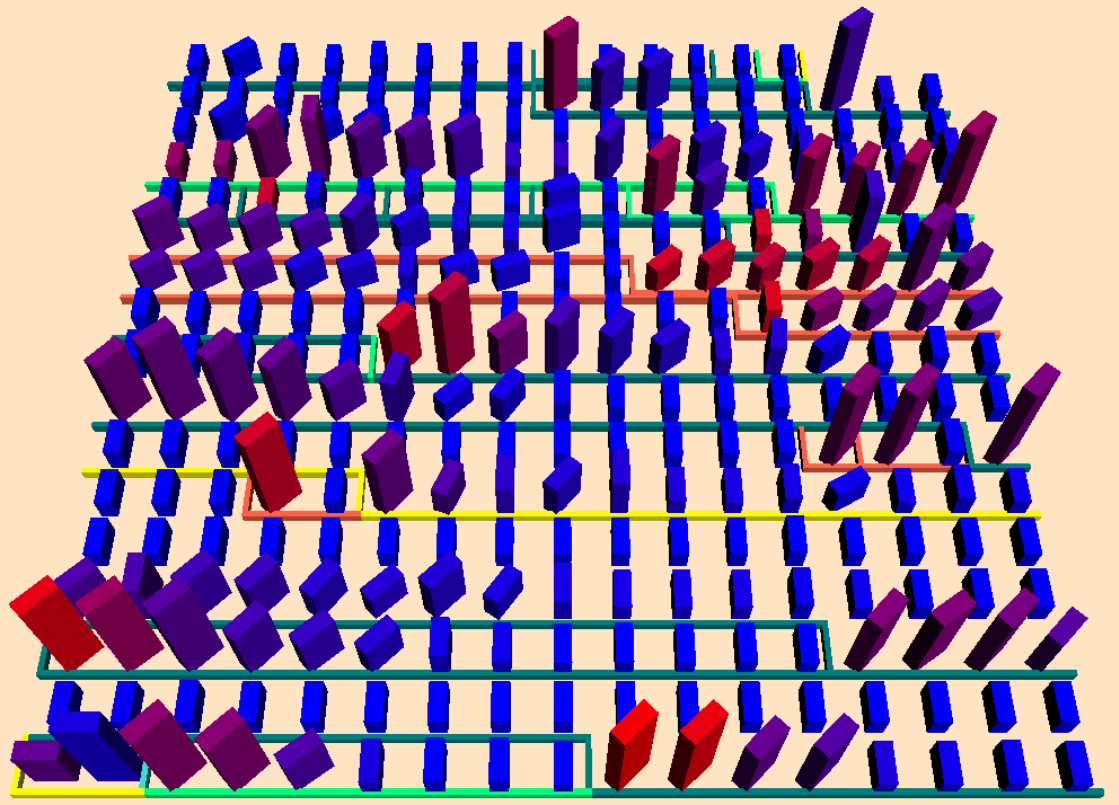


Figure 4.2 – Exemple de l'algorithme du *Treeline*. Démonstration de l'algorithme du *Treeline* sur le logiciel *Emma* (286 classes). Chacune des classes est placée en rangée et la couleur des séparateurs indique le niveau des classes dessinées après celui-ci.

– Quel système ressemble le plus à une ville selon vous ?

Les sujets sont appelés à répondre oralement à ces questions lorsqu'ils ont terminé les 20 questions automatiques. On leur demande aussi de fournir leurs commentaires sur l'expérience et sur le prototype de visualisation.

4.3.3 Sujets

15 sujets ont participé à l'expérience. Les sujets étaient principalement des étudiants gradués œuvrant dans le domaine du génie logiciel. Ils avaient une bonne connaissance des logiciels orientés objets. Les sujets étaient volontaires pour l'expérience donc leur intérêt pour celle-ci ne présente pas un biais. L'expérience a toujours eu lieu sur le même ordinateur et le même écran et avec des conditions ambiantes similaires. Les sujets

avaient tous une vue normale ou corrigée à l'aide de lentilles.

Les sujets suivaient tous une courte formation dont la durée était environ de 15 minutes. Elle leur permettait de se familiariser avec les concepts de VERSO et ses contrôles. Ils apprenaient entre autres comment se déplacer dans l'environnement, les correspondances entre les métriques et les caractéristiques graphiques ainsi que l'interprétation de l'architecture dans les trois différents types de placement.

4.4 Résultats

Les 15 sujets ont réussi à répondre à toutes les questions dans un temps raisonnable et ont tous apprécié leur utilisation du prototype basé sur l'approche présentée au chapitre 3. Les 15 sujets multipliés par les 20 questions posées à chacun donne un total de 300 entrées chronométrées. Les résultats des temps obtenus sont présentés au tableau 4.1. Il s'agit des moyennes globales des algorithmes pour toutes questions confondues. Ensuite nous présentons différents tests statistiques permettant de voir s'il existe une différence significative entre les temps obtenus pour les différents algorithmes. Les significations des différences obtenues pour les test-t et test de Wilcoxon (normalité non-vérifiée) sont présentées au tableau 4.2. Le tableau 4.3 montre les mêmes informations où la moyenne de chaque sujet pour une question et un alogrithme donné sont considérés pour faire la différenciation entre les algorithmes. De plus, le tableau 4.4 montre la signification de la différenciation pour les questions faisant appel aux informations quantitatives uniquement et pour les questions faisant appel à l'architecture uniquement. Les résultats de toutes les entrées enregistrées pour l'expérience sont contenus dans l'annexe II.

	Sunburst	Treemap	Treeline
Temps Moyen (sec.)	50.13	36.38	69.88
Temps Médian (sec.)	23.87	22.24	41.24
Meilleur Temps	35%	55%	10%
Système préféré	80%	13.33%	6.66%

Tableau 4.1 – Résultats de l'expérience. L'indication «Meilleur Temps» indique le pourcentage de questions pour lesquelles l'algorithme a eu la meilleure moyenne.

	Test-t	Wilcoxon
Sunburst-Treemap	0.107	0.449
Sunburst-Treeline	0.020	0.001
Treemap-Treeline	0.000	0.000

Tableau 4.2 – Différenciation des algrorithmes. Signification des différences de chacun des algorithmes considérés par paires.

	Test-t	Wilcoxon
Sunburst-Treemap	0.197	0.179
Sunburst-Treeline	0.054	0.037
Treemap-Treeline	0.006	0.002

Tableau 4.3 – Différenciation des algrorithmes par question. Signification des différences de chacun des algorithmes considérés par paires où la moyenne des sujets de chaque question est considérée pour éavaluer la différence.

Les sujets ont aussi émis quelques suggestions concernant les améliorations à apporter à VERSO. On parle entre autres d'une navigation améliorée et d'un affichage des noms des éléments plus interactifs. Il faut dire qu'au moment de l'expérience, la navigation à l'aide de la souris n'avait pas encore été implantée dans le prototype et qu'elle se faisait à l'aide des touches du clavier.

4.5 Discussion

Cette section est consacrée à la discussion sur les résultats obtenus dans l'expérience. Elle donne des indications sur ce qui a pu être montré et expose aussi les facteurs qui doivent être corrigés dans une future expérience.

4.5.1 Hypothèses

Nous passerons en revue ici toutes les hypothèses présentées à la section 4.2.

4.5.1.1 Hypothèse 1

L'hypothèse 1 est clairement vérifiée car les temps moyens et médians pour le *Treemap* et le *Sunburst* sont nettement inférieurs aux temps mesurés à partir du placement

	Test-t	Wilcoxon
Sunburst-Treemap (info quantitatives)	0.147	0.677
Sunburst-Treemap (architecture)	0.664	0.192

Tableau 4.4 – Différenciation des algorithmes par type de question. Signification des différences pour les questions faisant appel aux informations quantitatives et pour les questions faisant appel à l'architecture.

naïf du *Treeline*. La complexité influence donc de façon positive le temps de réponse d'un sujet dans les tâches proposées durant l'expérience. On peut remarquer que le *Treeline* obtient la meilleure moyenne seulement pour 2 questions sur les 20 au total. Ceci peut être expliqué par le fait que la réponse à la question se trouvait sous les yeux des sujets, en plein milieu de la représentation, alors que ce n'était pas le cas avec les deux autres types de placement. Les tests de Student et de Wilcoxon valident tout deux cette hypothèse et sont significatifs pour un niveau de confiance de 90%. L'hypothèse est donc vérifiée peu importe si les temps sont distribués selon la loi normale ou non.

4.5.1.2 Hypothèses 2 et 3

Les hypothèses 2 et 3 doivent être rejetées toutes les deux. Il est très difficile de déterminer un gagnant entre le *Treemap* et le *Sunburst* même si le *Treemap* semble avoir un petit avantage. Le *Sunburst* s'est vu accorder la meilleure moyenne dans 35% des cas alors que le *Treemap* a eu le dessus dans 55% des cas. L'un peut être meilleur que l'autre dépendamment de la tâche effectuée. On voit aussi qu'il y a peu de différence entre les temps médians des deux représentations. De plus, les tests de Student et de Wilcoxon ne sont pas vérifiés pour un niveau de confiance de 90%. Les deux hypothèses sont donc rejetées et l'expérience ne détermine pas de vainqueur entre les deux algorithmes. Il serait avantageux de refaire une expérience dans de meilleures conditions pour décerner un vainqueur.

4.5.1.3 Hypothèses 4 et 5

Ces hypothèses visaient à déterminer si un algorithme convenait mieux à certain type de question et vice-versa. Malheureusement, ces deux hypothèses doivent être re-

jetées parce que la différence entre le *Treemap* et le *Sunburst* n'est pas statistiquement significative entre les deux algorithmes (niveau de confiance inférieur à 90%). Pour les 5 questions concernant les caractéristiques ou les métriques des classes, par 3 fois le *Treemap* a eu le dessus alors que ce fut le cas 2 fois pour le *Sunburst*. Pour les questions concernant l'architecture, la conclusion n'est pas aussi radicale. En prenant les 5 questions individuellement, autant pour la moyenne que pour la médiane, le *Sunburst* donne des temps inférieurs au *Treemap*. De plus, les moyennes et médianes globales des temps des 5 questions réunies sont supérieures pour le *Treemap*. Il est toutefois impossible de conclure que cette différence est significative en raison de l'écart-type des deux distributions. Le niveau de confiance inférieur à 90% confirme l'inexistence de la supériorité du *Sunburst* pour les tâches d'analyse reliées à l'architecutre. Par contre, il serait très intéressant de mener une nouvelle expérience où la variabilité entre les sujets serait diminuée.

4.5.1.4 Hypothèse 6

L'hypothèse 6 est clairement vérifiée avec des moyennes sous la minute et des temps médians frôlant la demi-minute. Certaines tâches sont plus longues et d'autres sont plus courtes, mais les temps se tiennent pratiquement toujours sous la barre des deux minutes et les moyennes et médianes sont clairement sous la barre de la minute pour les types de placement principaux. Il est à noter que ces mêmes tâches prendraient beaucoup plus de temps à effectuer s'il fallait inspecter le code ou tout simplement retrouver les éléments à l'intérieur d'un tableau décrivant toutes les métriques. Par exemple il a fallu 10 secondes en moyenne pour identifier un paquetage cohésif et contenant très peu de classes dans le système *JDK 1.1* contenant 1662 classes. Certaines tâches très faciles peuvent toujours être effectuées dans des temps similaires avec des fonctions de recherche dans des tables, par contre la très grande majorité des tâches s'effectuent beaucoup plus rapidement que de façon manuelle. Une expérience impliquant différents systèmes de visualisation et incluant aussi une approche manuelle doit être conduite pour confirmer l'efficacité de notre approche toute entière. Ces travaux sont par contre laissés pour le futur.

4.5.2 Discussion générale

Un des résultats les plus intéressants présentés plus haut est la divergence entre la réponse des sujets face à la question leur demandant leur préférence et les temps enregistrés pour le *Treemap* et le *Sunburst*. En effet, les sujets semblent dans une large majorité préférer le *Sunburst*, alors que les questions automatiques privilégient le *Treemap*. Cette divergence est difficile à expliquer et est probablement due au fait que les gens trouvent le *Sunburst* plus esthétique et plus harmonieux. Il est aussi possible que les gens préfèrent le *Sunburst* parce qu'ils ont un point où débuter leur recherche, c'est-à-dire le milieu du cercle. Ceci peut être réconfortant, car leur regard peut se rattacher à quelque chose qu'ils connaissent et comprennent avant d'explorer le reste du système.

De plus, en observant les résultats de l'annexe II on peut remarquer de très grandes divergences dans les temps de réponse et ce même au sein d'un même groupe. Ce phénomène est expliqué par le fait qu'il y avait beaucoup de divergences entre les sujets au point de vue de l'expérience à l'intérieur d'un système en trois dimensions. En effet, les personnes ayant déjà été en contact avec des univers en trois dimensions et le contrôle de la caméra qui s'y rattache ont des temps très inférieurs à ceux qui n'ont pas l'habitude des environnements artificiels en trois dimensions. Souvent, il s'agit d'expérience acquise à travers les jeux vidéo. Un autre facteur important est la témérité. En effet ceux qui n'hésitaient pas à faire quelques essais-erreurs avant de trouver la bonne réponse obtenaient de meilleurs résultats au niveau du temps de réponse que ceux qui tentaient de revérifier leur réponse plusieurs fois avant de cliquer sur une classe. Par contre, ces deux phénomènes n'ont pas été une source importante de biais dans l'expérience. Les sujets ayant pris plus de temps pour répondre aux questions à cause de ces raisons l'ont fait pour toutes les questions et donc pour tous les algorithmes. Les comparaisons entre les temps moyens et médians gardent donc leur sens. Il faudrait par contre travailler à réduire ces différences entre les sujets dans les futures expériences.

La tendance montre que le *Treemap* est une manière efficace de placer les données. Le *Sunburst* peut s'avérer intéressant pour certaines tâches, mais n'est jamais beaucoup supérieur au *Treemap*. Nous avons tout de même choisi de poursuivre le développement

du logiciel en nous concentrant sur le *Treemap*. La raison principale est due au fait que la rotation est plus difficile à percevoir avec le *Sunburst*. En effet, les séparateurs n'étant pas alignés sur les axes, ils peuvent induire l'analyste en erreur quand vient le temps d'évaluer rapidement le degré de rotation d'une classe. Il vaut par contre la peine de continuer à investiguer des placements plus étoffés, car l'étude montre que le placement a une influence sur la compréhension d'un système.

CHAPITRE 5

VISUALISATION DE L'ÉVOLUTION

Pour bien comprendre le logiciel et réussir à interpréter sa qualité, il est important de comprendre son évolution [12]. En effet, des indicateurs de qualité observables à une version donnée sont des manifestations d'erreurs passées, mais n'identifient pas la cause du problème et encore moins sa solution. L'évolution permet aussi de voir le cheminement parcouru par les développeurs durant la construction du logiciel et ainsi de mieux comprendre son processus.

5.1 Analyse de l'évolution

Un problème évident avec l'évolution et sa visualisation est l'explosion de données et la représentation de cette quantité de données gigantesque. En effet, toutes les données présentes dans la visualisation d'une seule version sont multipliées par le nombre de versions à visualiser. Il est possible de compresser les données ou d'utiliser d'autres techniques pour ne pas avoir à les représenter en même temps. Certaines recherches [13, 56] présentent même des mesures sur le phénomène de l'évolution au lieu de représenter les changements. Cette solution est intéressante pour certaines tâches d'analyse, mais elle ne sera pas explorée dans ce mémoire.

Ce qui sera par contre exploré, c'est l'utilisation de l'animation pour représenter les changements de versions. Le principe est d'utiliser une forme de ligne du temps imaginaire représentant les versions et correspondant à des états graphiques. À partir de cette dimension temporelle, l'analyste peut contrôler le temps en choisissant la version qu'il veut visualiser et le moment qu'il passe sur chacune des versions. Selon Rilling et Mudur [68], il est important d'instaurer le sentiment d'immersion chez l'utilisateur. Dans leur artcile, leur propos a surtout trait à la navigation (on parle de fluidité et de détection de collisions entre la caméra et les objets par exemple), mais il est normal de penser que ce principe s'applique aussi à la notion du temps à l'intérieur de la visuali-

sation. Différentes stratégies servant à visualiser l'évolution ont été développées durant les recherches explicitées dans ce mémoire. La section 5.2 décrit l'utilisation d'un algorithme se servant d'une seule image fixe pour représenter toutes les versions d'un logiciel. La section 5.3 présente comment l'animation permet de représenter plus d'informations, mais à des moments différents. Diverses stratégies sont alors étudiées pour représenter l'évolution des logiciels (elles sont décrites dans les sous-sections suivantes). La section 5.3.1 présente une solution impliquant plusieurs images, mais où les classes demeurent toujours au même endroit tout au long de la visualisation. La section 5.3.2 montre l'utilisation d'images intermédiaires pour attirer le regard de l'analyste sur les changements et l'aider à départir les entités qui ont été modifiées de celles qui ne l'ont pas été. La section 5.3.3 présente un algorithme qui réarrange la position des classes selon celles qui sont présentes à une version précise. Finalement, la section 5.3.4 présente deux approches intermédiaires, alliant une meilleure utilisation de l'espace et un mouvement réduit pour le placement des entités.

Ces stratégies seront ensuite comparées selon des critères subjectifs pour déterminer leur qualité. Le premier critère est la quantité d'espace dans l'univers en trois dimensions utilisée par l'algorithme. Plus l'algorithme requiert un grand espace, et ce même quand le nombre d'éléments à représenter est restreint, plus il sera difficile pour l'analyste d'interpréter un grand nombre de données à la fois. Ensuite on évaluera si la cohérence temporelle est bonne. Par ce terme, on entend la facilité avec laquelle les analystes pourront suivre les modifications d'un logiciel, mais surtout la facilité avec laquelle ils comprendront quelles parties restent stables pour leur permettre de concentrer leurs efforts ailleurs. Cette forme de cohérence temporelle est surtout calculée en fonction de la facilité de l'analyste à retrouver des éléments de mêmes noms d'une version à l'autre. Ensuite, on évaluera la facilité avec laquelle les changements dans les métriques sont perçus. Et finalement, on évaluera s'il est toujours facile de comprendre et d'interpréter l'architecture d'un programme à l'aide des séparateurs.

5.2 Image unique

Plusieurs recherches [13, 45, 56, 63, 78] utilisent une image unique pour représenter toutes les versions d'un logiciel. Ce type de représentation a l'avantage de permettre une visualisation instantanée de toutes les informations disponibles. Toutefois, le défi devient de plus en plus grand à mesure que le nombre d'éléments à représenter augmente. Il devient donc pratiquement impossible de présenter autant d'informations par éléments qu'il est possible de le faire avec la visualisation de versions uniques (chapitre 3). Il faut alors compresser les données ou accepter que l'analyste prenne plus de temps pour observer les données en raison de la saturation d'informations.

La figure 5.1 présente les résultats obtenus avec cette stratégie d'évolution dans le cadre de VERSO. Les versions apparaissent les unes à côté des autres dans l'ordre chronologique. Les versions sont séparées comme si elles étaient des paquetages différents. Pour éviter la confusion, on utilise des doubles séparateurs entre les versions. Pour plus de simplicité, les éléments conservent la même position dans toutes les versions, même si des trous apparaissent dans la visualisation. Ceci rend plus facile le repérage des éléments évoluant d'une version à une autre. Ces images donnent souvent lieu à des représentations très larges compte tenu du nombre de séparations horizontales qui ne peuvent pas être réduites. Le nombre de séparations verticales multiplié par le nombre de versions empêchent toute optimisation horizontale.

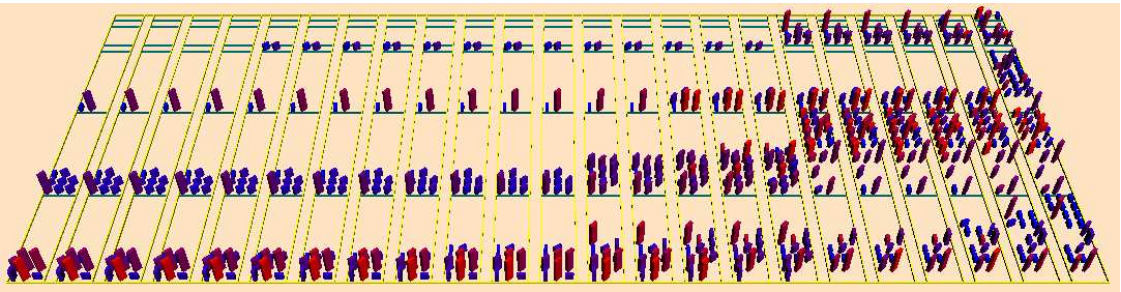


Figure 5.1 – Représentation de l'évolution à image unique. Cette représentation comporte 23 versions du logiciel *Quantum*, un plugiciel d'*Eclipse* permettant l'accès à des bases de données. Les versions sont séparées par des doubles lignes jaunes et les classes conservent la même position relative dans chacune des versions pour aider l'analyste à les retrouver.

Du point de vue de l'utilisation d'espace, cette solution n'est pas très bonne puisque l'écran est vite saturé d'éléments et il faut souvent naviguer dans la visualisation afin de rejoindre les versions aux extrémités. Ensuite, même si les classes restent toujours au même endroit dans la visualisation, la correspondance entre les classes demande du temps à l'analyste. Il faut qu'il s'assure que les deux items soient bien alignés en faisant des aller-retour rapides avec ses yeux. Si les versions sont éloignées, ça devient encore plus difficile. Pour ce qui est de la compréhension des caractéristiques, les classes sont difficiles à comparer entre elles puisqu'elles sont éloignées les unes des autres. Encore une fois, l'observation requiert des aller-retour rapides de l'œil. La compréhension de l'architecture aussi est biaisée par le fait qu'il y a beaucoup d'espace vide dans la représentation. Elle reste par contre relativement facile à déterminer à l'aide des séparateurs du *Treemap*.

5.3 Animation

L'utilisation de l'animation est venue du problème occasionné par la trop grande quantité de données. Toutes les données sont toujours représentées, mais à des temps différents. De plus, il est possible d'utiliser les différents attributs liés à la cohérence temporelle pour aider l'analyste à retrouver les éléments importants dans la visualisation. Les changements sont aussi beaucoup plus faciles à détecter car l'œil est automatiquement attiré par ceux-ci. Un autre avantage indéniable de l'animation est sa correspondance directe avec l'idée que l'on se fait du développement de projets itératifs. Le temps qui passe dans l'animation et son déroulement représente bien le temps qui passe dans un projet avec les versions supplémentaires qui s'ajoutent. Dans la suite de cette section nous allons présenter en détails différents algorithmes d'animation que nous avons développés.

5.3.1 Stratégie avec positions fixes

Cette stratégie utilise le même principe concernant la place unique des classes que l'algorithme où le placement présente les versions les unes à côté des autres (voir sec-

tion 5.2). Un arbre est formé contenant toutes les classes de toutes les versions en retirant les doublons. On applique l'algorithme du *Treemap* modifié sur cet arbre. Cet algorithme permet alors d'assigner une place à chacune des classes qui a existé à un moment ou à un autre dans le logiciel. Ensuite, selon la version sélectionnée par l'analyste, il ne reste plus qu'à afficher seulement les classes appartenant à cette version avec leurs caractéristiques. Les figures 5.2 et 5.3 montre comment les différents éléméments sont placés d'une version à une autre à l'aide d'un exemple simple. La transition entre deux versions successives se fait automatiquement et bien que ce soit des objets différents qui sont dessinés, c'est-à-dire les classes de la prochaine version, l'analyste ne le voit pas de cette manière. La transition est plutôt perçue comme des ajouts, des retraits et surtout des modifications par rapport aux classes présentes dans la version précédente.

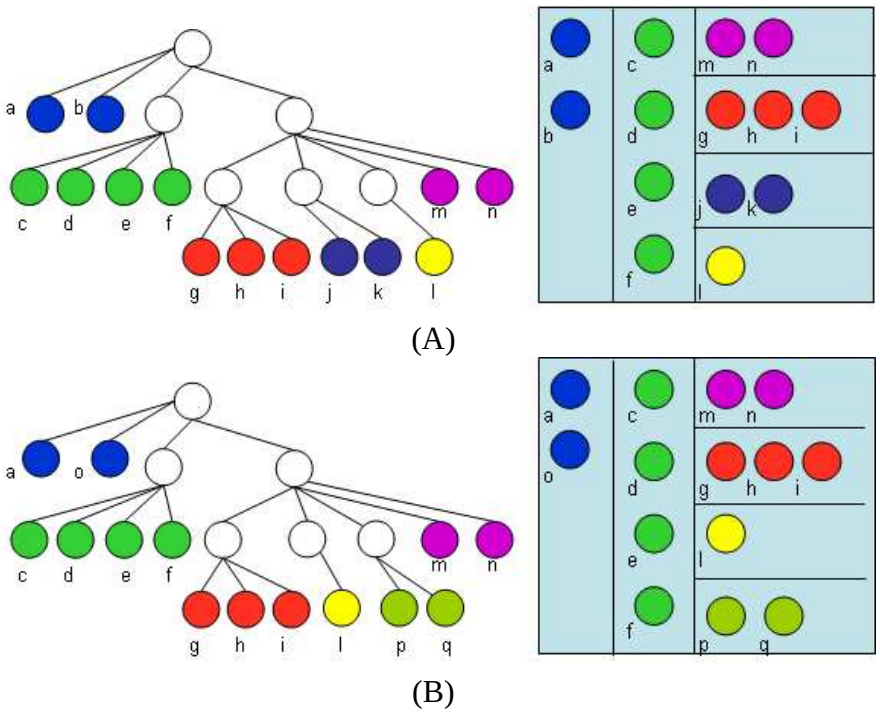
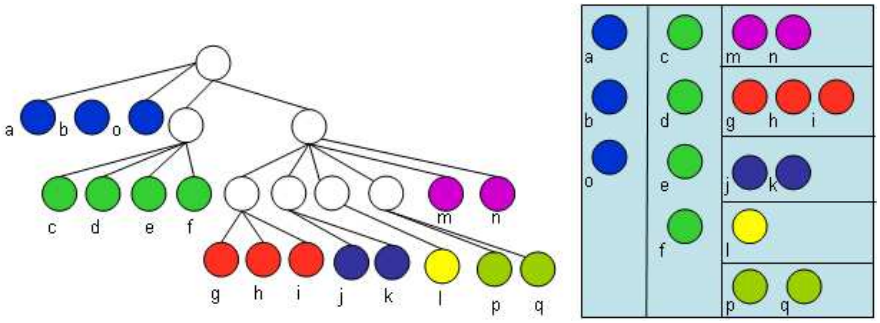
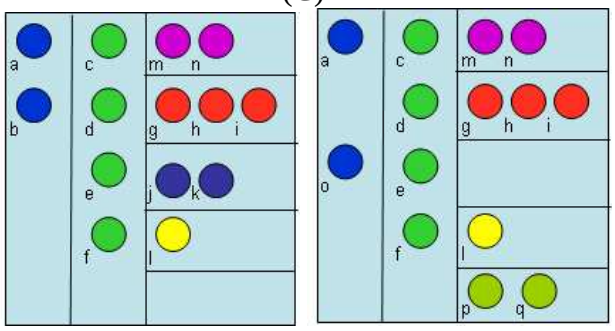


Figure 5.2 – Schéma de l'évolution à position fixe. (A) À gauche, on voit l'arbre de la première version d'un logiciel fictif alors qu'on voit sa représentation à l'aide de l'algorithme du *Treemap* modifié à droite. (B) On voit ici les mêmes éléments de la deuxième version du logiciel fictif.

D'une part, cette façon de présenter les données sauve beaucoup d'espace. D'autre



(C)



(D)

Figure 5.3 – Schéma de l'évolution à position fixe (suite). (C) La stratégie avec positions fixes fusionne les deux arbres en prenant tous les éléments des deux versions. (D) On voit ici la représentation vue par l'analyste de la première version à gauche et la représentation vue par l'analyste de la deuxième version à droite. Il y a des trous dans la première version pour laisser la place aux éléments qui vont s'ajouter et il y a des trous dans la deuxième version car des éléments ont été supprimés.

part, on peut mieux utiliser la cohérence temporelle et distinguer automatiquement ce qui est modifié de ce qui ne l'est pas. On peut suivre chacune des classes facilement car elles restent au même endroit. Les changements des caractéristiques sont aussi plus facilement observables parce que les classes sont stables. La compréhension de l'architecture est semblable à celle présentant de multiples versions sur la même image.

5.3.2 Stratégie avec positions fixes et animation des caractéristiques

Cette stratégie est bâtie sur le principe de la représentation en positions fixes. Le même type de placement pour chacune des versions est utilisé. La plus grande différence réside dans l'ajout d'images intermédiaires pour montrer les changements des métriques

de façon graduelle. Ces quelques images supplémentaires entre les versions fixes sont très efficaces pour attirer l'attention des analystes sur les changements. Quand il y a seulement deux images pour différencier deux versions, il est possible de manquer les changements les plus subtils ; les images intermédiaires laissent le temps à l'analyste de mieux voir tous les changements. Il y a par contre un défaut à cette représentation. Ces changements graduels n'ont pas d'équivalents dans le code lui-même. En effet, les changements sont soumis dans la base de données de façon directe et les métriques ne se modifient pas graduellement entre deux versions, mais plutôt de façon instantanée aussitôt que le code est ajouté. Cette distorsion de la réalité n'est pas grave en soi dans la mesure où elle contribue à faciliter la perception des changements. Comme expliqué dans l'article de Ramachandran [64], l'accentuation de la réalité, appelé *peak shift*, permet au cerveau de reconnaître plus facilement certaines images. La figure 5.4 donne un aperçu de l'évolution représentée à l'aide de la stratégie à positions fixes. Cette image correspond tout aussi bien aux sections 5.3.1 et 5.3.2 puisqu'il est impossible de montrer les images intermédiaires sur papier.

5.3.3 Stratégie avec animation du placement

Cette stratégie va un peu plus loin et introduit le mouvement du placement, c'est-à-dire que les classes sont appelées à se déplacer durant la visualisation vers leur nouvelle position. Pour une version donnée, on a donc l'impression de visualiser le logiciel comme présenté dans le chapitre 3. Il y a plusieurs images intermédiaires entre les représentations des versions pour permettre aux analystes de mieux suivre le mouvement des classes. Dans un premier temps, les classes sont déplacées sans apporter les modifications relatives aux métriques. Ensuite, les modifications des caractéristiques sont apportées à la configuration précédente de la même façon que pour la section 5.3.2. Comme stipulé par Plaisant et Fekete [18], cette stratégie est préférable à celle consistant à faire tous les changements en même temps. De plus, pour éviter qu'une classe ajoutée au centre d'un paquetage ne vienne chambouler toute la configuration, on choisit plutôt de l'insérer à la fin du paquetage même si elle ne respecte pas l'ordre alphabétique. Ceci diminue beaucoup les mouvements inutiles à l'intérieur des paquetages et contribue à

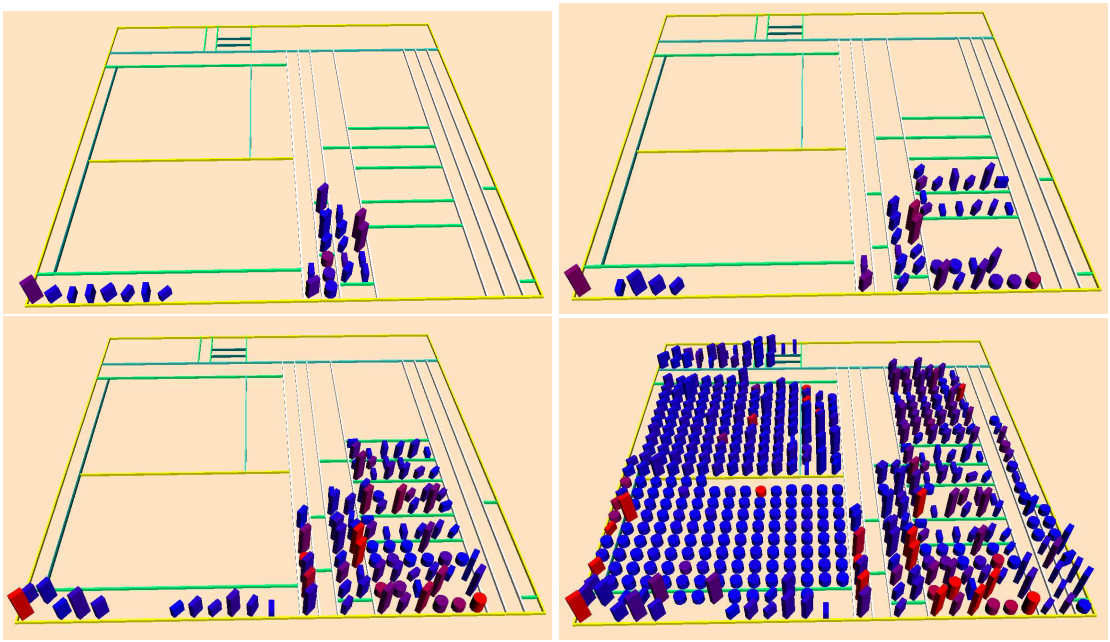


Figure 5.4 – Représentation de l'évolution à position fixe. Cette représentation comporte 4 versions spécifiques du logiciel *Freemind*, un outil servant à organiser les idées. Ces images ne sont jamais visibles en même temps et l'analyste navigue dans le temps à l'aide du clavier pour les visualiser. Dans cette stratégie, les classes conservent toujours la même position d'une version à l'autre.

une visualisation plus claire.

On remarque sans contredit une amélioration au niveau de l'espace utilisé parce que pour les versions préliminaires contenant moins de classes, la représentation est resserrée et de l'espace est ainsi économisé. La compréhension de l'architecture est elle aussi améliorée puisque les espaces vides qui apparaissent habituellement ne sont plus là et la compréhension ne demande pas plus d'efforts que pour la visualisation d'une version unique. Par contre, le suivi des éléments est plutôt difficile car l'algorithme modifié du *Treemap*, bien que déterministe, peut changer complètement si quelques classes sont ajoutées ou retirées. Il est évident que l'animation en deux temps, des déplacements d'une part et des modifications des caractéristiques d'autre part, aident à réduire ce problème. Cependant, il reste toujours plus difficile de faire le suivi des éléments avec ce type de représentation qu'avec l'algorithme utilisant des positions fixes. La figure 5.5 montre le résultat d'une telle stratégie sur l'animation de l'évolution.

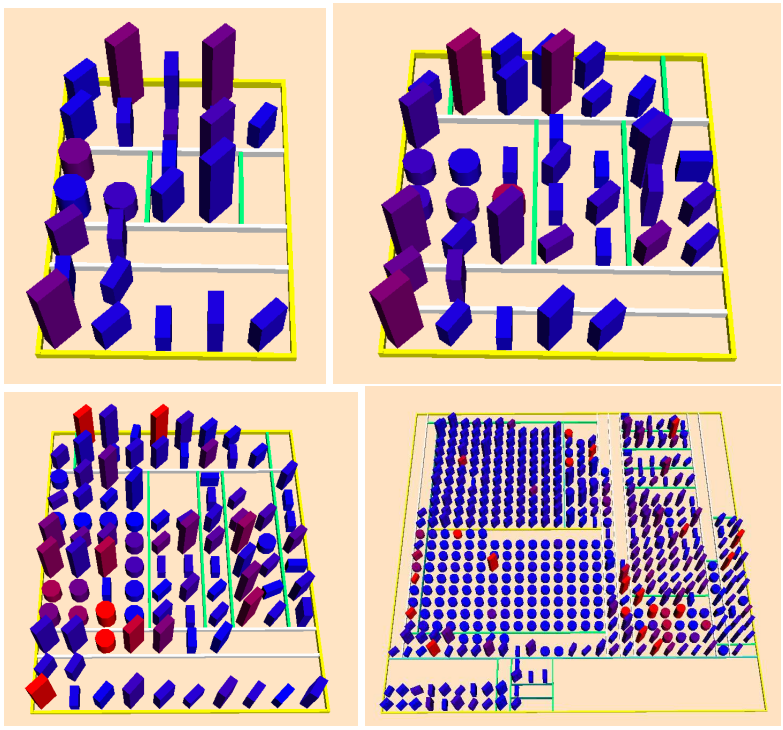


Figure 5.5 – Représentation de l'évolution avec animation du placement. Cette représentation comporte 4 versions spécifiques du logiciel *Freemind*, un outil servant à organiser les idées. Ces images ne sont jamais visibles en même temps et l'analyste navigue dans le temps à l'aide du clavier pour les visualiser. Ici, les classes ne gardent pas nécessairement la même position d'une image à l'autre pour économiser de l'espace. Les déplacements et les modifications des caractéristiques sont animés en deux phases.

5.3.4 Stratégies hybrides

Les stratégies hybrides se veulent un compromis entre les représentations à position fixe et les représentations où le placement peut varier. Une première stratégie se situant plus près de la solution où les positions sont fixes consiste à garder la forme de l'arbre où toutes les classes sont présentes en même temps. Par contre, toutes les régions vides qui se situent à l'extérieur du périmètre formé par les classes sont coupées. Cette solution réduit beaucoup la surface utilisée lors de la visualisation des premières versions où peu de classes sont présentes en général. Plusieurs paquetages sans classe sont toutefois toujours représentés, ce qui amène un peu de confusion. La figure 5.6 présente les résultats

de cette stratégie.

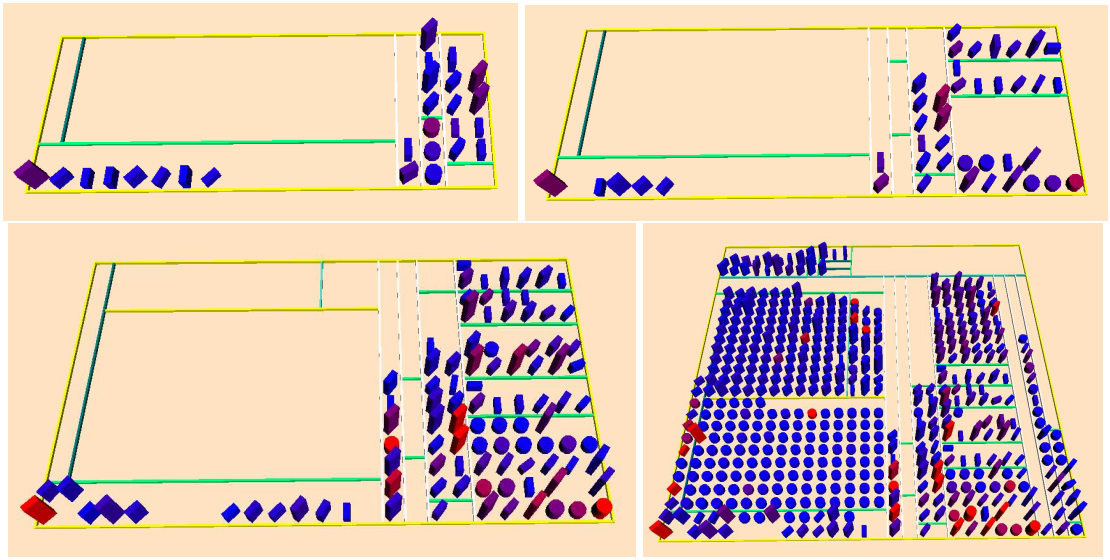


Figure 5.6 – Représentation de l'évolution avec la première approche hybride. Cette représentation comporte 4 versions spécifiques du logiciel *Freemind*, un outil servant à organiser les idées. Ces images ne sont jamais visibles en même temps et l'analyste navigue à l'aide du clavier pour les visualiser. Ici, les parties inutiles sont coupées pour améliorer l'utilisation de l'espace.

La deuxième stratégie est légèrement plus complexe et se rapproche aussi plus de la solution qui consiste à faire bouger les classes et les paquetages pour mieux utiliser l'espace à chaque fois. Il s'agit de garder la même configuration des classes, mais de resserrer les éléments les uns contre les autres quand c'est possible. Il s'agit seulement de s'assurer que les classes placées au-dessus ou à gauche d'une autre classe conservent cette position relative. La figure 5.7 montre un exemple simple de cette stratégie. Ceci amène des mouvements de classes qui ne sont pas nécessairement reliés à des modifications dans le code, un peu à la manière de l'algorithme où les classes bougent. Par contre, ces mouvements sont plus faciles à suivre que dans la version où l'espace est réorganisé à chaque fois. En effet, les mouvements sont toujours très constants et semblables pour toutes les classes. La figure 5.8 permet d'apprécier l'évolution sur plusieurs versions à l'aide de cette stratégie.

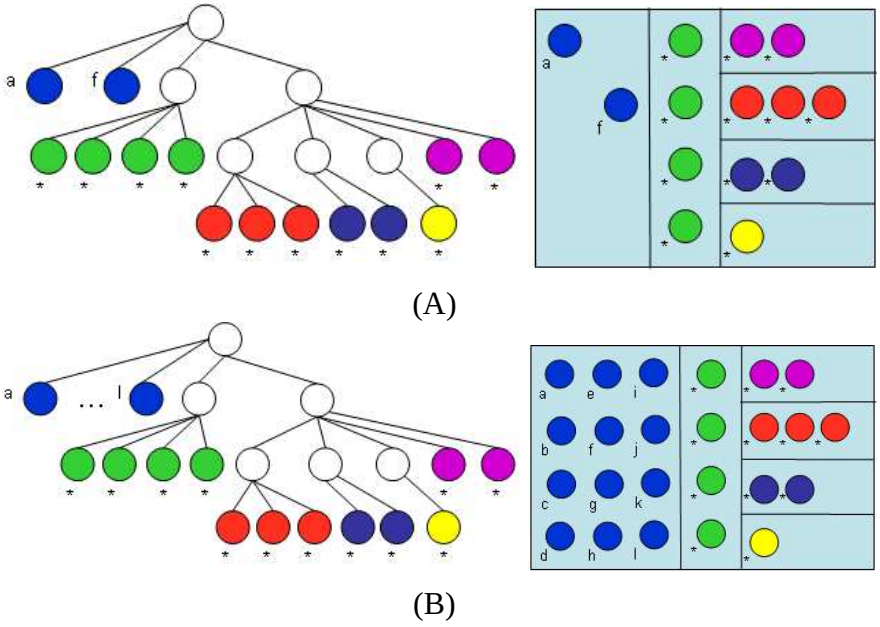


Figure 5.7 – Schéma de l'évolution de la deuxième stratégie hybride. (A) À gauche, on voit l'arbre de la première version d'un logiciel fictif alors qu'on voit sa représentation à l'aide de la deuxième stratégie hybride à droite. (B) On voit ici les mêmes éléments de la deuxième version du logiciel fictif. Il est à noter que dans la première version, les classes a et f se sont rapprochées pour enlever certains trous, mais ont toujours garder leur position relative les unes par rapport aux autres.

5.4 Comparaison des techniques

Ces stratégies ont des qualités différentes exploitables dans divers contextes. Le tableau 5.1 se veut une récapitulation des différentes stratégies servant à montrer l'évolution. Pour l'évaluation de ces stratégies, on utilisera les caractéristiques mentionnées plus haut. La diversité des tâches à accomplir dans la visualisation du logiciel rend non-pertinent le classement ordonné de ces stratégies ou le choix d'un vainqueur.

En général il vaut mieux sacrifier l'espace utilisé et opter pour une solution où la position est fixe pour mieux évaluer les modifications des différentes métriques à travers les versions. C'est donc le meilleur choix pour le genre de visualisation qui se base sur la qualité et les métriques. Par contre, les approches utilisant le mouvement des classes peuvent contribuer à mieux déceler les ajouts, les suppressions et les réorganisations à

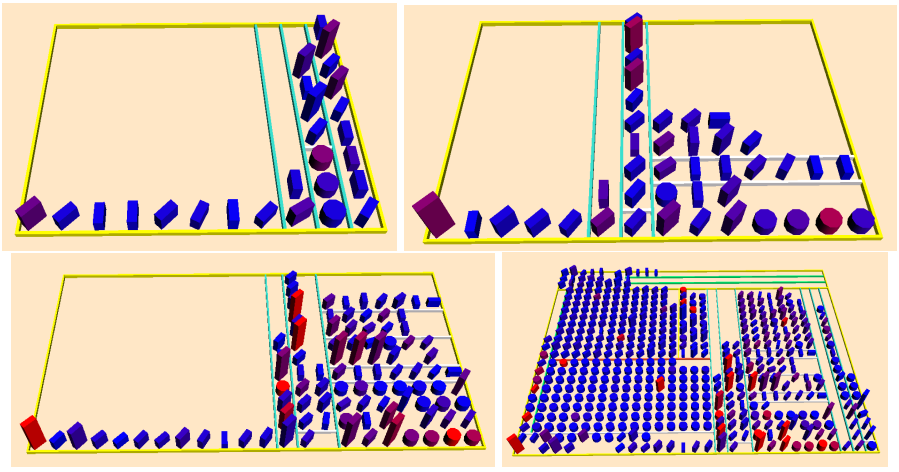


Figure 5.8 – Représentation de l'évolution avec la deuxième approche hybride. Cette représentation comporte 4 versions spécifiques du logiciel *Freemind*, un outil servant à organiser les idées. Ces images ne sont jamais visibles en même temps et l'analyste navigue à l'aide du clavier pour les visualiser. Ici, la configuration est resserrée pour réduire l'espace utilisé.

l'intérieur des paquetages. Les solutions intermédiaires peuvent aussi être considérées comme de bons compromis dans la plupart des situations. Ceci étant dit, pour la plupart des phénomènes répertoriés dans la section 5.5, il est plus simple pour l'analyste d'utiliser une stratégie où les positions restent fixes.

5.5 Applications possibles

Encore une fois, la visualisation de l'évolution du logiciel permet une approche exploratoire et une approche dirigée à la fois. En effet, tout comme c'est le cas avec la représentation de version unique, l'analyste peut regarder les animations en recherchant quelque chose qui attirera son attention ou encore avoir une liste de patrons graphiques précis à rechercher qui correspondent à des phénomènes reliés à la qualité ou au processus logiciel. Le premier cas permet de trouver des phénomènes inédits qui peuvent servir de base aux recherches dirigées une fois assimilés par l'analyste.

Il est aussi à noter que plusieurs anomalies décrites dans la section 3.3.2 peuvent être découvertes ou étudiées plus en profondeur à l'aide de la visualisation de l'évolution.

Stratégie	Utilisation de l'espace	Suivi des classes	Suivi des caractéristiques	Compréhension de l'architecture
Côte-à-Côte	mauvais	mauvais	très mauvais	moyen
Pos. fixe	moyen	très bon	bon	moyen
Pos. fixe (anim.)	moyen	très bon	très bon	moyen
Mouvement	très bon	moyen	bon	très bon
Hybride (fixe)	bon	très bon	très bon	moyen
Hybride (mouv.)	bon	bon	bon	bon

Tableau 5.1 – Tableau comparatif des stratégies d'évolution. Étude qualitative considérant individuellment les différentes caractéristiques de l'animation de l'évolution dans la visualisation.

Par exemple, il est possible de voir si le couplage élevé d'une classe est le résultat d'une conception initiale ou l'ajout successif de plusieurs connexions à cette classe. Pour l'anomalie *Shotgun Surgery* par exemple, il est intéressant de savoir si la classe a été ajoutée tardivement dans le système et qu'elle communique avec plusieurs classes réparties dans divers paquetages dès son arrivée ou si encore elle communiquait avec peu de classes au début et que tous les nouveaux paquetages construits y ont été liés par la suite. Dans le premier cas, il s'agit plutôt d'une erreur de conception ou de vocation pour la classe, alors que dans le deuxième cas, il s'agit plutôt d'une erreur dans le processus où les vérifications de couplage ont mal été faites tout au long du cycle de vie du logiciel.

Des phénomènes précis concernant la recherche dirigée ont été découverts jusqu'ici. En tout, quatre phénomènes ont déjà été identifiés et étudiés par notre équipe. Il existe donc une technique documentée pour les retrouver dans de nouveaux logiciels.

5.5.1 Changement d'identité

Le phénomène que nous avons nommé *changement d'identité* est tout simplement l'action de changer le nom d'une classe et de conserver cette classe à l'intérieur d'un même paquetage. Ce phénomène peut survenir parce que la nomenclature d'un logiciel a été modifiée ou parce que le nom de la classe ne correspond plus à son comportement. Bien sûr, le nom d'une classe est appelé à changer à différentes occasions au cours de

son cycle de vie. Par contre, quand la documentation n'est pas à jour ou inexistante, il devient extrêmement difficile de retrouver une classe renommée. Ce sont des situations qui peuvent survenir souvent puisque plusieurs programmeurs travaillent en général sur un même projet. Le problème est encore plus important quand le code concerné se trouve à l'intérieur d'une librairie populaire. Par exemple, il est commun de vouloir migrer du code de la version de *JDK 1.x* à *JDK 1.x+1*. Il en résulte alors des erreurs et des avertissements stipulant que les classes n'existent plus ou qu'elles seront enlevées dans les versions suivantes.

La stratégie de détection utilise le principe selon lequel les classes vont disparaître et apparaître à un autre endroit lors d'une transition. Puisque leur nom change, elles sont considérées comme de nouvelles classes dans le modèle de VERSO. Elles auront par contre vraisemblablement les mêmes caractéristiques graphiques avant et après le changement de nom.

La stratégie de détection pour ce phénomène se compose de la façon suivante :

1. Utiliser n'importe quelle association entre métriques possédant une grande variabilité entre les classes.
2. Naviguer à travers les versions.
3. Repérer une transition suspecte où des classes apparaissent et disparaissent.
4. Pour chaque transition suspecte, rejouer l'animation plusieurs fois pour identifier les classes apparaissant et disparaissant entre les versions, mais en conservant toujours la même forme. Quand une classe est renommée, elle ne change jamais radicalement et conserve des métriques semblables
5. Comparer le nom des classes ou le code pour s'assurer qu'il s'agit bien d'un cas de *changement d'identité*.

Souvent, le nom de plusieurs classes dans un même paquetage sera changé en même temps. Il peut s'agir de l'ajout ou du retrait d'un suffixe ou d'un préfixe par exemple. Ceci facilite la détection du phénomène car plusieurs classes bougent ensemble. Par exemple, la figure 5.9 montre un cas de *changement d'identité* dans *RSSOWL*.

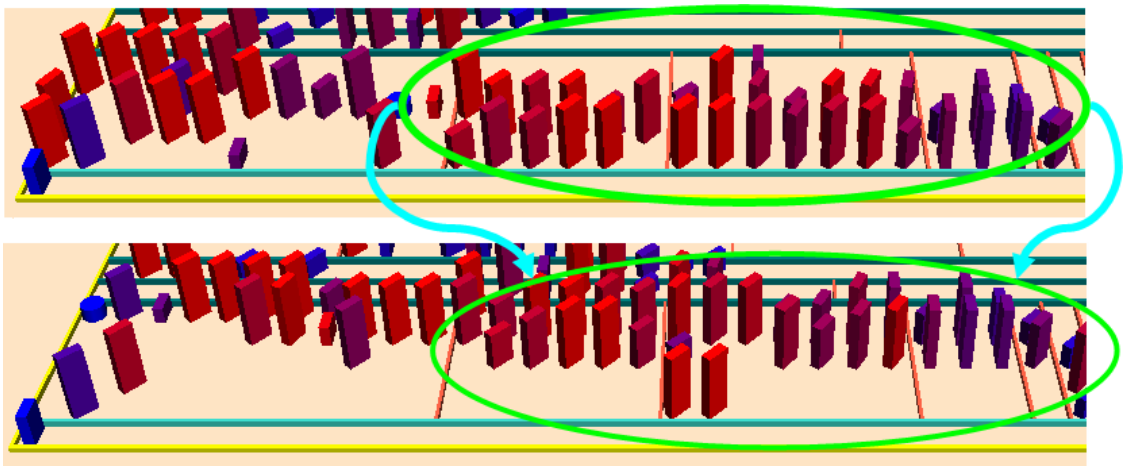


Figure 5.9 – Un exemple de *changement d'identité* à l'intérieur du logiciel *RSSOwl*. Les classes encerclées dans l'image du haut sont renommées et se déplacent donc en même temps à leur nouvelle position encerclée dans l'image du bas. En fait, le préfixe *RSSOWL* a été enlevé du nom de toutes les classes.

5.5.2 Protection des témoins

Le phénomène que nous avons nommé *protection des témoins* est très semblable à celui du *changement d'identité*. Dans ce cas particulier, les classes changent complètement de paquetage. Puisque dans notre approche de visualisation, les classes sont représentées par leur nom complet, il s'agit aussi d'une classe renommée. La classe peut avoir un nom individuel différent ou non. La stratégie de détection n'est pas beaucoup différente du *changement d'identité*. Par contre, dans ce cas, la détection du phénomène demande un peu plus de concentration car les classes sont généralement déplacées plus loin dans la visuation. La figure 5.10 montre un exemple de *protection des témoins* dans l'application *Freemind.*

5.5.3 Surcharge de responsabilité

Le phénomène que nous avons nommé *surcharge de responsabilité* se produit quand une classe a une taille raisonnable au début de l'évolution d'un logiciel, mais que cette dernière devient de plus en plus grande et de plus en plus couplée tout au long du processus d'évolution. Cette situation reflète un mauvais choix de conception dans les pre-

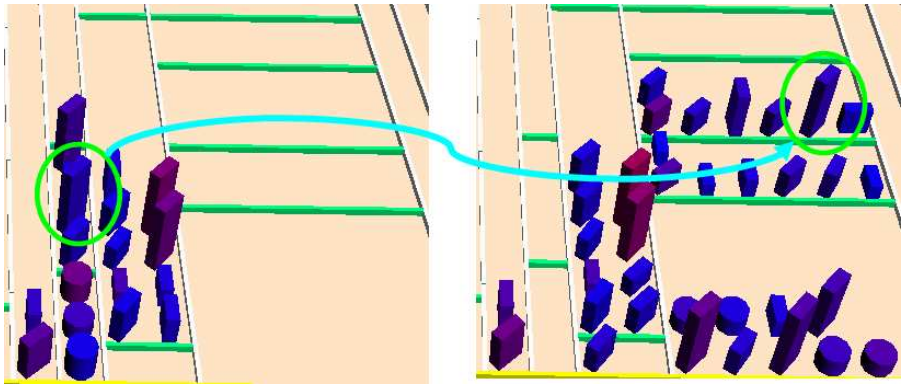


Figure 5.10 – Un exemple de *protection des témoins* trouvé dans l'application *Freemind.* La classe *NodeModel* (encerclée dans l'image de gauche) initialement dans le paquetage *freemind.model.simplemodel* a été renommée *MindMapNodeModel* (encerclée dans l'image de droite) et déplacée dans le paquetage *freemind.modes.mindmapnode*.

mières versions du logiciel puisque les développeurs ne prévoyaient pas les proportions que le logiciel allait prendre. Une solution raisonnable pour un problème de petite envergure est adoptée au début du développement, mais le problème est que cette solution est conservée même si le logiciel grossit. Il en résulte que plusieurs responsabilités sont ajoutées dans une classe. Cette classe devient vite ingérable et risque de causer des problèmes aux classes avec lesquelles elle communique. En général, la découverte d'un tel cas demande une importante refonte de la conception et une inspection des raisons qui ont mené à une telle situation.

Cette fois-ci, la détection est bien sûr basée sur la progression constante des métriques de couplage et de complexité pour une classe donnée. Les étapes à suivre sont décrites ici :

1. Utiliser l'association entre caractéristiques graphiques et métriques suivantes : (couleur et CBO, hauteur et WMC, orientation et LCOM5)
2. Naviguer à travers toutes les versions.
3. Dans les dernières versions, identifier une classe qui soit à la fois complexe, couplée et relativement peu cohésive. Elles peuvent être distinguées par leur couleur rouge, leur grande taille et leur rotation.

4. Pour chaque classe suspecte :
   - Rejouer l'animation pour vérifier si l'évolution suit le patron recherché.
   - Vérifier si la croissance peut être expliquée par la logique du logiciel. Ceci implique de consulter le nom de la classe et possiblement son code.

Il semble que ce phénomène puisse être facilement détecté à l'aide de moyens automatiques. Par contre, comme va le montrer le prochain exemple, des fluctuations peuvent venir fausser cette interprétation pour un algorithme automatique alors qu'un analyste sera en mesure de découvrir le phénomène sans être autant influencé. En effet, si une classe rétrécit ou son couplage diminue au cours de l'évolution, mais qu'elle récupère toujours ce qu'elle a perdu, il est fort possible qu'on soit tout de même en présence d'un cas de *surcharge de responsabilité*. La figure 5.11 montre un cas de *surcharge de responsabilité* trouvé dans le logiciel *Freemind.*

5.5.4 Retour sur décision

Le phénomène que nous avons nommé *retour sur décision* survient quand le logiciel quitte une configuration quelconque vers une autre et revient à la précédente dans une de ses versions subséquentes. Bien qu'il puisse être sage de revenir sur une décision prise antérieurement et constater qu'une ancienne version était préférable, ce phénomène pose des questions sur la communication à l'intérieur de l'équipe de développement ou de maintenance. Il peut s'agir d'un superviseur qui considère que les changements apportés n'en valent pas la peine ou tout simplement un autre programmeur qui revient en arrière par accident ou de façon délibérée. Dans tous les cas, cette opération peut être coûteuse et fait perdre du temps dans le processus de développement.

Pour la stratégie de détection, il s'agit cette fois d'observer s'il y a de la redondance dans les images montrées à l'écran lors du parcours des versions.

1. Utiliser n'importe quelle association de métriques, mais s'assurer qu'elle signifie quelque chose d'intéressant pour l'analyste et qu'il existe beaucoup de variabilité entre les valeurs de ces métriques.
2. Naviguer à travers toutes les versions.

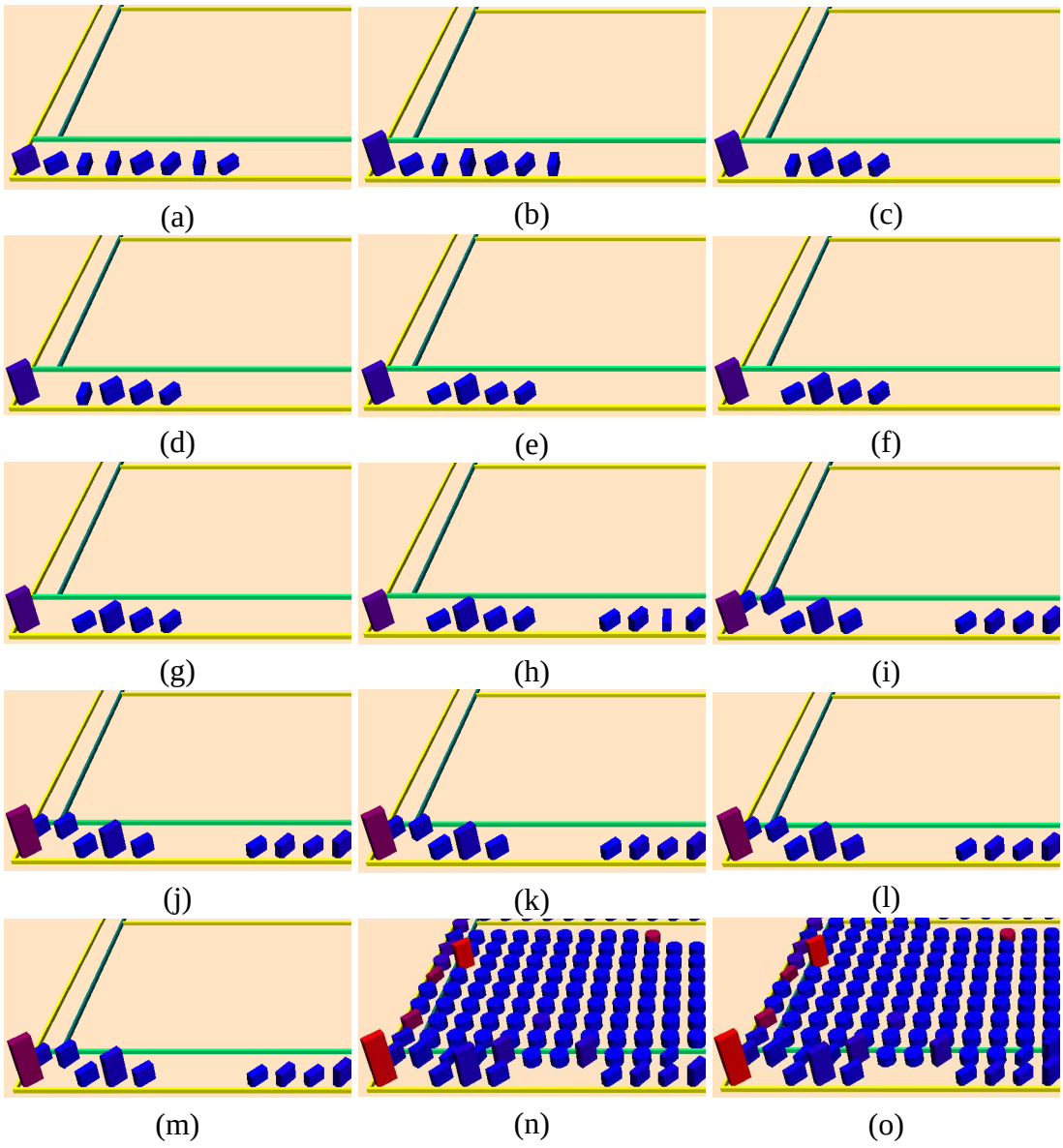


Figure 5.11 – Un exemple de *surcharge de responsabilité* trouvé à l'intérieur du logiciel *Freemind*. La classe *Controller* positionnée en bas à gauche devient de plus en plus couplée à mesure que le programme grossit. Veuillez noter qu'un réusinage a été tenté à la version 0.4 (g). La diminution est encore plus évidente en (h). Une approche uniquement automatique aurait pu manquer ce cas intéressant puisque la croissance n'est pas parfaitement continue.

3. Repérer une série de versions où des classes changent d'allure et reviennent ensuite à une représentation semblable.
4. Étudier le segment d'animation plusieurs fois.
5. Vérifier le code pour confirmer le *retour sur décision*.

Comme c'était le cas pour le *changement d'identité*, il se peut que plusieurs classes soient impliquées dans un *retour sur décision*. Cette situation rend la détection de ce phénomène plus facile à détecter pour l'analyste. La figure 5.12 montre un cas de *retour sur décision* trouvé dans le logiciel *Freemind.*

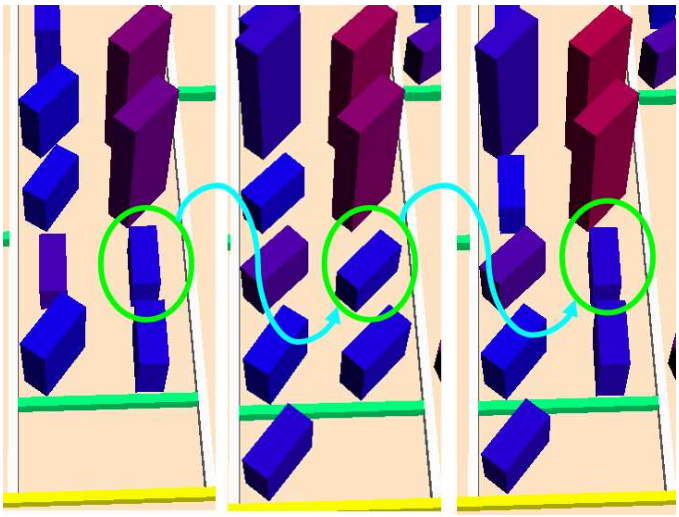


Figure 5.12 – Un exemple de *retour sur décision* trouvé dans l'application *Freemind.* On suit ici la classe *forkviewnode* qui change de configuration puis revient à celle qu'elle possédait initialement.

5.5.5 Observations générales

Il n'y a pas que les chercheurs qui peuvent utiliser VERSO, mais aussi les développeurs de logiciels. Une équipe peut suivre les modifications sur son logiciel et en évaluer la qualité tout au long du processus de création. On peut alors détecter à quel moment le logiciel a commencé à se détériorer ou à quel moment une faute a été introduite. Il est important de vérifier ce genre de situations pour éviter que les erreurs ne se reproduisent

lors des projets futurs. La visualisation de l'évolution nous permet aussi de comprendre comment un logiciel s'est retrouvé dans une situation particulière. Une fois la cause comprise, il devient beaucoup plus facile de régler le problème.

Pour finir, la visualisation de l'évolution permet de recueillir des informations non seulement sur le produit logiciel et sa qualité, mais aussi sur son processus de création. En effet, il est possible d'étudier dans quel ordre sont construites les classes et les paquetages, ou encore d'évaluer si les classes sont construites de façon incrémentale ou si une fois apparue, elles gardent leur forme tout au long du processus. On peut aussi étudier la différence entre la construction d'un logiciel et sa modification une fois qu'il est terminé. Il est aussi possible de changer les informations données par les caractéristiques graphiques en enlevant les métriques et en les remplaçant par des informations sur les dates de modifications ou sur les auteurs.

5.6 Évaluation et discussion

Cette section explique en quoi l'approche pour visualiser l'évolution se démarque de celles qui sont présentées dans l'état de l'art. De plus, nous donnons quelques exemples de situations dans lesquelles notre stratégie de visualisation s'avère utile.

5.6.1 Évaluation générale

Dans un premier temps, notre approche est la seule à représenter la qualité du logiciel à travers les métriques et à utiliser l'animation et les images intermédiaires pour suivre aussi leur évolution. Les approches présentent tantôt des données relatives à l'évolution sur une même image tantôt des données qui ne sont pas en lien avec la qualité et les métriques (voir section 2.4.3). Bien que les approches où une seule image est présentée permettent d'avoir une vue globale, elles empêchent de montrer un grand nombre d'entités à la fois. Si toutefois, on réussit à montrer plus d'entités, on est contraint à réduire les informations montrées pour chaque entité, par exemple, en montrant les entités comme un pixel unique. En raison de l'animation, notre approche peut montrer plus d'entités et plus de caractéristiques individuelles pour chaque entité. Elle peut donc être

efficacement utilisée pour représenter la qualité d'un logiciel et évaluer son évolution.

Dans un deuxième temps, VERSO simule bien la représentation mentale que l'on se fait d'un logiciel qui évolue. En effet, l'analyste a réellement l'impression de voir les modifications s'afficher devant lui. Le logiciel change de configuration en passant d'un état initial vers un état modifié comme si les changements du code étaient effectués en accéléré dans le système. Ceci donne l'impression à l'analyste d'être plus impliqué par les modifications. De ce fait, il est plus intéressé par ce qu'il voit et par l'analyse qu'il a à faire. Ce n'est pas nécessairement le cas avec des représentations de type matrice [45, 51, 56] qui sont plus éloignées de la réalité du processus évolutif du logiciel.

Dans un troisième temps, puisque le prototype présente les versions les unes à la suite des autres, il est possible de s'arrêter sur une version en particulier pour l'étudier plus en profondeur comme s'il s'agissait de la visualisation d'une version individuelle (voir chapitre 3). Avec les représentations en matrice ou en ligne de temps [78], il devient moins évident de connaître toutes les facettes d'un programme à un instant précis.

Il existe toutefois un point négatif, quand le logiciel change complètement de structures ou que des paquetages sont renommés. L'animation de l'évolution n'aidera pas l'analyste plus efficacement que la visualisation de versions individuelles. En effet, les classes sont considérées comme les mêmes entre les versions en fonction de leur nom complet incluant la hiérarchie de paquetages. Il s'agit de changer le nom d'un niveau de paquetage pour que le tout soit changé. Dans nos stratégies, le paquetage en entier disparaît et réapparaît dans un autre secteur de la visualisation. Par contre, nous avons vu dans la section 5.5 que ce problème peut être recyclé dans la détection de phénomènes logiciels.

Bien qu'aucune expérience formelle n'ait été menée concernant la partie évolution de ce travail, l'utilisation de VERSO par différents membres de l'équipe montre qu'il est polyvalent et utile pour évaluer la qualité. De plus, les observations faites sur plusieurs logiciels démontrent qu'il est véritablement possible de retrouver des évènements à l'intérieur du logiciel. Il est aussi possible de l'utiliser pour mieux comprendre le processus de l'évolution de ce dernier. Il n'en reste pas moins qu'une expérience impliquant des sujets est indispensable, mais faute de temps, elle sera laissée pour travaux futurs.

CHAPITRE 6

CONCLUSION

Ce mémoire s'est intéressé à la représentation du logiciel en trois dimensions dans le but d'en évaluer la qualité. Nous avons étudié les métriques et le système visuel humain. Ensuite nous avons montré comment le logiciel était présenté dans VERSO et nous avons fait une expérience sur les algorithmes de placement. Par la suite, nous nous sommes intéressés à l'évolution du logiciel.

6.1 Rétrospective

VERSO utilise la visualisation du logiciel pour permettre à l'analyste de le comprendre et de l'analyser. Il faut donc que l'analyste soit en mesure d'interpréter rapidement de grands ensembles de données. L'approche manuelle est précise, mais trop lente alors que l'approche complètement automatique ne permet pas à l'analyste d'utiliser son jugement et d'évaluer le contexte d'un logiciel. Cette technique permet d'être plus efficace compte tenu de la faiblesse de la théorie dans le domaine.

Le but premier de VERSO étant de permettre à l'analyste d'évaluer la qualité du logiciel, il a fallu trouver un modèle sous-jacent pour représenter cette qualité. Nous avons choisi d'utiliser les métriques qui ont prouvé leur efficacité à prédire des phénomènes de qualité à l'intérieur du logiciel par le passé. De plus, elles sont faciles à transformer vers des valeurs graphiques puisqu'elles sont toutes de type numérique.

Ensuite, nous avons étudié les forces et les faiblesses du système visuel humain. Il faut faire en sorte que la perception des caractéristiques du logiciel soit instanée et faire attention aux biais introduits par les différentes caractéristiques graphiques. Le système visuel humain est puissant, mais devient vite saturé.

Le chapitre 3 est consacré à la représentation du logiciel dans la visualisation. Les classes sont représentées comme des boîtes en trois dimensions dont la hauteur, la couleur et l'orientation par rapport à l'axe des Y forment les caractéristiques graphiques.

Les métriques sont associées à ces caractéristiques graphiques et les variations dans les caractéristiques graphiques sont interprétées comme des variations dans les métriques. Pour les différencier des classes, les interfaces sont représentées par des cylindres. Ensuite, ces éléments sont placés sur un plan en deux dimensions suivant deux algorithmes soient le *Treemap* et le *Sunburst*. Ces algorithmes conservent les informations sur l'architecture tout en utilisant bien l'espace disponible.

Dans le but d'évaluer la différence entre les deux algorithmes et de montrer leur utilité dans la détection de phénomènes liés à la qualité, nous avons mené une expérience. Les 15 sujets devaient répondre à 20 questions portant sur la qualité en utilisant les deux algorithmes de placement décrits plus haut et un placement naïf pour comparer. Les temps obtenus montrent que le *Treemap* et le *Suinburst* sont plus efficaces que l'algorithme naïf et l'approche purement manuelle. Il a, par contre, été impossible de déterminer de façon statistiquement significative lequel des deux algorithmes principaux est le meilleur. Nous avons toutefois décidé de continuer le développement de VERSO avec le *Treemap*.

Pour finir, nous avons étudié l'évolution du logiciel à travers la visualisation. Nous avons opté pour une approche utilisant l'animation et le contrôle par l'analyste sur le temps pour représenter visuellement l'évolution. La cohérence entre les différentes versions d'un logiciel est représentée par la cohérence entre les différentes images de l'animation. Nous avons étudié quatre stratégies : une première où toutes les classes conservent la même position durant l'évolution, une deuxième où les classes bougent de façon à réduire l'espace utilisé et deux stratégies hybrides. Pour les besoins actuels d'analyse, il s'est avéré que les stratégies où les classes gardent une position fixe sont plus efficaces. Nous avons aussi identifié quatre phénomènes reliés à la qualité qu'il est possible de détecter à l'aide de la visualisation : *changement d'identité, protection des témoins, surcharge de responsabilité, retour sur décision.*

6.2 Contributions

Les recherches effectuées dans le cadre de ce mémoire ont amené plusieurs contributions significatives par rapport à l'état de l'art dans le domaine de la visualisation du logiciel. De plus, plusieurs pistes observées peuvent encore être approfondies pour mener à des progrès concrets en visualisation.

6.2.1 Analyse de qualité

Pour commencer, notre approche de visualisation va au-delà de la simple représentation des données et propose à l'analyste des techniques pour évaluer le logiciel. Nous passons aussi à l'étape suivante en proposant la détection détaillée de certains phénomènes de qualité tant pour les versions uniques que pour l'évolution du logiciel.

6.2.2 Grande quantité d'information visible

L'approche proposée est en mesure de montrer une grande quantité d'éléments en même temps. Il est aussi possible de visualiser trois caractéristiques par élément en plus des informations sur l'architecture du programme. Pour un système de 20 000 classes, il est toujours possible de naviguer aisément dans le système et les informations peuvent être interprétées rapidement. Certains systèmes montrent plus d'information simultanément, mais la quantité d'information assimilée par l'analyste est moins grande.

6.2.3 Animation pour représenter l'évolution

Une autre innovation de VERSO est l'utilisation de l'animation pour représenter l'évolution du logiciel. Cette stratégie permet de représenter un plus grand nombre de version tout en évitant de saturer le système visuel humain. L'analyste passe d'une version à l'autre en observant les changements tout comme s'il passait d'une version du code à l'autre. De plus, la cohérence du logiciel est transformée en cohérence graphique pour permettre à l'analyste de se concentrer sur les changements et pour réduire l'effort cognitif nécessaire au suivi et à la comparaison des différents éléments du logiciel.

6.2.4 Mise en œuvre des principes de perception

Pour finir notre approche a été conçue en tenant compte des principes de perception et des forces et des faiblesses du système visuel humain. Cette problématique est rarement considérée dans les différentes approches de visualisation. Un point important est la perception instanée qui permet à l'analyste de repérer certaines caractéristiques sans faire le balayage des éléments. Un second point est d'éviter les interférences des caractéristiques graphiques. Ces interférences peuvent induire un biais dans la lecture des métriques.

6.3 Perspectives

Bien que des résultats intéressants aient été obtenus lors de notre expérience servant à démontrer l'efficacité de l'approche, la recherche concernant la visualisation reste en constant mouvement pour permettre des ajouts et peaufiner la visualisation. En plus des améliorations, il faut aussi considérer des nouveaux types de données et des nouveaux aspects du logiciel à évaluer. Il est aussi possible d'utiliser la visualisation de logiciel à d'autres fins que l'analyse de qualité en modifiant certains aspects de VERSO.

6.3.1 Améliorations de la visualisation de l'évolution

Nous avons vu divers algorithmes d'évolution au cours du chapitre 5. Ces algorithmes représentent un compromis entre l'utilisation de l'espace et la facilité avec laquelle on peut suivre les éléments d'une version à l'autre. Il serait intéressant de trouver une solution qui peut combler les deux besoins à la fois.

Une piste de solution réside dans la relaxation des positions dans le but d'en trouver une optimale. On peut par exemple utiliser des cercles contenus dans d'autres cercles et optimiser l'espace utilisé par la suite. Cette solution n'est pas la meilleure au point de vue de l'utilisation de l'espace pour une seule version, car généralement, on perd beaucoup d'espace entre les cercles de différentes grosseurs. Par contre, quand il s'agit d'évolution, ce type de placement peut s'adapter en fonction du nombre d'éléments présents. L'ajout d'éléments amène plusieurs petits déplacements ciblés. Il est possible de

partir d'une configuration précédente et de relaxer le placement précédent vers une autre configuration en laissant l'analyste observer les images intermédiaires. Il faut par contre trouver un moyen de faire converger plus rapidement les différents algorithmes. Des travaux non publiés de Kai Wetzel soulignent que c'est une tâche difficile. Par contre, des techniques de remplissage de cercles à base d'hexagones curvilignes montrés dans [24] peuvent accélérer la convergence des cercles unités à l'intérieur de plus grands cercles.

La recherche concernant le placement est la plus importante pour être en mesure de présenter de plus en plus d'éléments de façon cohérente. C'est pourquoi il faut continuer à mettre des efforts dans ce domaine et rechercher de nouvelles façons d'afficher les informations et surtout de les disposer efficacement.

6.3.2 Métaphore

La visualisation du logiciel existe non seulement pour l'analyse de celui-ci, mais aussi pour qu'il soit plus facile de le comprendre. En ce sens, la métaphore est un moyen efficace pour aider l'analyste à comprendre le logiciel qui lui est présenté. Les métaphores en informatique sont déjà utilisées depuis longtemps, on a qu'à penser à celle du bureau pour représenter les systèmes de fichiers. L'analogie entre un domaine connu comme le milieu urbain et un domaine moins connu comme le logiciel peut aider les débutants à apprendre certains concepts plus rapidement [55]. Une théorie concernant l'apprentissage par métaphores est que les liens entre les concepts généraux existent déjà dans le cerveau, donc les liens entre les synapses n'ont pas besoin d'être reconstruits entièrement lors de l'apprentissage [71].

Une métaphore privilégiée est celle de la ville. Bien qu'il existe plusieurs études qui ont été faites à l'aide de cette métaphore [15, 39, 62], certaines restent superficielles en se contentant d'avoir une correspondance graphique entre les éléments. D'autres ne sont pas orientées sur la qualité du logiciel mais plutôt sur la représentation des entités logicielles. Dans notre cas, les travaux seraient orientés sur les ressemblances sémantiques entre l'utilisation des quartiers et des immeubles dans une ville et l'utilisation des paquetages et des classes dans un logiciel. Par exemple, il a été remarqué que les paquetages utilitaires contiennent des classes qui sont grandes, mais qui interagissent avec un

nombre limité de personnes dans la ville. Ceci est analogique aux quartiers industriels où les usines sont aussi de gros bâtiments, mais elles n'interagissent qu'avec leurs employés, les magasins et les autres usines utilisant leurs produits. Ceci contraste avec le quartier des affaires ou le paquetage central qui communique avec un peu tout le monde dans la ville ou le logiciel. Déjà, une ébauche de résultats concernant le logiciel et la métaphore a été présentée dans une conférence [44].

D'autres analogies vouées à l'apprentissage ou tout simplement à nommer certains phénomènes logiciels devront être développées. Le seul principe de représenter les données graphiquement représente une métaphore par rapport à leur signification brute. Il est donc aussi souhaitable de trouver d'autres façons de représenter ces données même si c'est de manière abstraite.

6.3.3 Développement d'une solution intégrée

Notre approche est implantée dans un prototype servant à démontrer les principes de visualisation étudiés. Pour le faire fonctionner, on a besoin d'un extracteur de métriques, en l'occurence *POM* [26], qui lit les fichiers *CLASS*. Il est donc nécessaire de connaître le format d'entrée de VERSO pour être en mesure d'adapter le format de sortie de l'extracteur de métriques. Cette solution est efficace pour le développement de nouveaux composants à l'intérieur de la visualisation. Par contre, ce n'est pas la solution la plus efficace pour ce qui est de la convivialité.

À long terme, il serait préférable que l'approche englobe toutes les parties du projet de l'extraction de métriques jusqu'à la correction des anomalies dans le code. Il est même possible d'introduire un volet conception à VERSO. Il ne servirait plus seulement à la rétro-conception du logiciel, mais aussi à la conception et au développement de logiciels. Il serait possible de créer l'architecture des classses et d'entrer du code en naviguant à travers la visualisation offerte par VERSO. De cette façon, la visualisation serait présente de la conception jusqu'à la distribution du produit fini.

L'efficacité et la convivialité accrues encourageraient différents utilisateurs à s'en servir, ce qui amènerait inévitablement de nouvelles idées d'amélioration à l'approche dans son ensemble. Il serait aussi pertinent de diversifier les styles de visualisation pour

obtenir plus de renseignements sur un composant précis tout en utilisant la visualisation. On pense par exemple à la représentation des méthodes à l'intérieur des classes et à la représentation des instructions par la suite. Des vues différentes de celles de la qualité peuvent aussi être intégrées, par contre il faut toujours garder des liens forts entre les éléments communs des différentes vues pour éviter les discontinuités cognitives lors du passage d'une vue à l'autre.

6.3.4 Recherche d'anomalies précises

Une fois que notre approche est disponible pour visualiser les logiciels, il faut commencer à l'utiliser pour découvrir des anomalies précises qui permettront aux analystes de faire le réusinage de leur code. Une liste de différentes anomalies existe déjà dans la littérature. On pense entre autres aux *Code Smells* [21] et aux antipatrons de conception [8]. Il faut donc déterminer une série de correspondances entre des patrons graphiques et les anomalies du logiciel. Étant donné que l'analyste peut configurer les métriques affichées à sa guise, les possibilités sont quasiment infinies. Ce type d'anomalies est très complexe à rechercher à l'aide d'outils complètement automatiques parce que ces problèmes dépendent du contexte. L'humain est plus en mesure de comparer les éléments entre eux tout en tenant compte du contexte. Selon les caractéristiques du logiciel comme sa taille, sa complexité et sa fonction (pire caractéristique pour la détection automatique), il pourra prendre une décision juste. La décision prise est aussi en relation avec l'importance du logiciel et les critères de qualité requis pour un logiciel donné.

Un collègue, a déjà commencé à visualiser avec VERSO différentes anomalies à l'intérieur des logiciels. Jusqu'à maintenant nous avons obtenu de bons résultats pour trouver les anomalies : *Data Classes, Blob, Shotgun Surgery, Misplaced Class, God Package, Wide and shallow class hierarchies, Prolifération de paquetages et autres heuristiques de qualité*. Une liste de phénomènes logiciels impliquant l'évolution a déjà été discutée au chapitre 5. Il s'agit du *changement d'identité*, de la *protection des témoins*, de la *surcharge de responsabilité* et du *retour sur décision*.

La visualisation d'un logiciel offre une nouvelle perspective sur la qualité. En changeant la manière de voir des chercheurs, elle ouvre la porte à la découverte de nouvelles

causes pour les problèmes du logiciel et à de nouvelles constatations théoriques sur sa qualité. Le changement de point de vue met en lumière des liens et des éléments qui sont dissimulés avec d'autres techniques d'analyse.

6.3.5 Recherche de règles précises de qualité

Finalement, comme stipulé au début de ce mémoire, le génie logiciel est un domaine à théorie faible. À l'aide de VERSO, il est possible pour un analyste d'étudier plusieurs codes sources de plusieurs logiciels. De plus, contrairement à l'utilisation d'algorithmes automatiques, l'analyste peut naviguer à sa guise dans le logiciel sans but précis. Ce genre d'exploration est propice à la découverte de nouveaux patrons encore inconnus. L'écriture de règles de recherche qui seraient utilisées par des algorithmes automatiques est fastidieuse et l'exercice est complètement inapproprié pour trouver de nouvelles règles. Même l'apprentissage machine ne peut pas être utilisé pour de l'observation générale car des cas exemples doivent être donnés pour inférer les règles.

La visualisation est un moyen attrayant de regarder plusieurs phénomènes logiciels librement. De plus, le support graphique permet d'aider l'analyste à reconnaître ces phénomènes une fois qu'il les a observés. Une nomenclature imagée peut être dérivée des graphismes représentant le logiciel. Tout comme la métaphore, une telle nomenclature aide l'analyste à se souvenir des phénomènes et à les comprendre.

BIBLIOGRAPHIE

[1] Thomas Ball et Stephen G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996. ISSN 0018-9162.

[2] Michael Balzer, Oliver Deussen et Claus Lewerentz. Voronoi treemaps for the visualization of software metrics. Dans *Proceedings. ACM Symposium on Software Visualization*, pages 165–172, 2005. ISBN 1-59593-073-6.

[3] Victor R. Basili, Lionel C. Briand et Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transaction on Software Engineering*, 22(10):751–761, 1996. ISSN 0098-5589.

[4] Benjamin B. Bederson, Ben Shneiderman et Martin Wattenberg. Ordered and quantum treemaps : Making effective use of 2d space to display hierarchies. *ACM Transactions on Graphics*, 21(4):833–854, 2002. ISSN 0730-0301.

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

[6] Lionel C. Briand, John W. Daly et Jürgen Wüst. A unified framework for cohesion measurement in object-orientedsystems. *Empirical Software Engineering*, 3(1): 65–117, 1998. ISSN 1382-3256.

[7] Lionel C. Briand et Jürgen Wüst. Empirical studies of quality models in object-oriented systems. *Advances in Computers*, 59:97–166, 2002.

[8] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III et Thomas J. Mowbray. *AntiPatterns : Refactoring Software, Architectures, and Projects in Crisis*. John Wiley Press, 1998. ISBN 0-471-19713-0.

[9] Andrea Capiluppi. Models for the evolution of os projects. Dans *ICSM '03 : Proceedings of the International Conference on Software Maintenance*, pages 65–74, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1905-9.

[10] 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.

[11] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts et Kevin Wampler. A system for graph-based visualization of the evolution of software. Dans *Proceedings ACM Symposium on Software Visualization*, pages 77–86, 2003.

[12] 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, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2719-1.

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

[14] Serge Demeyer, Stephane Ducasse et Oscar Nierstrasz. *Object Oriented Reengineering Patterns*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1558606394. Foreword By-Ralph E. Johnson.

[15] Andreas Dieberger. A city metaphor to support navigation in complex information spaces. Dans *COSIT '97 : Proceedings of the International Conference on Spatial Information Theory*, pages 53–67, London, UK, 1997. Springer-Verlag. ISBN 3-540-63623-4.

[16] 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.

[17] Vadim Engelson, Hakan Larsson et Peter Fritzson. A design, simulation and visualization environment for object-oriented mechanical and multi-domain models in modelica. Dans *IV '99 : Proceedings of the 1999 International Conference on Information Visualisation*, pages 188–193, 1999.

[18] Jean-Daniel Fekete et Catherine Plaisant. Interactive information visualization of a million items. Dans *INFOVIS '02 : Proceedings of the IEEE Symposium on Information Visualization*, pages 117–124, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1751-X.

[19] Norman E. Fenton et Martin Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999. ISSN 0098-5589.

[20] Norman E. Fenton et Shari Lawrence Pfleeger. *Software Metrics : A Rigorous and Practical Approach*. Course Technology, 1998.

[21] Martin Fowler, Kent Beck, John Brant, William Opdyke et Don Roberts. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, 1999.

[22] 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, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8779-3.

[23] Hamish Graham, Hong Yul Yang et Rebecca Berrigan. A solar system metaphor for 3D visualisation of object oriented software metrics. Dans *Proceedings Australasian Symposium on Information Visualisation*, pages 53–59, 2004.

[24] R. L. Graham, B. D. Lubachevsky, K. J. Nurmela et P. R. J. Östergard. Dense packings of congruent circles in a circle. *Discrete Mathematics*, 181(1-3):139–154, 1998. ISSN 0012-365X.

[25] David Grosser, Houari A. Sahraoui et Petko Valtchev. An analogy-based approach for predicting design stability of java classes. Dans *METRICS '03 : Proceedings of the 9th International Symposium on Software Metrics*, pages 252–261, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1987-3.

[26] Yann-Gaël Guéhéneuc, Houari Sahraoui et Farouk Zaidi. Fingerprinting design patterns. Dans *Proceedings Working Conference on Reverse Engineering*, pages 172–181, November 2004.

[27] Yann-Gaël Guéhéneuc et Hervé Albin-Amiot. Recovering binary class relationships : putting icing on the uml cake. *SIGPLAN Notices*, 39(10):301–314, 2004. ISSN 0362-1340.

[28] Christopher G. Healey. Choosing effective colours for data visualization. Dans *VIS '96 : Proceedings of the 7th conference on Visualization '96*, pages 263–270., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press. ISBN 0-89791-864-9.

[29] Christopher G. Healey, Kellogg S. Booth et James T. Enns. Harnessing preattentive processes for multivariate data visualization. Rapport technique, Vancouver, BC, Canada, Canada, 1992.

[30] Christopher G. Healey et James T. Enns. Building perceptual textures to visualize multidimensional datasets. Dans *VIS '98 : Proceedings of the conference on Visualization '98*, pages 111–118, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press. ISBN 1-58113-106-2.

[31] 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. ISSN 1077-2626.

[32] Danny Holten, Roel Vliegen et Jarke J. van Wijk. Visual realism for the visualization of software metrics. Dans *Proceedings International Workshop on Visualizing Software for Understanding and Analysis*, pages 27–32, 2005.

[33] D. E. Hubert et Christopher G. Healey. Visualizing data with motion. Dans *Proceedings of IEEE Visualization 2005*, pages 527–534, Minneapolis, Minnesota, USA, 2005. IEEE Computer Society Press.

[34] Daniel Jackson et Martin Rinard. Software analysis : a roadmap. Dans *ICSE '00 : Proceedings of the Conference on The Future of Software Engineering*, pages 133–145, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-253-0.

[35] 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, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press. ISBN 0-8186-2245-8 (PAPER).

[36] Frederick P. Brooks Jr. No silver bullet : essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987. ISSN 0018-9162.

[37] Chris F. Kemerer et Sandra Slaughter. An empirical approach to studying software evolution. *IEEE Transactions Software Engineering*, 25(4):493–509, 1999. ISSN 0098-5589.

[38] Claire Knight et Malcolm Munro. Comprehension with[in] virtual environment visualisations. Dans *IWPC '99 : Proceedings of the 7th International Workshop on Program Comprehension*, pages 4–11, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0179-6.

[39] Claire Knight et Malcolm Munro. Virtual but visible software. Dans *Proceedings International Conference on Information Visualisation*, pages 198–205, July 2000.

[40] Guillaume Langelier, Houari Sahraoui et Pierre Poulin. Visualisation and analysis of software quantitative data. Dans *Proceedings 9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE)*, 2005.

[41] 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, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-993-4.

[42] Guillaume Langelier, Houari Sahraoui et Pierre Poulin. Animation coherence in representing software evolution. Dans *Proceedings 10th ECOOP Workshop on*

Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE), 2006.

[43] Guillaume Langelier, Houari Sahraoui et Pierre Poulin. Visualisation du logiciel et de son évolution. Dans *Acte Atelier sur l'évolution du logiciel(AEL)*, 2006.

[44] Guillaume Langelier, Houari A. Sahraoui et Pierre Poulin. Visualisation et analyse de logiciels de grande taille. Dans *Langages et Modèles à Objets 2005*, pages 159–173, mars 2005.

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

[46] Michele Lanza et Stéphane Ducasse. Polymetric views–a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9): 782–795, September 2003.

[47] M M. Lehman, J F. Ramil, P D. Wernick, D E. Perry et W M. Turski. Metrics and laws of software evolution - the nineties view. Dans *METRICS '97 : Proceedings of the 4th International Symposium on Software Metrics*, page 20, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8093-8.

[48] Wei Li et Sallie Henry. Object-oriented metrics that predict maintainability. *Journal on Systems and Software*, 23(2):111–122, 1993. ISSN 0164-1212.

[49] B. P. Lientz, E. B. Swanson et G. E. Tompkins. Characteristics of application software maintenance. *Communication of ACM*, 21(6):466–471, 1978. ISSN 0001-0782.

[50] K. Liu, S. Zhou et H. Yang. Quality metrics of object oriented design for software development and re-development. Dans *APAQS '00 : Proceedings of the The First Asia-Pacific Conference on Quality Software*, pages 127–135, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0825-1.

[51] Gerard Lommerse, Freek Nossin, Lucian Voinea et Alexandru Telea. The visual code navigator : An interactive toolset for source code investigation. Dans *Proceedings IEEE Symposium on Information Visualization*, pages 25–32, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7803-9464-x.

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

[53] Y. Mao, H. Sahraoui et H. Lounis. Reusability hypothesis verification using machine learning techniques : A case study. Dans *ASE '98 : Proceedings of the 13th IEEE international conference on Automated software engineering*, pages 84–93, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8750-9.

[54] Andrian Marcus, Louis Feng et Jonathan I. Maletic. 3D representations for software visualization. Dans *SoftVis '03 : Proceedings of 2003 ACM symposium on Software visualization*, pages 27–36, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-642-0.

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

[56] Cédric Mesnage et Michele Lanza. White coats : Web-visualization of evolving software in 3d. Dans *Proceedings IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 40–45, 2005.

[57] Barbara Mirel. *Interaction Design for Complex Problem Solving*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 1558608311.

[58] Quang Vinh Nguyen et Mao Lin Huang. A space-optimized tree visualization. Dans *Proceedings IEEE Symposium on Information Visualization*, pages 85–92, 2002.

[59] *UML 2.0 Superstructure Specification*. Object Management Group, Octobre 2004.

[60] Alessandro Orso, James Jones et Mary Jean Harrold. Visualization of program-execution data for deployed software. Dans *Proceedings ACM Symposium on Software Visualization*, pages 67–76, 2003. ISBN 1-58113-642-0.

[61] S.E. Palmer. *Vision Science : Photons to Phenomenology*. The MIT Press, 1999.

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

[63] Martin Pinzger, Harald Gall, Michael Fischer et Michele Lanza. Visualizing multiple evolution metrics. Dans *Proceedings ACM Symposium on Software Visualization*, pages 67–75, 2005.

[64] V. S. Ramachandran et William Herstein. The science of art : A neurological theory of aesthetic experience. *Journal of Consciousness Studies*, 6(6):15–51, June 1999.

[65] Juan F. Ramil et Meir M. Lehman. Defining and applying metrics in the context of continuing software evolution. Dans *METRICS '01 : Proceedings of the 7th International Symposium on Software Metrics*, pages 199–209, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1043-4.

[66] Steve P. Reiss et Manos Renieris. Chapter 11 : The bloom software visualization system. Dans Kang Zhang, éditeur, *Software Visualization : From Theory to Practice*, pages 311–358. Kluwer Academic Publishers, 2003.

[67] Steven P. Reiss. Visualizing java in action. Dans *SoftVis '03 : Proceedings of the 2003 ACM symposium on Software visualization*, pages 57–65, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-642-0.

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

[69] Mohamed Rouabi. Analyse de la qualité du logiciel : une approche par visualisation et simulation. Mémoire de maîtrise, Université de Montréal, 2005.

[70] H. A. Sahraoui, A. M. Boukadoum, H. Lounis et F. Etheve. Predicting class libraries interface evolution : an investigation into machine learning approaches. Dans *APSEC '00 : Proceedings of the Seventh Asia-Pacific Software Engineering Conference*, pages 456–464, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0915-0.

[71] Marc L. Schnitzer et Mark A. Pedreira. A neuropsychological theory of metaphor. *Language Sciences*, 27(1):31–49, January 2005.

[72] Ben Shneiderman. Treemap. URL `http ://www.cs.umd.edu/hcil/treemap/index.shtml`.

[73] John Stasko. An evaluation of space-filling information visualizations for depicting hierarchical structures. *International Journal on Human-Computation Studies*, 53 (5):663–694, 2000. ISSN 1071-5819.

[74] John Stasko et Eugene Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. Dans *INFOVIS '00 : Proceedings of the IEEE Symposium on Information Vizualization 2000*, page 57, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0804-9.

[75] Roger C. Tam, Christopher G. Healey, Borys Flak et Peter Cahoon. Volume rendering of abdominal aortic aneurysms. Dans *IEEE Visualization*, pages 43–50, 1997.

[76] Maurice Termeer, Christian Lange, Alexandru Telea et Michel Chaudron. Visual exploration of combined architectural and metric information. Dans *Proceedings 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2005.

[77] Edward R. Tufte. *Visual explanations : images and quantities, evidence and narrative*. Graphics Press, Cheshire, CT, USA, 1997. ISBN 0-9613921-2-6.

[78] S. L. Voinea et Alexdru Telea. A file-based visualization of software evolution. Dans *Proceedings ASCI*, 2006.

[79] Jarke J. Van Wijk et Huub van de Wetering. Cushion treemaps : Visualization of hierarchical information. Dans *INFOVIS '99 : Proceedings of the 1999 IEEE Symposium on Information Visualization*, pages 73–78, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0431-0.

[80] Jingwei Wu, Richard C. Holt et Ahmed E. Hassan. Exploring software evolution using spectrographs. Dans *Proceedings Working Conference on Reverse Engineering*, pages 80–89, 2004. ISBN 0-7695-2243-2.

[81] Stephen S. Yau et Jeffery J.-P. Tsai. A survey of software design techniques. *IEEE Transaction on Software Engineering*, 12(6):713–721, 1986. ISSN 0098-5589.

Annexe I

Questionnaire pour l'expérience de comparaison des algorithmes de placement

1. **Abbot** : Retrouvez une classe qui est seule dans son paquetage.
2. **Abbot et ses librairies** : Retrouvez le paquetage où une très grande classe très peu couplée est réunie avec des petites classes peu couplées aussi.
3. **Art of Illusion** : Retrouvez un paquetage comportant plus de 10 classes et aucune classe fortement couplée.
4. **BeanBrowser96** : Retrouvez un paquetage contenant trois petites classes très fortement couplées.
5. **Borg** : Retrouvez le paquetage le plus couplé.
6. **CfParse** : Retrouvez la classe la plus couplée du système.
7. **Emma** : Retrouvez une classe seule qui se retrouve assez bas dans l'architecutre.
8. **FindBugs** : Retrouvez un paquetage de plus de 10 classes très peu couplé.
9. **FindBugs et ses librairies** : Retrouvez un paquetage se trouvant au premier niveau de la hiérarchie qui contient seulement quelques classes.
10. **Freemind** : Retrouvez une grande classe très peu couplée.
11. **GanttProject** : Parmi les sous-paquetages de net.sourceforge.ganttproject, lequel vous semble le plus couplé ?
12. **Hsqldb** : Retrouvez le paquetage le plus profond.
13. **Ibatis** : Deux paquetages de plus de quatre classes présentent des classes avec une cohésion parfaite, trouvez celui qui possède un sous-paquetage avec les mêmes caractéristiques.
14. **IReport** : Recherchez le paquetage qui vous semble être le noyau (core) du programme.
15. **Jalopy** : Recherchez le paquetage le plus couplé.

16. **JDK 1.1** : Trouvez un paquetage de plus de 20 classes affichant une cohésion parfaite.
17. **Jedit** : Recherchez une classe qui est seule à être très couplée dans son sous-paquetage. Il y a aussi peu de classes dans son paquetage.
18. **PcGen** : Repérez le core(noyau) de cette application.
19. **Jgpd** : La classe recherchée présente une cohésion parfaite malgré sa grande taille. Elle n'est pas seule dans son paquetage non plus.
20. **MegaMek** : On cherche le plus grand paquetage se trouvant au troisième niveau de l'architecture.

Annexe II

Résultats complets pour les questions automatiques de l'expérience sur le placement

Tableau II.1 – Entrées chronométrées de l'expérience du placement

goupe	numéro	fichier	typevis	temps
profileA1	8	C :/Systemes/findbugs.xml	Sunburst	8.642513
profileA1	12	C :/Systemes/hsqldb.xml	TreeLine	82.44938
profileA1	4	C :/Systemes/beanBrowser96.xml	TreeMap	22.49257
profileA1	14	C :/Systemes/iReport.xml	Sunburst	3.635264
profileA1	10	C :/Systemes/freemind.xml	TreeMap	2.653842
profileA1	13	C :/Systemes/Ibatis.xml	TreeMap	44.88499
profileA1	2	C :/Systemes/AbbotEtLib.xml	Sunburst	29.26237
profileA1	7	C :/Systemes/emma.xml	TreeMap	12.02741
profileA1	3	C :/Systemes/ArtOfIllusion.xml	TreeLine	44.99515
profileA1	16	C :/Systemes/jdk1.1.xml	TreeMap	10.38504
profileA1	20	C :/Systemes/MegaMek.xml	Sunburst	23.50403
profileA1	5	C :/Systemes/borg.xml	Sunburst	2.073002
profileA1	17	C :/Systemes/jedit.xml	Sunburst	8.932935
profileA1	6	C :/Systemes/cfparse.xml	TreeLine	4.226119
profileA1	1	C :/Systemes/abbot.xml	TreeMap	7.891426
profileA1	9	C :/Systemes/FindBugsEtLib.xml	TreeLine	80.8771
profileA1	15	C :/Systemes/jalopy-1.0b11.xml	TreeLine	7.530904
profileA1	19	C :/Systemes/jgpd.xml	TreeMap	7.100281
profileA1	11	C :/Systemes/ganttproject-1.9.11.xml	Sunburst	6.128874

goupe	numéro	fichier	typevis	temps
profileA1	18	C :/Systemes/pcgen.xml	TreeLine	9.083151
profileA2	19	C :/Systemes/jgpd.xml	TreeMap	10.7756
profileA2	3	C :/Systemes/ArtOfIllusion.xml	TreeLine	76.14024
profileA2	7	C :/Systemes/emma.xml	TreeMap	31.78602
profileA2	16	C :/Systemes/jdk1.1.xml	TreeMap	25.65715
profileA2	17	C :/Systemes/jedit.xml	Sunburst	99.21365
profileA2	13	C :/Systemes/Ibatis.xml	TreeMap	66.69657
profileA2	18	C :/Systemes/pcgen.xml	TreeLine	42.99225
profileA2	14	C :/Systemes/iReport.xml	Sunburst	16.92451
profileA2	12	C :/Systemes/hsqldb.xml	TreeLine	24.44539
profileA2	8	C :/Systemes/findbugs.xml	Sunburst	9.443673
profileA2	10	C :/Systemes/freemind.xml	TreeMap	3.905655
profileA2	20	C :/Systemes/MegaMek.xml	Sunburst	14.73133
profileA2	5	C :/Systemes/borg.xml	Sunburst	2.51364
profileA2	11	C :/Systemes/ganttproject-1.9.11.xml	Sunburst	7.560947
profileA2	6	C :/Systemes/cfparse.xml	TreeLine	5.097381
profileA2	9	C :/Systemes/FindBugsEtLib.xml	TreeLine	28.38109
profileA2	4	C :/Systemes/beanBrowser96.xml	TreeMap	22.62276
profileA2	1	C :/Systemes/abbot.xml	TreeMap	8.772702
profileA2	2	C :/Systemes/AbbotEtLib.xml	Sunburst	45.15538
profileA2	15	C :/Systemes/jalopy-1.0b11.xml	TreeLine	2.363422
profileA3	9	C :/Systemes/FindBugsEtLib.xml	TreeLine	131.9911
profileA3	17	C :/Systemes/jedit.xml	Sunburst	11.05601
profileA3	12	C :/Systemes/hsqldb.xml	TreeLine	128.8065
profileA3	14	C :/Systemes/iReport.xml	Sunburst	30.82463
profileA3	4	C :/Systemes/beanBrowser96.xml	TreeMap	364.8583
profileA3	5	C :/Systemes/borg.xml	Sunburst	1.942813

goupe	numéro	fichier	typevis	temps
profileA3	6	C :/Systemes/cfparse.xml	TreeLine	3.565162
profileA3	16	C :/Systemes/jdk1.1.xml	TreeMap	22.7229
profileA3	15	C :/Systemes/jalopy-1.0b11.xml	TreeLine	1.60232
profileA3	20	C :/Systemes/MegaMek.xml	Sunburst	24.07486
profileA3	10	C :/Systemes/freemind.xml	TreeMap	7.52089
profileA3	13	C :/Systemes/Ibatis.xml	TreeMap	14.7814
profileA3	19	C :/Systemes/jgpd.xml	TreeMap	55.7407
profileA3	3	C :/Systemes/ArtOfIllusion.xml	TreeLine	195.1025
profileA3	7	C :/Systemes/emma.xml	TreeMap	65.25449
profileA3	8	C :/Systemes/findbugs.xml	Sunburst	24.866
profileA3	18	C :/Systemes/pcgen.xml	TreeLine	5.397816
profileA3	2	C :/Systemes/AbbotEtLib.xml	Sunburst	40.40851
profileA3	1	C :/Systemes/abbot.xml	TreeMap	12.17763
profileA3	11	C :/Systemes/ganttproject-1.9.11.xml	Sunburst	11.07604
profileA4	19	C :/Systemes/jgpd.xml	TreeMap	140.4834
profileA4	18	C :/Systemes/pcgen.xml	TreeLine	11.57676
profileA4	9	C :/Systemes/FindBugsEtLib.xml	TreeLine	47.22838
profileA4	3	C :/Systemes/ArtOfIllusion.xml	TreeLine	237.1433
profileA4	1	C :/Systemes/abbot.xml	TreeMap	15.09185
profileA4	6	C :/Systemes/cfparse.xml	TreeLine	4.085916
profileA4	2	C :/Systemes/AbbotEtLib.xml	Sunburst	7.791281
profileA4	13	C :/Systemes/Ibatis.xml	TreeMap	177.2967
profileA4	12	C :/Systemes/hsqldb.xml	TreeLine	98.95327
profileA4	14	C :/Systemes/iReport.xml	Sunburst	45.67614
profileA4	16	C :/Systemes/jdk1.1.xml	TreeMap	50.7635
profileA4	5	C :/Systemes/borg.xml	Sunburst	2.89419
profileA4	11	C :/Systemes/ganttproject-1.9.11.xml	Sunburst	16.79432

goupe	numéro	fichier	typevis	temps
profileA4	20	C :/Systemes/MegaMek.xml	Sunburst	54.05827
profileA4	15	C :/Systemes/jalopy-1.0b11.xml	TreeLine	3.134538
profileA4	4	C :/Systemes/beanBrowser96.xml	TreeMap	13.13902
profileA4	8	C :/Systemes/findbugs.xml	Sunburst	9.914355
profileA4	17	C :/Systemes/jedit.xml	Sunburst	100.0749
profileA4	7	C :/Systemes/emma.xml	TreeMap	93.29508
profileA4	10	C :/Systemes/freemind.xml	TreeMap	4.6968
profileA5	15	C :/Systemes/jalopy-1.0b11.xml	TreeLine	25.81764
profileA5	8	C :/Systemes/findbugs.xml	Sunburst	42.92258
profileA5	17	C :/Systemes/jedit.xml	Sunburst	77.65321
profileA5	12	C :/Systemes/hsqldb.xml	TreeLine	137.0999
profileA5	10	C :/Systemes/freemind.xml	TreeMap	2.543708
profileA5	20	C :/Systemes/MegaMek.xml	Sunburst	23.88482
profileA5	13	C :/Systemes/Ibatis.xml	TreeMap	107.0961
profileA5	2	C :/Systemes/AbbotEtLib.xml	Sunburst	47.92987
profileA5	1	C :/Systemes/abbot.xml	TreeMap	18.35676
profileA5	18	C :/Systemes/pcgen.xml	TreeLine	15.73294
profileA5	16	C :/Systemes/jdk1.1.xml	TreeMap	49.5222
profileA5	19	C :/Systemes/jgpd.xml	TreeMap	24.78613
profileA5	14	C :/Systemes/iReport.xml	Sunburst	16.95472
profileA5	3	C :/Systemes/ArtOfIllusion.xml	TreeLine	109.6961
profileA5	6	C :/Systemes/cfparse.xml	TreeLine	4.446172
profileA5	7	C :/Systemes/emma.xml	TreeMap	30.34212
profileA5	9	C :/Systemes/FindBugsEtLib.xml	TreeLine	90.11508
profileA5	11	C :/Systemes/ganttproject-1.9.11.xml	Sunburst	19.58719
profileA5	4	C :/Systemes/beanBrowser96.xml	TreeMap	114.3019
profileA5	5	C :/Systemes/borg.xml	Sunburst	2.363446

goupe	numéro	fichier	typevis	temps
profileB1	7	C :/Systemes/emma.xml	TreeLine	93.15488
profileB1	9	C :/Systemes/FindBugsEtLib.xml	Sunburst	178.7588
profileB1	10	C :/Systemes/freemind.xml	TreeLine	35.42129
profileB1	8	C :/Systemes/findbugs.xml	TreeMap	24.76586
profileB1	16	C :/Systemes/jdk1.1.xml	TreeLine	40.99936
profileB1	1	C :/Systemes/abbot.xml	TreeLine	27.69009
profileB1	13	C :/Systemes/Ibatis.xml	TreeLine	58.4346
profileB1	6	C :/Systemes/cfparse.xml	Sunburst	3.555148
profileB1	14	C :/Systemes/iReport.xml	TreeMap	5.317699
profileB1	20	C :/Systemes/MegaMek.xml	TreeMap	25.58705
profileB1	12	C :/Systemes/hsqldb.xml	Sunburst	10.48518
profileB1	4	C :/Systemes/beanBrowser96.xml	TreeLine	17.31507
profileB1	17	C :/Systemes/jedit.xml	TreeMap	20.69997
profileB1	11	C :/Systemes/ganttproject-1.9.11.xml	TreeMap	22.39242
profileB1	19	C :/Systemes/jgpd.xml	TreeLine	24.62566
profileB1	18	C :/Systemes/pcgen.xml	Sunburst	7.490847
profileB1	2	C :/Systemes/AbbotEtLib.xml	TreeMap	14.67124
profileB1	3	C :/Systemes/ArtOfIllusion.xml	Sunburst	20.55977
profileB1	15	C :/Systemes/jalopy-1.0b11.xml	Sunburst	4.766902
profileB1	5	C :/Systemes/borg.xml	TreeMap	6.789831
profileB2	14	C :/Systemes/iReport.xml	TreeMap	35.59153
profileB2	4	C :/Systemes/beanBrowser96.xml	TreeLine	147.123
profileB2	19	C :/Systemes/jgpd.xml	TreeLine	413.9144
profileB2	6	C :/Systemes/cfparse.xml	Sunburst	5.238002
profileB2	20	C :/Systemes/MegaMek.xml	TreeMap	22.35415
profileB2	13	C :/Systemes/Ibatis.xml	TreeLine	263.3345
profileB2	18	C :/Systemes/pcgen.xml	Sunburst	11.08616

goupe	numéro	fichier	typevis	temps
profileB2	1	C :/Systemes/abbot.xml	TreeLine	67.86895
profileB2	8	C :/Systemes/findbugs.xml	TreeMap	5.427913
profileB2	11	C :/Systemes/ganttproject-1.9.11.xml	TreeMap	20.84038
profileB2	12	C :/Systemes/hsqldb.xml	Sunburst	152.5614
profileB2	10	C :/Systemes/freemind.xml	TreeLine	14.32073
profileB2	17	C :/Systemes/jedit.xml	TreeMap	27.53988
profileB2	16	C :/Systemes/jdk1.1.xml	TreeLine	226.6982
profileB2	2	C :/Systemes/AbbotEtLib.xml	TreeMap	69.4806
profileB2	9	C :/Systemes/FindBugsEtLib.xml	Sunburst	262.5702
profileB2	15	C :/Systemes/jalopy-1.0b11.xml	Sunburst	8.892876
profileB2	3	C :/Systemes/ArtOfIllusion.xml	Sunburst	50.62329
profileB2	7	C :/Systemes/emma.xml	TreeLine	7.861382
profileB2	5	C :/Systemes/borg.xml	TreeMap	1.972856
profileB3	2	C :/Systemes/AbbotEtLib.xml	TreeMap	17.75553
profileB3	10	C :/Systemes/freemind.xml	TreeLine	69.27962
profileB3	18	C :/Systemes/pcgen.xml	Sunburst	221.839
profileB3	1	C :/Systemes/abbot.xml	TreeLine	59.58568
profileB3	7	C :/Systemes/emma.xml	TreeLine	41.48966
profileB3	19	C :/Systemes/jgpd.xml	TreeLine	310.9163
profileB3	20	C :/Systemes/MegaMek.xml	TreeMap	51.37235
profileB3	14	C :/Systemes/iReport.xml	TreeMap	6.310521
profileB3	11	C :/Systemes/ganttproject-1.9.11.xml	TreeMap	25.80302
profileB3	4	C :/Systemes/beanBrowser96.xml	TreeLine	221.1464
profileB3	3	C :/Systemes/ArtOfIllusion.xml	Sunburst	28.94162
profileB3	15	C :/Systemes/jalopy-1.0b11.xml	Sunburst	4.176005
profileB3	16	C :/Systemes/jdk1.1.xml	TreeLine	32.95739
profileB3	13	C :/Systemes/Ibatis.xml	TreeLine	42.14059

goupe	numéro	fichier	typevis	temps
profileB3	9	C :/Systemes/FindBugsEtLib.xml	Sunburst	31.09471
profileB3	8	C :/Systemes/findbugs.xml	TreeMap	8.211808
profileB3	6	C :/Systemes/cfparse.xml	Sunburst	43.85306
profileB3	17	C :/Systemes/jedit.xml	TreeMap	31.51532
profileB3	5	C :/Systemes/borg.xml	TreeMap	11.0559
profileB3	12	C :/Systemes/hsqldb.xml	Sunburst	35.36084
profileB4	1	C :/Systemes/abbot.xml	TreeLine	44.43833
profileB4	14	C :/Systemes/iReport.xml	TreeMap	24.78811
profileB4	4	C :/Systemes/beanBrowser96.xml	TreeLine	162.3142
profileB4	10	C :/Systemes/freemind.xml	TreeLine	5.037293
profileB4	6	C :/Systemes/cfparse.xml	Sunburst	4.446438
profileB4	7	C :/Systemes/emma.xml	TreeLine	35.92201
profileB4	5	C :/Systemes/borg.xml	TreeMap	12.6383
profileB4	18	C :/Systemes/pcgen.xml	Sunburst	13.50956
profileB4	2	C :/Systemes/AbbotEtLib.xml	TreeMap	35.3612
profileB4	3	C :/Systemes/ArtOfIllusion.xml	Sunburst	55.22997
profileB4	12	C :/Systemes/hsqldb.xml	Sunburst	26.76876
profileB4	19	C :/Systemes/jgpd.xml	TreeLine	262.7805
profileB4	13	C :/Systemes/Ibatis.xml	TreeLine	113.8448
profileB4	20	C :/Systemes/MegaMek.xml	TreeMap	53.91807
profileB4	17	C :/Systemes/jedit.xml	TreeMap	23.03335
profileB4	15	C :/Systemes/jalopy-1.0b11.xml	Sunburst	8.10173
profileB4	16	C :/Systemes/jdk1.1.xml	TreeLine	93.92599
profileB4	9	C :/Systemes/FindBugsEtLib.xml	Sunburst	35.92201
profileB4	8	C :/Systemes/findbugs.xml	TreeMap	11.43656
profileB4	11	C :/Systemes/ganttproject-1.9.11.xml	TreeMap	50.20269
profileB5	13	C :/Systemes/Ibatis.xml	TreeLine	107.8072

goupe	numéro	fichier	typevis	temps
profileB5	12	C :/Systemes/hsqldb.xml	Sunburst	25.64739
profileB5	3	C :/Systemes/ArtOfIllusion.xml	Sunburst	45.92696
profileB5	5	C :/Systemes/borg.xml	TreeMap	9.313578
profileB5	16	C :/Systemes/jdk1.1.xml	TreeLine	35.64196
profileB5	10	C :/Systemes/freemind.xml	TreeLine	12.70853
profileB5	8	C :/Systemes/findbugs.xml	TreeMap	7.861461
profileB5	20	C :/Systemes/MegaMek.xml	TreeMap	52.16605
profileB5	11	C :/Systemes/ganttproject-1.9.11.xml	TreeMap	13.74003
profileB5	6	C :/Systemes/cfparse.xml	Sunburst	3.184643
profileB5	19	C :/Systemes/jgpd.xml	TreeLine	125.7333
profileB5	18	C :/Systemes/pcgen.xml	Sunburst	23.85478
profileB5	9	C :/Systemes/FindBugsEtLib.xml	Sunburst	9.934484
profileB5	15	C :/Systemes/jalopy-1.0b11.xml	Sunburst	6.219067
profileB5	2	C :/Systemes/AbbotEtLib.xml	TreeMap	12.26789
profileB5	14	C :/Systemes/iReport.xml	TreeMap	7.260585
profileB5	4	C :/Systemes/beanBrowser96.xml	TreeLine	31.42581
profileB5	1	C :/Systemes/abbot.xml	TreeLine	16.44397
profileB5	17	C :/Systemes/jedit.xml	TreeMap	47.11869
profileB5	7	C :/Systemes/emma.xml	TreeLine	15.07197
profileC1	5	C :/Systemes/borg.xml	TreeLine	7.110366
profileC1	15	C :/Systemes/jalopy-1.0b11.xml	TreeMap	7.681199
profileC1	14	C :/Systemes/iReport.xml	TreeLine	107.7371
profileC1	11	C :/Systemes/ganttproject-1.9.11.xml	TreeLine	56.15186
profileC1	13	C :/Systemes/Ibatis.xml	Sunburst	59.8172
profileC1	20	C :/Systemes/MegaMek.xml	TreeLine	33.2685
profileC1	17	C :/Systemes/jedit.xml	TreeLine	121.7074
profileC1	8	C :/Systemes/findbugs.xml	TreeLine	18.81743

goupe	numéro	fichier	typevis	temps
profileC1	2	C :/Systemes/AbbotEtLib.xml	TreeLine	72.77609
profileC1	4	C :/Systemes/beanBrowser96.xml	Sunburst	26.76903
profileC1	6	C :/Systemes/cfparse.xml	TreeMap	3.995826
profileC1	3	C :/Systemes/ArtOfIllusion.xml	TreeMap	12.07761
profileC1	19	C :/Systemes/jgpd.xml	Sunburst	63.26223
profileC1	1	C :/Systemes/abbot.xml	Sunburst	5.708322
profileC1	9	C :/Systemes/FindBugsEtLib.xml	TreeMap	38.89671
profileC1	18	C :/Systemes/pcgen.xml	TreeMap	40.34882
profileC1	12	C :/Systemes/hsqldb.xml	TreeMap	22.12225
profileC1	10	C :/Systemes/freemind.xml	Sunburst	4.366365
profileC1	16	C :/Systemes/jdk1.1.xml	Sunburst	9.553928
profileC1	7	C :/Systemes/emma.xml	Sunburst	12.95889
profileC2	4	C :/Systemes/beanBrowser96.xml	Sunburst	71.8941
profileC2	13	C :/Systemes/Ibatis.xml	Sunburst	465.2536
profileC2	16	C :/Systemes/jdk1.1.xml	Sunburst	154.033
profileC2	8	C :/Systemes/findbugs.xml	TreeLine	132.9625
profileC2	1	C :/Systemes/abbot.xml	Sunburst	62.09992
profileC2	14	C :/Systemes/iReport.xml	TreeLine	20.1692
profileC2	2	C :/Systemes/AbbotEtLib.xml	TreeLine	375.5438
profileC2	5	C :/Systemes/borg.xml	TreeLine	17.87588
profileC2	9	C :/Systemes/FindBugsEtLib.xml	TreeMap	267.0166
profileC2	19	C :/Systemes/jgpd.xml	Sunburst	248.2795
profileC2	12	C :/Systemes/hsqldb.xml	TreeMap	29.39256
profileC2	17	C :/Systemes/jedit.xml	TreeLine	45.32563
profileC2	18	C :/Systemes/pcgen.xml	TreeMap	10.05456
profileC2	11	C :/Systemes/ganttproject-1.9.11.xml	TreeLine	7.580977
profileC2	10	C :/Systemes/freemind.xml	Sunburst	25.76731

goupe	numéro	fichier	typevis	temps
profileC2	7	C :/Systemes/emma.xml	Sunburst	29.66295
profileC2	20	C :/Systemes/MegaMek.xml	TreeLine	85.58392
profileC2	6	C :/Systemes/cfparse.xml	TreeMap	6.839903
profileC2	3	C :/Systemes/ArtOfIllusion.xml	TreeMap	167.1921
profileC2	15	C :/Systemes/jalopy-1.0b11.xml	TreeMap	13.96021
profileC3	12	C :/Systemes/hsqldb.xml	TreeMap	14.58097
profileC3	20	C :/Systemes/MegaMek.xml	TreeLine	50.19217
profileC3	5	C :/Systemes/borg.xml	TreeLine	4.967143
profileC3	16	C :/Systemes/jdk1.1.xml	Sunburst	238.1124
profileC3	17	C :/Systemes/jedit.xml	TreeLine	63.32105
profileC3	11	C :/Systemes/ganttproject-1.9.11.xml	TreeLine	110.7893
profileC3	1	C :/Systemes/abbot.xml	Sunburst	19.60819
profileC3	13	C :/Systemes/Ibatis.xml	Sunburst	80.93639
profileC3	18	C :/Systemes/pcgen.xml	TreeMap	31.28499
profileC3	9	C :/Systemes/FindBugsEtLib.xml	TreeMap	37.50393
profileC3	6	C :/Systemes/cfparse.xml	TreeMap	2.553672
profileC3	15	C :/Systemes/jalopy-1.0b11.xml	TreeMap	2.623773
profileC3	14	C :/Systemes/iReport.xml	TreeLine	1.902736
profileC3	7	C :/Systemes/emma.xml	Sunburst	14.1203
profileC3	2	C :/Systemes/AbbotEtLib.xml	TreeLine	187.3394
profileC3	4	C :/Systemes/beanBrowser96.xml	Sunburst	25.36648
profileC3	10	C :/Systemes/freemind.xml	Sunburst	4.085875
profileC3	3	C :/Systemes/ArtOfIllusion.xml	TreeMap	181.4609
profileC3	19	C :/Systemes/jgpd.xml	Sunburst	459.8808
profileC3	8	C :/Systemes/findbugs.xml	TreeLine	32.82863
profileC4	4	C :/Systemes/beanBrowser96.xml	Sunburst	17.92613
profileC4	12	C :/Systemes/hsqldb.xml	TreeMap	34.04964

goupe	numéro	fichier	typevis	temps
profileC4	1	C :/Systemes/abbot.xml	Sunburst	4.63676
profileC4	8	C :/Systemes/findbugs.xml	TreeLine	42.40182
profileC4	16	C :/Systemes/jdk1.1.xml	Sunburst	50.57373
profileC4	9	C :/Systemes/FindBugsEtLib.xml	TreeMap	64.60419
profileC4	15	C :/Systemes/jalopy-1.0b11.xml	TreeMap	4.827037
profileC4	11	C :/Systemes/ganttproject-1.9.11.xml	TreeLine	7.991651
profileC4	10	C :/Systemes/freemind.xml	Sunburst	39.67785
profileC4	19	C :/Systemes/jgpd.xml	Sunburst	139.2931
profileC4	5	C :/Systemes/borg.xml	TreeLine	2.643854
profileC4	13	C :/Systemes/Ibatis.xml	Sunburst	227.8922
profileC4	14	C :/Systemes/iReport.xml	TreeLine	21.16085
profileC4	17	C :/Systemes/jedit.xml	TreeLine	147.5151
profileC4	3	C :/Systemes/ArtOfIllusion.xml	TreeMap	25.27685
profileC4	20	C :/Systemes/MegaMek.xml	TreeLine	13.92029
profileC4	18	C :/Systemes/pcgen.xml	TreeMap	7.430834
profileC4	7	C :/Systemes/emma.xml	Sunburst	69.53136
profileC4	6	C :/Systemes/cfparse.xml	TreeMap	1.74254
profileC4	2	C :/Systemes/AbbotEtLib.xml	TreeLine	69.59145
profileC5	3	C :/Systemes/ArtOfIllusion.xml	TreeMap	49.77959
profileC5	2	C :/Systemes/AbbotEtLib.xml	TreeLine	83.81718
profileC5	14	C :/Systemes/iReport.xml	TreeLine	13.74922
profileC5	4	C :/Systemes/beanBrowser96.xml	Sunburst	31.4558
profileC5	19	C :/Systemes/jgpd.xml	Sunburst	31.09534
profileC5	5	C :/Systemes/borg.xml	TreeLine	2.363446
profileC5	20	C :/Systemes/MegaMek.xml	TreeLine	26.76903
profileC5	18	C :/Systemes/pcgen.xml	TreeMap	31.01522
profileC5	6	C :/Systemes/cfparse.xml	TreeMap	2.63384

goupe	numéro	fichier	typevis	temps
profileC5	13	C :/Systemes/Ibatis.xml	Sunburst	44.68515
profileC5	10	C :/Systemes/freemind.xml	Sunburst	5.017314
profileC5	11	C :/Systemes/ganttproject-1.9.11.xml	TreeLine	25.06654
profileC5	12	C :/Systemes/hsqldb.xml	TreeMap	11.26643
profileC5	7	C :/Systemes/emma.xml	Sunburst	15.05194
profileC5	1	C :/Systemes/abbot.xml	Sunburst	5.858541
profileC5	17	C :/Systemes/jedit.xml	TreeLine	25.01647
profileC5	8	C :/Systemes/findbugs.xml	TreeLine	12.60838
profileC5	15	C :/Systemes/jalopy-1.0b11.xml	TreeMap	3.605256
profileC5	9	C :/Systemes/FindBugsEtLib.xml	TreeMap	33.71916
profileC5	16	C :/Systemes/jdk1.1.xml	Sunburst	17.87606

Annexe III

Résultats complets pour les questions automatiques de l'expérience sur le placement

Acronyme	Nom	Catégorie
CBO	Coupling Between Objetcs	couplage
DIT	Depth in Inheritance Tree	structure
LCOM 5	Lack of Cohesion in Methods	cohésion
WMC	Weighted Method per class	taille
ACAIC	Ancestor class-attribute import coupling	couplage
ACMIC	Ancestor class-method import coupling	couplage
AID	Average Inheritance Depth of a class	couplage
CLD	Class-to-leaf Depth	structure
Connectivity	Connectivity	cohesion
DCAEC	Descendants coupling attributes export coupling	coupling
DCMEC	Descendants coupling methods export coupling	coupling
NCM	Number of Class Methods	coupling
NMA	Number of new Methods	taille
NMI	Number of Methods Inherited	taille
NMO	Number of Methods overriden	taille
NOA	Number of Ancestors	structure
NOC	Number of Children	structure
NOD	Number od Descendants	structure
NOP	Number of Parents	structure
SIX	Specialisation Index NMO*DIT/(NM)	structure

Tableau III.1 – Métriques couramment utilisées dans notre approche de visualisation.