Analyse statique et dynamique de code et visualisation des logiciels via la métaphore de la ville: contribution à l'aide à la compréhension des programmes.

Pierre Caserta

UNIVERSITÉ DE LORRAINE

Département de formation doctorale en informatique

Institut National
Polytechnique de Lorraine

École doctorale IAEM Lorraine

Analyse statique et dynamique de code et visualisation des logiciels via la métaphore de la ville : contribution à l'aide à la compréhension des programmes

THÈSE

présentée et soutenue publiquement le 07 décembre 2012

pour l'obtention du

Doctorat de l'université de Lorraine

(spécialité informatique)

par

Caserta Pierre

Composition du jury

Président :		
Rapporteurs :	Roland Ducournau	Professeur de LIRMM de Montpellier
	Houari A. Sahraoui	Professeur de l'Université de Montréal
Examinateurs :	Stéphane Ducasse	Directeur de recherche à l'INRIA Lille
	Antoire Beugnard	Professeur à l'ENST de Bretagne
	Olivier Festor	Directeur de recherche à l'INRIA Nancy
	Jean-Pierre Thomesse	Professeur à l'INPL Nancy
	Olivier Zendra	Chargé de recherche à l'INRIA Nancy

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Loria
Laboratoire lorrain de recherche en informatique et ses applications

Mis en page avec la classe thloria.

Remerciements

Tout au long de mon travail, de nombreuses personnes m'ont aidé et soutenu. Ce sont ces personnes que je voudrais tout particulièrement remercier.

En tout premier lieu, je tiens à remercier Olivier Zendra, co-encadrant de cette thèse, pour sa disponibilité, son dynamisme et son efficacité. Je le remercie également pour toutes les collaborations qu'il m'a permis d'entreprendre et pour m'avoir financé mes missions qui furent très enrichissantes.

Je remercie Jean-Pierre Thomesse qui a dirigé cette thèse pour la confiance qu'il m'a témoignée.

Je remercie très chaleureusement Houari A Sahraoui pour son accueil au sein du Laboratoire DIRO de l'Université de Montréal durant l'été 2011. Ces deux mois passés au laboratoire ont été très agréables et instructifs, j'en garde un excellent souvenir.

Je remercie très chaleureusement Roland Ducournau pour son accueil au sein du Laboratoire LIRMM de l'Université de Montpellier durant une semaine en février 2011.

Un grand merci aux assistantes Marie-Francoise Loubressac et Laurence Benini, pour leur efficacité et leur patience.

Merci à Olivier Festor, mon référent interne, pour avoir pris le temps de lire mes rapports d'avancements annuels.

Merci à Ye-Qiong Song, Jean-Philippe Mangeot, Nicolae Brinzei, Nadia Bellalem, Yolande Belaid, Horatiu Cirstea, Vincent Thomas, Emmanuel Nataf et tous les enseignants avec lesquels j'ai eu le plaisir de travailler.

Je tiens à remercier Slim Ouni, mon tuteur de monitorat, pour m'avoir conseillé et appris à devenir un meilleur enseignant.

Un grand merci à mes collègues docteurs et amis qui ont contribué à l'ambiance chaleureuse du LORIA. A Christian Gillot, Sylvain Raybau, Dorin Maxim, Cristian Maxim, Adrien Guenard, Aurélien Monot, Hugo Cruz Sanchez, Code Lo, Bilel Nefzi, Stéphanie Girod, merci pour votre bonne humeur et surtout pour tous ces instants partagés à Nancy.

Je pense également à toutes les personnes qui, de près ou de loin, ont contribué à cette thèse.

A Fabrice Vergnaud, pour sa relecture profonde de ce présent manuscrit et l'extermination de nombreuses fautes, coquilles et mauvaises tournures de phrases.

Merci à Houari A Sahraoui et Roland Ducournau qui m'ont fait l'honneur d'être rapporteurs de ma thèse, ainsi que Antoine Beugnard, Stéphane Ducasse et Olivier Festor pour avoir accepté de faire partie du jury de thèse et d'évaluer ce travail, côtés bien entendu de Jean-Pierre Thomèsse et Olivier Zendra.

Un grand merci à ma famille, plus particulièrement à mes parents qui m'ont permis de réaliser mes études dans les meilleures conditions. Pour toutes ces raisons, je vous dédie cette thèse.

Je dédie cette thèse à mes parents
en remerciement de leur soutien
tout au long de mes études.

Table des matières

Table des figures

1

Introduction

1.1 Avant-propos et historique

Ce mémoire présente un travail de thèse débuté fin 2008, mais dont les motivations remontent à bien plus loin.

En effet, j'ai commencé à me passionner pour les langages à objets dès 2003 lors de mes premières années d'études supérieures. Au travers de projets de programmation personnels et scolaires, j'ai découvert et perfectionné mes connaissances dans ce domaine. C'est vers janvier 2006, durant ma maîtrise, que j'ai découvert l'importance des modèles de conception à objets. Étant fasciné par ces modèles, je me suis replongé dans mes développements passés pour re-concevoir leurs structures en appliquant des patrons de conception reconnus pour leur qualité. Pourtant j'ai trouvé ce travail de re-conception très fastidieux et très complexe. Certains projets de re-conception ont d'ailleurs été des échecs, car mes anciens programmes n'avaient pas les fondations solides qui permettent une modularité du code. De plus, j'ai trouvé que le code de certaines fonctionnalités de mes programmes était très complexe à comprendre, car il y avait un énorme couplage avec d'autres composants. C'est à ce moment-là que j'ai compris qu'avoir une conception de qualité est très important pour permettre une évolution constante du logiciel. Mais j'ai également compris que ceci n'est pas toujours facile et que la notion de conception de qualité n'est pas une science exacte. Il semblait donc utile de se concentrer sur des outils permettant d'aider à la conception des logiciels. C'est à cette époque que j'ai quitté *Notepad* pour me rabattre vers *Eclipse IDE*, avec ses outils qui facilitent (un peu) le développement et contribuent (un peu) à la qualité du code.

Ce n'est que peu de temps avant le début de ma thèse en 2008 que je me suis décidé à explorer les outils de visualisation du logiciel permettant d'en faciliter la compréhension. J'ai tout de suite compris que cette technique promettait un développement beaucoup plus efficace des logiciels et ai donc décidé d'en faire mon sujet de thèse, sous la direction d'Olivier Zendra et Jean-Pierre Thomesse.

Nous avons donc exploré de manière approfondie le domaine de la visualisation du code source (statique) et du comportement (dynamique) des logiciels. Celui-ci étant extrêmement vaste, nous nous sommes attachés à faire une bibliographie complète qui serve également de point de référence dans la littérature du domaine. Ce travail ambitieux mais nécessaire a donné lieu à une publication bibliographique dans le journal *IEEE Transactions on Visualization and Computer Graphics (TVCG)* de très haut niveau du domaine [CZ11b]. Il nous a également fourni une connaissance approfondie du domaine et une vision générale qui nous a fait orienter certains de nos travaux sur l'aspect dynamique du logiciel. En effet, de nombreux travaux portaient déjà

sur la visualisation du code source et très peu sur la visualisation de son exécution. Certains experts dans le domaine considéraient que ce phénomène était dû, en partie, au manque d'outil d'analyse de l'exécution du logiciel, situation elle-même causée par le fait que l'implémentation d'outils d'analyse dynamique est considérée plus complexe que pour les outils statiques.

Nos travaux ont donc consisté, dans un premier temps, à développer des outils qui permettent une analyse de l'exécution des programmes, surpassant ce qui se faisait déjà dans le domaine. Pour cela nous avons dû notamment faire face aux problèmes de performance de notre analyseur, qui s'exécute en même temps que le programme analysé. Nous avons donc conçu notre outil, nommé VITRAIL JBInsTrace, en utilisant des techniques de conceptions novatrices et ingénieuses pour permettre une analyse de l'exécution avec un niveau de précision très fin. L'effort consacré à ce projet a été considérable et a permis d'atteindre un bon niveau de résultat, tant en précision d'analyse qu'en performance, qui a été bien accueilli par la communauté [CZ11a, CZ12]. En effet, notre article [CZ12] a été publié dans l'édition spéciale *Experimental Software and Toolkits (EST)* de la revue *Elsevier's Science of Computer Programming*.

Dans un deuxième temps, la complexité des traces résultantes de nos analyses dynamiques nous a poussé à développer une nouvelle technique de visualisation offrant une présentation appropriée des informations dynamiques rassemblées dans nos traces. Nous nous sommes donc inspirés de ce qui se faisait de mieux, à notre sens, dans le domaine de la visualisation de l'aspect statique du logiciel, en terme de représentation d'appels de méthodes et de représentation de la structure du logiciel. Ce travail de conception nous a conduit à développer VITRAIL Visualizer, qui combine la technique des *Hierarchical Edge Bundles* en fonction de la structure du logiciel, et la représentation sous forme de ville [CZB11]. En ajoutant une animation à la visualisation, nous avons ainsi pu montrer de façon lisible l'ensemble des (très nombreux) appels de méthodes qui ont eu lieu lors de l'exécution du programme. Encore une fois, gérer une telles quantité de données n'a pas été une chose simple et le développement de VITRAIL Visualizer a demandé beaucoup d'attention et d'optimisation. Ces travaux sur la visualisation ont été publiés à *VISSOFT*, workshop de référence dans le domaine de la visualisation des logiciels [CZB11].

Ce mémoire de thèse de doctorat présente la synthèse de mes travaux de recherches incluant ces projets.

1.2 Problématique et contexte

La maintenance d'un logiciel et le temps de développement de nouvelles fonctionnalités sont des éléments cruciaux en termes de gestion de projet, car le non-respect des délais, parfois très courts, peut conduire à un surcoût imprévu, voire à l'échec du projet. Ceci implique fort logiquement un souci de bien maitriser le logiciel et son fonctionnement interne. Néanmoins, le logiciel étant complexe, de par la taille de son code source et les nombreuses relations entre ses composants, appréhender et maitriser sa conception est souvent difficile. En situation réelle, les développeurs et les personnes chargées de la maintenance doivent sans cesse (re)découvrir l'implantation de certaines fonctionnalités du logiciel, au prix d'efforts important et d'une perte de temps.

C'est à ce niveau, *faciliter la compréhension du logiciel*, que se situe la problématique générale de cette thèse, avec pour objectif de répondre - au moins partiellement, car il semble qu'une réponse définitive soit utopique - à cette question : *Quelle est la manière la plus efficace de présenter le logiciel pour faciliter efficacement sa compréhension ?*

Une question à la fois aussi simple et aussi vaste pouvait être abordée de façons très différentes et avoir des réponses multiples.

Les recherches menées durant cette thèse sont triples. D'une part, nous avons souhaité établir un état de l'art qui permette à la fois de référencer, mais aussi de comparer les différentes analyses, métriques et visualisations de logiciel. D'autre part, nous avons implanté un analyseur de l'exécution permettant une analyse approfondie de l'exécution encore peu voire jamais atteinte par d'autres outils. Enfin, nous avons créé une technique de visualisation facilitant la compréhension de l'exécution du logiciel et donc du logiciel lui-même.

La première partie de notre travail de thèse du point de vue chronologique consistait donc à référencer et étudier les différentes techniques de visualisation. Le logiciel étant virtuel et intangible, lui inventer une représentation visuelle en permettait une meilleure compréhension, car l'être humain est souvent plus efficace pour comprendre une information lorsque celle-ci peut être représentée concrètement. La visualisation du logiciel est un vaste domaine qui regroupe trois aspects fondamentaux : l'aspect statique, l'aspect dynamique et l'évolution [Die07]. La visualisation de *l'aspect statique du logiciel* se focalise sur la visualisation de code source du logiciel. Inversement, la visualisation *de l'aspect dynamique du logiciel* apporte des informations sur une ou des exécution(s) particulière(s) du logiciel et d'examiner comment les fonctionnalités du logiciel se réalisent. Enfin, la visualisation de *l'évolution du logiciel* ou encore de *l'évolution de l'aspect statique du logiciel* ajoute la notion de temps aux visualisations de l'aspect statique. De nombreux travaux de recherche concernaient la visualisation de l'aspect statique du logiciel et de son évolution. C'est sur ces deux catégories que s'est focalisé principalement notre bibliographie qui fait référence à 190 articles et livres. Cette mine d'information nous a permis d'avoir une vue d'ensemble et un regard critique sur les différentes approches des acteurs dans le domaine de la visualisation du logiciel.

Le second axe pour nos recherche a été l'implémentation d'un analyseur permettant une analyse fine de l'exécution des logiciels. En effet, il n'existait pas d'analyseur qui nous permettrait d'aller jusqu'au bout de nos recherches, en traitant par exemple du polymorphisme des sites d'appels à l'exécution. Nous avons donc choisi de développer notre propre outil et de concevoir une technique d'analyse suffisamment flexible pour nous permettre de recueillir les informations dynamiques que nous souhaitions pour nos recherches. Nous avons donc proposé un outil qui permet de tracer l'exécution du logiciel à grain fin et en récupérant les informations sur les références des sites d'appels.

Enfin, nous avons exploré une troisième direction de recherche complémentaire aux précédentes, car nous avons créé une technique de visualisation adaptée pour visualiser les résultats de l'analyse de l'exécution. En effet, la visualisation de l'exécution du logiciel nécessite une attention particulière qui n'a pas forcement les mêmes exigences que pour la représentation du code source. Ainsi, dans les langages à objets, les instances communiquent grâce à l'envoi de messages. Or ces communications entre objets sont essentielles pour comprendre le fonctionnement du logiciel. La visualisation doit donc être particulièrement adaptée pour représenter les relations entre les composants. Nous avons donc proposé un outil qui permet la visualisation des relations an sein d'une représentation graphique basée sur la métaphore de la ville. Pour cela nous avons conçu une technique nommé : *3D Hierarchical Edge Bundles*, permettant l'élévation et le regroupement des arrêtes (ou liens) pour une visualisation efficace des relations.

Première partie

Analyses, métriques et visualisations du logiciel : un état de l'art

2

Analyses et métriques statiques du logiciel

Sommaire

2.1 Introduction

Une métrique logicielle est une mesure issue des propriétés techniques ou fonctionnelles d'un logiciel. Celle-ci peuvent être divisées en trois sous-catégories [AC94a, FN00, WB01] :

métrique produit est calculée à partir du code source du logiciel. Elle quantifie certains aspects du logiciel [XSZC00, PV03] tels que la taille, complexité, couplage, etc.

métrique processus est liée au processus de développement. Elle donne des informations sur certains aspects du processus de développement [AG00] tels que le temps, l'effort investi, etc.

métrique projet est liée au projet lui-même tels que les métriques d'ordonnancement, de coût, du nombre de ressources humaines, etc.

Les métriques liées au produit lui-même sont particulièrement intéressantes du point de vue des développeurs parce qu'elles sont calculées à partir d'une analyse du code source ou de l'exécution du logiciel. Ces métriques apportent donc des informations plus précises sur l'état du code source et de son exécution. Dans ce chapitre nous traitons uniquement des métriques produits

et plus précisément des métriques statiques. Les métriques dynamiques, moins nombreuses, sont traitées dans le chapitre 4.

Les métriques statiques caractérisent et quantifient un aspect précis du code source du logiciel. Pour pallier le risque de subjectivité d'une métrique, celle-ci et toutes les mesures permettant de la calculer doivent être formellement définies [Sch92] ; une validation empirique doit de plus être faite pour démontrer l'utilité de la métrique dans la pratique.

Aujourd'hui les développeurs, managers, et responsables de la maintenance ont besoin de mieux comprendre, contrôler, gérer, prévoir et améliorer le processus de développement et de maintenance du logiciel [BDR96]. Or les métriques produits peuvent être utilisées pour évaluer la qualité du logiciel [CG90, TZ93, Kan02, CKK01], car elles aident à développer de meilleurs logiciels en mettant en évidence les parties qui méritent plus d'attention de la part des développeurs [Sch02, Kla03].

Elles aident également à mieux gérer la complexité du cycle de vie du logiciel. Les différentes phases qui composent le génie logiciel sont, de façon très simplifiée : la conception préliminaire, la conception détaillée, le codage, les tests, la maintenance. Plus les métriques sont prises en compte tôt lors du cycle de vie du logiciel, plus sa qualité peut être évaluée et contrôlée rapidement [BMB94].

De nombreux travaux de recherche se concentrent sur l'évaluation de différents aspects de la qualité grâce aux métriques. La qualité d'un logiciel à objet est habituellement dépendante de ces six facteurs : fonctionnalité, fiabilité, utilisabilité [AKSS03], efficacité (du système), possibilité de maintenance [FT96], et portabilité [JKC04, LCM+04].

Ce chapitre est organisé de la manière suivante. La section 2.2, présente les différents types de métriques statiques des logiciels. Cette section est divisée en plusieurs sous-sections décrivant chacune un type de métrique particulier. Nous discutons ainsi des métriques statiques de taille (section 2.2.1, de complexité (section 2.2.2), de couplage (section 2.2.3), de cohésion (section 2.2.4) et d'héritage (section 2.2.5). La section 2.3, traite de l'utilisation de plusieurs métriques pour définir un aspect particulier du logiciel. Ainsi la sous-section 2.3.1 présente des ensembles de métriques qui permettent d'estimer la qualité du logiciel et permettent également de prédire les classes qui sont plus susceptibles de contenir des bogues. Puis, dans la sous-section 2.3.2 nous discutons de la normalisation de métriques pour comparer différents systèmes. Enfin, la sous-section 2.3.3 présente une technique composant des opérateurs logiques et métriques pour déterminer les anomalies de conception. Nous donnons une conclusion sur ce chapitre dans la section 2.4.

2.2 Types de métriques du logiciel

Cette section apporte les bases pour comprendre les métriques logicielles du code source (métriques statiques). Nous présentons les métriques :

- de taille et de complexité des composants,
- de couplage et de cohésion entre les composants,
- d'héritage.

2.2.1 Métriques de taille

Les métriques de taille sont les métriques statiques les plus simples à définir, comprendre et calculer. Elles donnent une idée de la taille du logiciel, en calculant par exemple le nombre de

lignes de code source dans les classes, le nombre de classes dans les paquets, etc. La plupart des métriques sont calculées à plusieurs niveaux de granularité, tels que la ligne de code, la méthode, la classe, le paquet ou même le système.

Par exemple, la métrique *Lines Of Code (LOC)* (en français Lignes De Code) est bien souvent utilisée au niveau de la classe et est définie comme étant le nombre de lignes de code du composant. La métrique *Number Of Classes (NOC)* (en français Nombre De Classes) est souvent définie au niveau du paquet et calcule le nombre de classes dans le paquet.

Comme les métriques de taille sont directement basées sur le code source du logiciel, elles ne sont pas disponibles lors des phases de conception du cycle de vie du logiciel. Généralement les métriques de taille sont utilisées pour comparer la taille de différents logiciels, pour prévoir les efforts de développement et de maintenance.

2.2.2 Métriques de complexité

Ces métriques sont utilisées pour mesurer la complexité d'une partie du logiciel. Elles sont souvent corrélées avec les métriques de taille parce qu'un très grand système a de grandes chances d'être également complexe. Plusieurs définitions de la complexité peuvent être trouvées dans [BH83, EBD99, HS95] mais la métrique de complexité la plus utilisée dans la littérature est la métrique *Cyclomatic Complexity (CYCLO)* (en français Nombre Cyclomatique de McCabe) [McC76]. Cette mesure comptabilise le nombre de chemins linéairement indépendants qu'il est possible de suivre au sein d'une méthode.

Les métriques de complexité sont souvent utilisées comme critère de qualité du logiciel. En effet, un code source très complexe n'est pas un gage de bonne qualité du logiciel car cette complexité implique que le code est difficile à comprendre et probablement peu modulaire.

2.2.3 Métriques de couplage

Les métriques de couplage décrivent les dépendances entre les composants du système [EKS92, AK99]. Plus le couplage entre deux composants est fort, plus le système est compliqué à comprendre, modifier ou corriger [SB91]. Le couplage est souvent calculé à différents niveaux de granularité tels que les méthodes, les classes et paquets. De plus, beaucoup de métriques de couplage ont été définies dans la littérature. Ceci rend très difficile le choix des métriques de couplage appropriées pour une tâche spécifique[BDJ99]. La métrique de couplage la plus connue est probablement la métrique *Coupling Between Objects (CBO)* (en français Couplage Entre Les Objets) mesure pour chaque classe, le nombre d'autres classes qui sont couplées avec cette classe.

On peut distinguer trois types de métriques de couplage dans les langages à objets :

interaction : lorsque les classes communiquent par l'envoi de messages ou le partage de données.

composant : lorsqu'une classe utilise une instance d'une autre classe comme attribut, paramètre d'une de ses méthodes ou variable locale d'une de ses méthodes.

héritage : lorsqu'une classe est directement ou indirectement une sous-classe d'une autre classe.

Le calcul des métriques de couplage du code source est complexe, surtout lorsque l'on considère le couplage indirect (via la fermeture transitive) [YT07]. Ce dernier permet de mettre en évidence des dépendances "éloignées" entre les classes [YR01].

Une bonne pratique dans les langages à objets est de minimiser le couplage, pour limiter les efforts de maintenance et pour améliorer la réutilisabilité du code. Il est convenu que certaines formes de couplage font baisser la qualité du logiciel car elles demandent plus d'efforts de

maintenance à cause de leurs complexité [HM95]. Par exemple, la forme de couplage la moins préconisée arrive lorsqu'une méthode accède directement à un composant non publique d'une autre classe (ceci est possible en C++ lorsque deux classes sont déclarées "friend"). Il en est de même lorsqu'une sous-classe modifie directement un attribut d'une super-classe, car on s'attend à ce que se soit la classe qui définit l'attribut qui modifie sa valeur. La meilleure forme de couplage est celle où deux classes communiquent par l'envoi de messages sans aucun paramètre inutile.

2.2.4 Métriques de cohésion

Les métriques de cohésion décrivent le taux de connectivité au sein d'un composant du logiciel. Celle-ci mesure donc le taux d'utilisation des attributs d'une classe par les méthodes de cette même classe et mesure donc l'application des principes d'encapsulation des données.

C'est un facteur important pour la portabilité, la fiabilité et donc pour la qualité globale du système. Une forte cohésion engendre un couplage faible ; les deux métriques sont donc corrélées. En pratique, on calcule le manque de cohésion plutôt que le taux de cohésion lui-même [CK91]. La métrique *Lack of Cohesion on Methods (LCOM)* (en français Manque De Cohésion Entre Les Méthodes), compte simplement le nombre de méthodes qui n'utilisent pas les attributs de la classe. L'idée est ici qu'une classe avec des méthodes n'utilisant pas les attributs peut probablement être divisée en plusieurs sous-classes, toutes plus cohésive que la classe initiale.

Même si la cohésion est souvent calculée au niveau d'une classe, celle-ci peut être également définie au niveau d'une méthode, une classe, un paquet ou toute une partie de l'arbre d'héritage [EKS92, HM95, BDW98, AK99].

2.2.5 Métriques d'héritage

Les métriques d'héritage sont calculées à partir de l'arbre d'héritage. Par exemple, la métrique *Number Of Children of a Class (NOC)* (en français Nombre D'Enfants D'une Classe), mesure le nombre d'héritiers immédiats de la classe. La métrique NOC permet donc d'évaluer si une classe parent (classe de base) est réutilisée immédiatement par beaucoup d'autres classes.

Une autre métrique étroitement liée à NOC est *Depth of Inheritance Tree (DIT)* (en français Profondeur De l'Arbre d'Héritage). Elle représente la profondeur de la classe dans l'arbre d'héritage. NOC quantifie la taille de l'arbre d'héritage en largeur alors que DIT quantifie la taille en profondeur. En règle générale, une valeur de DIT élevée est à privilégier par rapport à une valeur de NOC élevée parce qu'un arbre d'héritage profond permet la réutilisation et la redéfinition des méthodes des classes parents dans l'arbre d'héritage. Au contraire, une trop grande utilisation de l'héritage peut engendrer une valeur maximale de DIT élevée qui n'est pas un gage de qualité dans ce cas précis.

Les métriques d'héritage sont utiles pour mesurer le taux de complexité de l'architecture du logiciel [AC94b, Bre07]. Une propriété intéressante est qu'elles peuvent être calculées dès la fin du cycle de conception (avant le codage).

2.3 Composition de métriques logicielles

Utiliser une métrique seule permet d'avoir des informations sur un aspect particulier du logiciel, mais les métriques logicielles prennent toute leur importance lorsqu'elles sont considérées au sein d'un ensemble [Kla03].

2.3.1 Ensembles de métriques logicielles

Un ensemble de métriques permet d'étudier plusieurs aspects du logiciel. La composition de métriques permet de faire des suppositions qui se révèlent vraies dans la plupart des cas. Par exemple, lorsqu'une classe a beaucoup de lignes de code, une complexité élevée et un couplage élevé, il est très probable que cette classe offre beaucoup de fonctionnalités et occupe une place importante dans le système et cela n'exclus pas le fait que cette classe est potentiellement mal codée.

Choisir un ensemble de métriques approprié permet de révéler des informations intéressantes qu'il aurait été difficile de trouver autrement que par l'utilisation de l'ensemble de métrique.

Le plus célèbre ensemble de métriques et le plus utilisé dans la littérature est celui de Chidamber et Kemerer [CK91, CK94] appelé plus communément *CK*. Cet ensemble est composé de six métriques définies au niveau de la classe.

WMC *Weighted Methods per Class* (en français Poids Des Méthodes Par Classe), mesure la complexité de la classe en faisant la somme de toutes les métriques de complexité des méthodes de cette classe.

DIT *Depth of Inheritance Tree* (voir section 2.2.5), représente la profondeur de la classe dans l'arbre d'héritage.

NOC *Number Of Children of a Class* (voir section 2.2.5), mesure le nombre d'enfants immédiats de la classe.

CBO *Coupling Between Objects* (voir section 2.2.3), mesure le nombre d'autres classes qui sont couplées avec cette classe.

RFC *Response For a Class* (en français Réponses D'une Classe), mesure la cardinalité de l'ensemble des méthodes qui peuvent être directement appelée lors de l'exécution de n'importe quelle méthode de cette classe.

LCOM *Lack of Cohesion on Methods* (voir section 2.2.4), compte le nombre de méthodes qui n'utilisent directement pas les attributs de la classe.

Cet ensemble de métriques mesure différents aspects importants des langages à objets tels que l'abstraction (avec LCOM, NOC et RFC), l'encapsulation (avec LCOM), la modularité (avec WMC, RFC et CBO) et l'héritage (avec DIT et NOC). L'ensemble CK a été validé de manière empirique comme un bon indicateur de qualité du logiciel et comme un bon prédicateur d'erreurs pour les classes [BBM96, TKC99, SK03, OEGQ07].

Abreu et Carapuça [AC94b, AGE95, AM96, eAC98] ont développé un autre ensemble de métriques appelé : *The Methodology for Object Oriented Development (MOOD)* (en français La Méthodologie Pour Les Développements Orientés-Objets). Cet ensemble MOOD se focalise sur des aspects liés l'architecture du logiciel, tels que le polymorphisme, l'encapsulation, l'héritage et l'envoi de message. L'avantage de cet ensemble est qu'il peut être obtenu très tôt dans le cycle de vie du logiciel (avant le codage) [BM99]

MOOD est composé de six métriques définies au niveau du système :

MHF *Method Hiding Factor* (en français Coefficient De Méthodes Cachées), mesure pour une méthode, le pourcentage de classe pour qui cette méthode n'est pas visible.

AHF *Attribute Hiding Factor* (en français Coefficient d'Attributs Cachés), mesure pour un attribut, le pourcentage de classe pour qui cet attribut n'est pas visible.

MIF *Method Inheritance Factor* (en français Coefficient De Méthodes Héritées), mesure le pourcentage de méthodes héritées (et non-réécrites) dans le système.

AIF *Attribute Inheritance Factor* (en français Coefficient d'Attributs Hérités), mesure le pourcentage d'attributs hérités dans le système.

PF *Polymorphism Factor* (en français Coefficient De Polymorphisme), est le nombre total de méthodes refédinies, divisé par le nombre maximum de surcharges possibles.

CF *Coupling Factor* (en français Coefficient De Couplage), est le nombre de relations entre les classes divisé par le nombre maximum de couplages possibles.

Les deux ensembles précédents peuvent être complémentaires, car l'ensemble CK est utile lors de la phase de codage, alors que le MOOD opère au niveau du système et permet d'avoir une vue d'ensemble plus directe pour les managers [HCN98].

Plusieurs travaux ont montré que l'utilisation d'ensembles de métriques permet de déterminer la probabilité qu'une classe contienne des bugs [MK92, BWDVP00, BMW02]. Ces travaux sont très intéressants, car ils montrent empiriquement que l'utilisation d'ensembles de métriques permet de cibler des modules sur lesquels la maintenance améliorative doit se concentrer. L'article [BW02] résume les différents résultats de plusieurs études sur les ensembles de métriques logicielles.

Ces deux ensembles sont les plus courants dans le domaine des métriques logicielles mais l'attrait grandissant pour ces dernières a conduit à la création d'un grand nombre d'autres ensembles de métriques [BD97, CKK01, BD02, OEGQ07, LNH07].

2.3.2 Normalisation de métriques

Comparer les métriques d'un logiciel avec celles d'un autre peut se révéler très utile pour permettre d'estimer la qualité du logiciel en développement par rapport à un logiciel plus abouti.

Pourtant la comparaison n'est pas immédiate. En effet, une métrique d'un système est difficilement comparable à une métrique d'un autre système. Pour que cette comparaison soit possible il faut normaliser les métriques en fonction de la taille de chaque logiciel. Une *métrique normalisée* (ou *densité de métrique*) est tout simplement la valeur de la métrique divisée par une métrique de taille du système. Ainsi, une métrique normalisée peut être utilisée pour comparer deux systèmes, car chacune des valeurs sera relatives à la taille de son système respectif.

Pour illustrer et donner quelques exemples de métriques nous utiliseront une représentation graphique et textuelle proposé par Lanza, Marinescu et Ducasse dans leur livre [LM06]. Cette représentation nommée *Overview Pyramid* (Fig.2.1) utilise un ensemble de métriques définies au niveau du système. L'Overview Pyramid permet d'avoir une idée générale du logiciel, mais ne permet pas vraiment de le visualiser. Nous traiterons de l'état de l'art des techniques de visualisation de métriques dans les chapitres 3 et 5.

La représentation en pyramide est divisée en trois parties colorées :

- la partie en jaune représente les métriques logicielles associées à la taille et la complexité du logiciel. Les métriques sont : *Number Of Package (NOP)* (en français Nombre De Paquets), NOC, *Number Of Methods (NOM)* (en français Nombre De Méthodes), LOC et CYCLO.
- la partie en bleu représente les métriques du logiciel associées au couplage entre les éléments du logiciel : *Fan-Out (FANOUT)* (en français Nombre De Classes Appelées) et *Number Of Method Calls (CALLS)* (en français Nombre d'Appels De Méthodes).
- la partie verte représente les métriques du logiciel associées à l'héritage. Les métriques sont : *Average Number of Descendants Classes (ANDC)* (en français le nombre moyen de classes héritées) et *Average Hierarchy Height (AHH)* (en français la taille moyenne de la hiérarchie d'héritage).

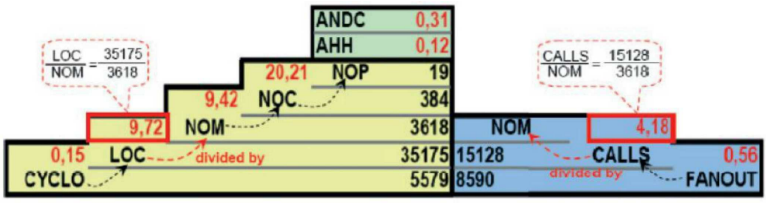


FIGURE 2.1 – Overview Pyramid [LM06]

Les valeurs de métriques sont représentées en noir dans la figure 2.1. Les valeurs en rouge sont les ratios entre deux valeurs superposées. Par exemple, la valeur 9,72 encadrée en rouge correspond au ratio entre LOC et NOM, et elle représente le nombre moyen de lignes de code par méthode dans le logiciel. Ceci permet d'avoir une idée globale du logiciel et permet également la comparaison des métriques de ce logiciel avec celles d'autres logiciels.

2.3.3 Composition logique de métriques

Certaines conceptions orientées objets sont considérées comme néfastes pour la qualité du logiciel. Ces mauvaises architectures peuvent poser des problèmes plus tard, lors de l'ajout de nouvelles fonctionnalités ou lors de la correction de bogues lors de la maintenance du logiciel [Fow99, MT04]. Pour cette raison, il est préférable d'avoir des outils qui permettent d'évaluer la qualité assez tôt dans le processus de développement du logiciel.

Marinescu [Mar04] introduit une technique appelée *Detection Strategy* qui permet de déceler les parties du logiciel qui ne respectent pas les "bonnes" règles de conception des langages à objets définies par Riel [Rie96]. Ces travaux consistent, entre autres, à utiliser des métriques logicielles et des conditions logiques pour détecter les mauvaises architectures. Par exemple une classe *God* est définie comme étant une classe qui a tendance à centraliser toutes les fonctionnalités du système. Cette pratique est considérée comme mauvaise dans les systèmes à objets car ils tentent au contraire de répartir les fonctionnalités entre les classes. La figure 2.2 montre la détection d'une classe God en testant si cette classe utilise directement des attributs d'autres classes *ET* que la complexité de cette classe est très grande *ET* que la cohésion de la classe est basse.

Marinescu transpose ces conditions en formule logique (Fig.2.2) en utilisant les métriques suivantes :

ATFD *Access to Foreign Data* (en français Accès à Des Données Etrangères), est le nombre d'accès que fait une classe aux attributs d'une autre classe (directement ou via l'utilisation de méthodes).

WMC *Weighted Method Count* (en français Accès à Des Données Etrangères), est la somme des complexités cyclomatiques de toutes les méthodes d'une classe.

TCC *Tight Class Cohesion* (en français Ajustement De La Cohésion Des Classes), est le nombre de méthodes publiques directement connectées (qui s'appellent directement) au sein d'une classe, divisé par le nombre maximum de connexions possibles entre les méthodes de cette classe.

Le problème sous-jacent avec cette technique est de déterminer convenablement les seuils : *FEW, VERY HIGH* et *ONE THIRD* qui jouent un rôle crucial dans la composition logique [LM06].

Ces seuils peuvent être choisis empiriquement, en se basant sur des métriques de densité de plusieurs systèmes, de manière automatique [MM05] ou en analysant plusieurs versions d'un

même logiciel [Rat03, RDGM04].

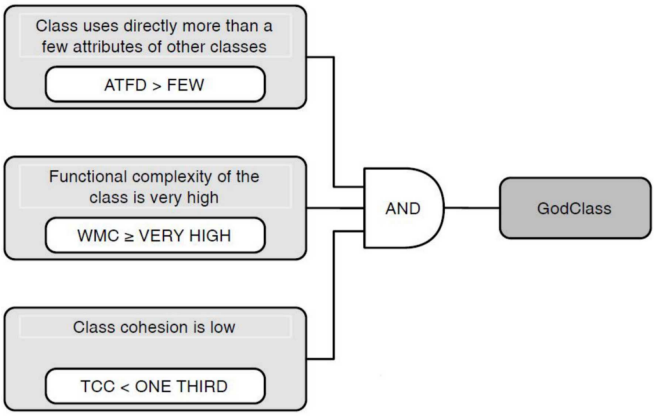


FIGURE 2.2 – Détection d'une classe *God* [LM06]

2.4 Conclusion

Dans cette section, nous avons décrit plusieurs métriques statiques et quelques ensembles de métriques logicielles qui permettent de mesurer la qualité des logiciels. Ces métriques aident les développeurs, managers, et responsables de la maintenance à mieux comprendre, contrôler, gérer, prévoir et améliorer le processus de développement et de maintenance du logiciel. Nous verrons dans les chapitres suivants comment les métriques logicielles statiques sont utilisées pour visualiser le logiciel.

3

Visualisation de l'aspect statique du logiciel

Sommaire

3.1 Introduction

Plus la taille du code source du logiciel augmente durant son développement, plus celui-ci devient complexe. Le code source contient de nombreuses relations entre les composants, à plusieurs niveaux de granularité. La nécessité de prendre en compte d'autres types d'informations, comme l'utilisation des structures de données, l'héritage, le flot inter-procédural et les fils d'exécution rendent le fonctionnement du logiciel très complexe et difficile à comprendre. Ceci est d'autant plus vrai lorsqu'on souhaite comprendre le fonctionnement d'un logiciel sans avoir été impliqué dans son développement. Or, le processus de maintenance est connu pour être le processus du cycle de vie du logiciel le plus couteux en temps et en argent [AGE95]. De plus une grosse partie du temps de maintenance est consacrée uniquement à comprendre le fonctionnement (et donc le code) du logiciel. Les outils dont le but principal est de faciliter la compréhension du logiciel peuvent donc fort logiquement réduire ces temps et coûts de maintenance [VMV96, SWM00].

Cependant, le logiciel est virtuel et intangible [KM00]. Or visualiser un logiciel c'est en dessiner une image [RC92] et donc obtenir une vue tangible du logiciel [Sta98, Die07]. De plus, l'être humain est plus efficace pour comprendre l'information lorsque celle-ci est représenté de manière concrète (graphiquement) plutôt que virtuelle ou théorique [Mar82, Bie87, Spe90]. Ainsi

représenter graphiquement le logiciel est un bon moyen pour permettre une compréhension plus rapide et plus précise de la structuration du logiciel et de ses fonctionnalités. Des expérimentations empiriques ont montré qu'utiliser des techniques de visualisation lors du développement logiciel augmente les chances de réussite [Gra92, Zha03].

Néanmoins, la manière dont l'être humain perçoit et interagit avec la visualisation influence grandement la compréhensibilité de l'information [SFM99, TM04, CMS99]. Les mécanismes de perception de l'être humain doivent donc être pris en compte pour améliorer certains aspects cognitifs [SFM99, SR96, PBG98, Tud03].

Dans ce chapitre, nous établissons l'état de l'art des visualisations de l'aspect statique des logiciels, qu'elles soient 2D ou 3D, en présentant leurs points positifs et négatifs [Sta93, Rei95]. Les visualisations 2D ont été bien plus étudiées et ont fait l'objet de beaucoup de publications. Certaines ont même fait l'objet de produits commerciaux à l'échelle industrielle [TV08, TBV09]. Cependant, ces dernières années, nous notons un attrait grandissant pour la recherche sur les visualisations de logiciel en 3D [You96, TC09] en dépit du fait que la navigation dans un environnement 3D est plus problématique que pour la visualisation 2D [PFW98].

Durant ces dernières années, les chercheurs ont proposé beaucoup de techniques de visualisation et quelques de moyens de classification ont été publiés [PBS93, RCR$^+$93, Kni01, MMC02]. Des travaux [BK01, GME05, SCG05, PdQ06] traitent de l'état de l'art de la visualisation des logiciels, notamment notre article [CZ11b] qui est l'état de l'art le plus récent et le plus complet à ce jour dans le domaine de la visualisation de l'aspect statique des logiciels et de son évolution. En effet, certains travaux bibliographiques [Car08, TC09] couvrent uniquement un sous-ensemble particulier du domaine. L'article [Car08] traite uniquement la visualisation de l'architecture du logiciel et l'article [TC09] uniquement des visualisations du logiciel en 3D. Dans [CZ11b] nous parcourons la plupart les travaux existants et expliquons chacune des techniques de visualisation en apportant notre vision critique. Nous apportons également une catégorisation des différentes visualisations.

En effets, il est très difficile de classifier les techniques de visualisation, car celles-ci sont très diverses, avec bien entendu certaines caractéristiques communes. Ces points communs nous ont permis de proposer notre classification. Dans ce chapitre nous allons présenter les caractéristiques et les fonctionnalités de chacune des techniques de visualisation afin d'apporter un état de l'art qui puisse intéresser autant l'industrie que la recherche. Un de nos objectifs est de promouvoir de nouvelles directions de recherche en faisant le bilan de ce qui a été fait jusqu'à présent.

La visualisation de l'aspect statique du logiciel peut être divisée en deux catégories principales :

- les visualisations qui donnent une image du logiciel à l'instant t.
- les visualisations qui montrent l'évolution du logiciel à travers plusieurs versions.

Même si la notion de temps est ajoutée à la deuxième catégorie, toutes deux appartiennent au domaine de la visualisation de l'aspect statique parce que nous nous intéressons à visualiser le code source. Pour ces deux catégories, nous considérons à chaque fois trois niveaux de granularité basés sur les niveaux d'abstraction suivants : la ligne de code, le niveau intermédiaire (paquets, méthodes, classes), le niveau de l'architecture globale [Kos03, PLL04].

Ce chapitre est organisé de la manière suivante : Dans la première partie (section 3.2 à 3.4), nous considérons les techniques de visualisation de l'aspect statique du logiciel à l'instant t. La section 3.2 présente les techniques de visualisations qui facilitent la compréhension au niveau de la ligne de code. Dans la section 3.3 nous discutons des techniques de visualisation qui fournissent des renseignements au niveau des classes et de leurs fonctionnements interne.

La section 3.4 traite des techniques de visualisation qui se concentrent sur trois aspects de la visualisation de l'architecture. Le premier (section 3.4.1) permet la visualisation de toute l'organisation du logiciel (paquets, classes, méthodes). Le second (section 3.4.2) se concentre sur la visualisation des relations entre les composants, que ces relations soient des relations d'héritage ou des relations d'appels de méthode. Le troisième aspect (section 3.4.3) traite des techniques de visualisation qui s'intéressent à visualiser les métriques calculées sur le logiciel.

Dans la seconde partie du chapitre (section 3.5), nous présentons des techniques de visualisation permettant de visualiser l'évolution du logiciel. La section 3.5.1 présente donc les techniques de visualisation qui aident à visualiser l'évolution du code source au niveau de la ligne de code. Dans la section 3.5.2, nous discutons des techniques de visualisation qui fournissent des renseignements sur l'évolution des classes à travers les différentes versions du logiciel. La section 3.5.3 traite des techniques de visualisation qui se concentrent sur la visualisation de l'évolution de l'architecture. Deux différentes catégories de visualisations y sont présentées. La première catégorie (section 3.5.3.1) visualise les changements de l'organisation du code source du logiciel. La seconde catégorie (Section 3.5.3.2) visualise l'évolution des métriques calculées sur le logiciel à travers les différentes versions. Enfin, nous ferons une conclusion sur ce chapitre d'état de l'art dans la section 3.6.

Afin d'aider le lecteur à parcourir ce chapitre, le tableau 3.1 en apporte une vue synthétique, avec des liens vers les sections correspondantes. Ce tableau permet également d'avoir une vue globale de notre méthode de classification. L'année d'apparition de la technique de visualisation y est également affichée pour montrer l'évolution du domaine de la visualisation des logiciels.

3.2 Visualisation orientée sur les lignes du code source

Cette section traite des techniques de visualisation au niveau de la ligne de code source (voir section 3.2 du tableau 3.1). Néanmoins, les visualisations présentées ici ne se contentent pas d'afficher le code source mais présentent un niveau d'abstraction plus élevé.

SeeSoft [ESSJ92, BE96] est une technique de visualisation par Eick, Steffen et Summer. Cette technique est basée sur la ligne de code source. En effet, la visualisation est en fait une représentation miniaturisée du code source du logiciel. Une ligne de code est représentée par une ligne colorée avec une longueur proportionnelle à la longueur de la ligne dans le code source. L'indentation du code est préservée donc l'imbrication des blocs du code source reste visible. L'inconvénient de cette technique est que l'espace disponible à l'écran n'est pas optimisé. En effet, préserver l'indentation et la taille des lignes de code occupe beaucoup d'espace.

Pour permettre la visualisation simultanée d'un très grand nombre de lignes de code source, les auteurs créent une visualisation qui transforme une ligne de code en un seul *pixel coloré* ou *carré coloré* (Fig. 3.1(a)). De cette manière, la représentation des lignes du code source est réduite à son minimum. Ceci permet de visualiser beaucoup de fichiers sources simultanément en les plaçant les uns à côté des autres dans la fenêtre de visualisation. Néanmoins, cette deuxième technique ne permet pas de représenter l'indentation du code source et il est donc très difficile de s'y repérer. Les auteurs de SeeSoft font remarquer que même si les pixels (ou carrés) sont petits, leurs couleurs permettent tout de même de faire ressortir des motifs qui peuvent interpeler l'utilisateur.

Marcus, Feng et Maletic [MFM03a, MFM03b] se sont inspirés de la technique de visualisation SeeSoft et ont ajouté une troisième dimension dans leur technique de visualisation nommée *Sv3D*.

	Niveau	Thème	Sect.	Technique de visualisation	Représentation	Références	Année
Instant t Visualisation	Ligne de code	Propriété	3.2	Seesoft	pixels colorés 2D	[ESSJ92, BE96]	1992
				Sv3d	cubes colorés 3D	[MFM03a, MFM03b]	2003
	Classe	Opérations	3.3	Class Blueprint	graphe en couche 2D	[Lan03b, LD01, DL05]	1999
	Architecture	Organisation	3.4.1	Treemap	boites colorés encapsulés 2D/3D	[JS91, Shn92, TJ92]	1991
				Circular Treemap	cercles colorés encapsulés 2D/3D	[JS91, WWDW06]	1991
				Ville/villes	ville 3D	[Die93, KM00, PBG03]	1993
				Sunburst	arbre radial coloré 2D	[AH98, Chu98, SZ00]	1998
				Système solaire	système solaire 3D	[YG03, GYB04]	2003
				Voronoi Treemap	formes irrégulière colorés 2D	[BDL05]	2005
		Relations	3.4.2	Dependency Struct. Matrix	grille 2D	[Ste81, BCW01, SJSJ05]	1981
				UML	diagrammes 2D	[GJK+03]	1996
				Geon	diagrammes Geon 3D	[GK98, GRR99, RG00]	1998
				Système solaire	système solaire 3D	[YG03, GYB04]	2003
				Software Landscape	paysage 3D	[BNDL04, BD04]	2004
				Hierarchical Edge Bundles	graphe 2D	[Hol06]	2006
				Ville/Villes	ville avec liens 3D	[AP07, AD07, PEQ+07]	2007
				3D Clustered Graph	graphe 3D	[BD07]	2007
		Métriques	3.4.3	Polymetric Views	graphe 2D	[Lan03b, DDL99, LD03]	1999
				Système Solaire	système solaire avec liens 3D	[YG03, GYB04]	2003
				UML MetricView	diagrammes UML 2D et 3D	[TLTC05]	2005
				Treemap Metrics	boites colorés texturés encapsulés 2D	[HVvW05]	2005
				Ville	ville 3D	[LSP05, DSP08, WL08b]	2005
				UML Area Of Interest	diagrammes avec zones d'intérêts 2D	[BT06, BT09]	2006
Visualiser l'évolution	Ligne	Changements	3.5.1	Métaphore de câbles	câbles	[CAT07, TA08]	2007
	Class		3.5.2	TimeLine	Métaphore du bâtiment	[WL08a]	2008
	Archi.	Organisation	3.5.3.1	Hierarchical Edge Bundles	graphe avec regroupement de liens 2D	[HvW08]	2008
		Métriques		Evolution Matrix	matrice 2D	[Lan01, LD02]	2001
			3.5.3.2	RelVis	diagrammes Kiviat en 2D	[PGFL05]	2005
				Ville/Villes	ville 3D avec animation	[LSP08, WL08a]	2008

TABLE 3.1 – Vue synthétique de l'organisation de ce chapitre et de la classification que nous proposons pour catégoriser les visualisations de l'aspect statique du logiciel.

Cette visualisation représente chaque ligne du code source par un *cube* de la même manière qu'avec SeeSoft. La 3D a été ajoutée pour pallier le manque de visualisation de l'indentation du code de SeeSoft. En effet, Sv3D représente l'indentation grâce à la taille de chaque cube. Ainsi plus la ligne de code est indentée dans le code source, plus la taille du cube sera grande. Cette technique offre un moyen simple pour représenter le code de manière compacte tout en gardant une idée de la structure du code source. Un fichier est représenté par un groupe de cubes placés sur même un plan 3D (Fig. 3.1(b)). Ainsi le logiciel peut être visualisé sous forme de groupes de cubes placés dans l'espace 3D et représentant les fichiers sources. L'utilisateur peut naviguer librement dans l'espace 3D et peut aussi réorganiser les groupes de cube à volonté. Ceci offre des possibilités d'interaction beaucoup plus riche qu'avec la version 2D. Zoomer, pour se focaliser sur certaines parties de la visualisation, est un moyen efficace pour lire en détails certaines parties du code. L'information peut aussi être filtrée en utilisant la transparence afin de garder une vue d'ensemble du système [GR93]. Une autre méthode pour faciliter la compréhension détaillée de certaines parties du code est la dispersion des cubes avec une caractéristique commune sur plusieurs niveaux d'élévation distincts.

Dans les deux figures 3.1(a) et 3.1(b), la couleur montre le type de la structure de contrôle de la ligne de code. Un autre moyen d'affecter la couleur serait de considérer l'âge de chaque ligne en utilisant un dégradé de couleur allant de la couleur rouge pour les lignes récemment ajoutées ou modifiées à la couleur bleue pour les lignes anciennes. Cette dernière technique permet de se focaliser sur les lignes de code les plus récentes en prenant comme hypothèse qu'une portion de code qui n'a pas était modifiée depuis longtemps est plus stable qu'une ligne récente.

Avec la technique de visualisation Sv3D, un grand nombre de caractéristiques visuelles peuvent être utilisées pour coder des informations (hauteur, profondeur, forme, position sur l'axe x, position sur l'axe y, la couleur au-dessus et en dessous de l'axe z). Malgré tout, utiliser toutes les caractéristiques visuelles disponibles pour coder de l'information a de grandes chances de surcharger la visualisation et de la rendre incompréhensible. C'est pour cela qu'il faut un équilibre entre expressivité et efficacité [Mac86]. Dans la figure 3.1(b), la taille des cubes au-dessus de l'axe z représente le niveau d'indentation de la ligne de code (plus le cube est grand, plus il est profondément imbriqué). Les autres caractéristiques visuelles ne sont pas utilisées pour ne pas surcharger la visualisation.

SeeSoft a été utilisé au niveau industriel par Bell Laboratories durant les années 90. Le logiciel visualisé contenait plusieurs millions de lignes de code et plusieurs milliers de développeurs travaillaient pour son développement. Le feedback des développeurs concernant la technique de visualisation Seesoft était positif [BE96]. Ils ont notamment particulièrement apprécié d'être capable d'avoir une vue globale du système et de pouvoir avoir une vue détaillée si nécessaire.

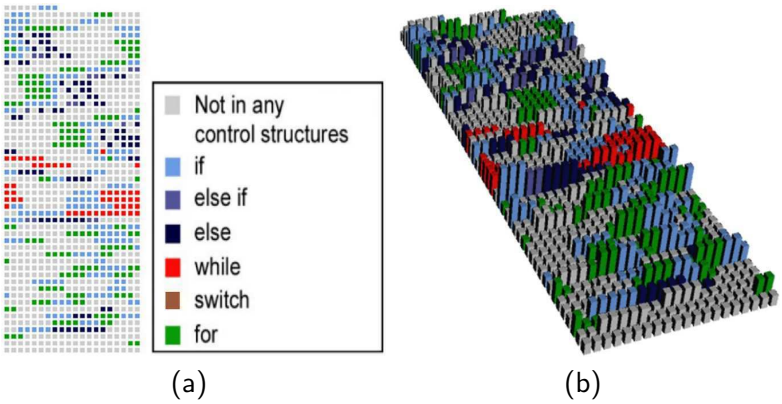


FIGURE 3.1 – (a) SeeSoft et (b) Sv3D [MFM03a].

Si l'on compare les deux visualisations SeeSoft et Sv3D, nous voyons que SeeSoft permet de visualiser l'indentation de manière immédiate. Sv3D est aussi capable de montrer cette information mais celle-ci est représentée par la hauteur d'un cube. L'avantage de Sv3D est qu'il permet de visualiser d'autres informations en utilisant d'autres caractéristiques visuelles.

Néanmoins, nous pensons que la métaphore utilisée par Sv3D n'est pas très intuitive et qu'utiliser plus de deux composantes visuelles complique grandement la compréhension de la visualisation.

Nous pensons qu'un inconvénient majeur de la représentation en pixel est qu'elle change l'orientation des lignes de codes. Traditionnellement, les lignes de code sont représentées les unes en dessous des autres (sous forme textuelle). Or avec la représentation en pixel, les lignes sont représentées les unes à coté des autres ce qui est troublant et nuit à la lisibilité. En plus de ce problème, seul un développeur très impliqué dans le développement du logiciel va être capable de faire correspondre la structure de contrôle représentée par la représentation en pixel avec le code lui-même. Pour toutes ces raisons, nous pensons que la représentation SeeSoft avec une vision miniaturisée du code source est bien plus efficace que la représentation en pixel.

À notre connaissance, la visualisation Sv3D, qui a été créé en 2003, était la dernière visualisation à représenter directement la ligne du code source avec un élément graphique. Il semble que les techniques de visualisation du logiciel plus récentes s'intéressent davantage à représenter le logiciel avec un plus grand niveau d'abstraction.

3.3 Visualisation orientée : classe

Dans cette section, nous présentons une technique de visualisation qui facilite la compréhension du fonctionnement interne d'une classe isolée ou dans le contexte de son héritage (voir détails dans la section 3.3 du tableau 3.1 page 17).

Des recherches ont été réalisées pour visualiser des métriques de cohésion d'une classe [CIK03]. Nous avons vu dans le chapitre 2 que les métriques de cohésions décrivent le degré de connectivité au sein d'un composant. Visualiser la cohésion est un moyen efficace pour détecter les problèmes de conception et d'estimer la qualité du code source. En revanche, la mesure de cohésion ne permet pas de comprendre le fonctionnement d'une classe mais renseigne juste sur le taux d'utilisation des attributs de cette classe.

Le *class blueprint* [Lan03b, LD01, DL05] est une technique de visualisation très sobre de Lanza et Ducasse qui permet de visualiser la structure complète d'une classe, les interactions entre les méthodes au sein de cette classe et la manière dont les méthodes accèdent aux attributs.

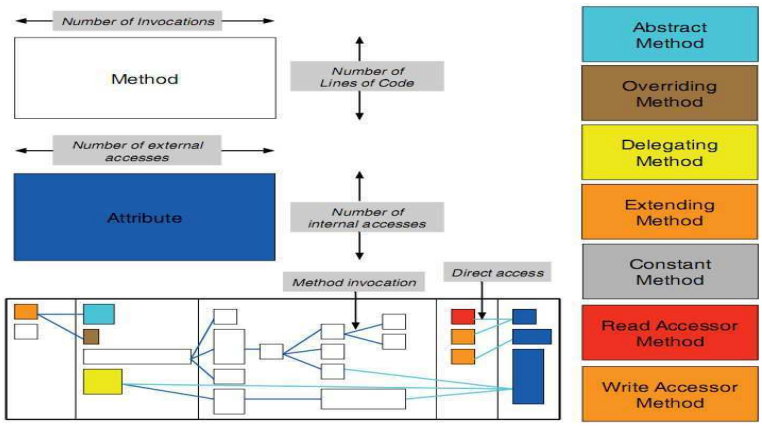


FIGURE 3.2 – Class Blueprint [LM06].

Le *Class Blueprint* (en bas à gauche de la figure 3.2) est divisé en cinq zones (de la gauche vers la droite) : initialisation, interface, implémentation, accesseurs et attributs. Les méthodes et attributs sont représentés par des nœuds placés dans la zone qui correspond à leurs fonctionnalités. Par exemple, une méthode qui crée un objet est placée dans la première zone. Les appels de méthode sont représentés par des arêtes orientées de la gauche vers la droite. La taille des nœuds dépend de plusieurs valeurs de métrique. Dans la figure 3.2, la hauteur d'un noeud représentant une méthode est liée au nombre de lignes de codes dans la méthode ; la largeur est liée au nombre d'invocations de cette méthode. D'autres métriques sont associées aux noeuds représentant les attributs. Dans ce cas la largeur indique le nombre d'accès par d'autres classes et la hauteur représente le nombre d'accès par la classe elle-même. On peut remarquer que toutes ces métriques utilisées permettent de visualiser le couplage, sauf pour le nombre de lignes de codes par méthode. Ces métriques de couplage permettent de localiser les modules qui nécessitent plus d'effort de compréhension pour leur fonctionnement. La couleur permet de rappeler la fonctionnalité de la méthode (par exemple, un nœud rouge est une méthode *Getter*).

La technique de visualisation Class Blueprint permet de visualiser les fonctionnalités de chaque méthode dans une classe et les relations entre les méthodes et les attributs. Cette visualisation permet de distinguer des relations qui autrement auraient été difficiles à déceler autrement qu'en parcourant tout le code source ligne par ligne. De plus, des motifs permettent de reconnaitre rapidement la manière dont interagit une classe.

Les motifs sont identifiable grâce aux métriques de la visualisation comme la taille des nœuds, la manière dont les nœuds sont distribués dans les zones, la couleur des nœuds, les arêtes et la manière dont les méthodes accèdent aux attributs. Par exemple :

- beaucoup de grosses méthodes indiquent une classe qui réalise beaucoup de fonctionnalités.
- une zone interface particulièrement chargée démontre une classe qui agit comme une interface.
- beaucoup de méthodes accesseurs (rouge) et d'attributs (bleu) avec peu d'autres méthodes révèlent une classe qui définit principalement des attributs et des méthodes accesseurs.
- un groupe de méthodes qui forment un arbre d'appel permet de visualiser une décomposition d'un algorithme complexe en petites méthodes qui s'appellent les unes les autres (potentiellement pour un souci de réutilisabilité).
- des attributs qui sont accédés uniformément par un ensemble de méthodes révèlent une certaine cohésion de la classe.

Plus d'exemples sont donnés dans [Lan03b, LD01, DL05].

Peu d'autres techniques de visualisation permettent de faciliter la compréhension du fonctionnement interne des classes. De plus, le Class Blueprint permet également de visualiser la classe dans son contexte d'héritage en affichant comment les sous-classes interagissent avec leurs parents et leurs fils. En effet, comprendre en détails le fonctionnement d'une classe implique également d'étudier le fonctionnement des classes en relation avec elle. Des motifs peuvent également être définis dans le contexte d'héritage de la classe. Ceci permet d'étendre la visualisation avec plus d'informations sur le fonctionnement des classes. Les expérimentations de [DL05] ont montré que l'utilisation de la technique de visualisation Class Blueprint permet une compréhension plus rapide des classes que si l'utilisateur devait explorer lui-même le code source.

3.4 Visualisation de l'architecture

Visualiser l'architecture du logiciel [DOL02] est l'un des sujets les plus étudiés dans le domaine de la visualisation du logiciel [Hat04, GHM05, GHM08, Car08]. Comme nous pouvons le voir dans le tableau 3.1, beaucoup de techniques de visualisations traitent de la visualisation de l'architecture.

Les logiciels à objets sont structurés hiérarchiquement, avec des paquets qui contiennent récursivement des sous-paquets, puis des classes qui contiennent des méthodes et attributs. Il est donc logique de vouloir utiliser cette structuration pour représenter les composants du logiciel. Un autre aspect important de l'architecture des logiciels à objets sont les relations d'héritage et d'appel de méthode.

Visualiser l'architecture consiste donc à visualiser cette hiérarchie et ces relations entre les composants du logiciel. Notons qu'avoir une visualisation permettant une vue globale du système est considéré comme un critère très important et efficace [WC99] pour décider quels composants nécessitent de plus amples investigations. Certaines visualisations vont jusqu'à permettre de zoomer sur certaines parties de la visualisation sans perdre le contexte global [SZ00].

Cette section traite donc des représentations de l'architecture du logiciel, et donc de représentations d'arbres, de graphes et de diagrammes. Nous nous focalisons sur les techniques de visualisation qui s'intéressent à visualiser trois différents aspects du logiciel. Le premier aspect est l'organisation globale du logiciel (paquets, classes, méthodes). Le deuxième aspect est les relations entre les composants du logiciel, qu'elles soient de type héritage ou graphe d'appel. Le troisième aspect est les visualisations qui montrent les métriques logicielles. Comme nous le verrons par la suite, certaines représentations permettent de mieux visualiser certains aspects

plutôt que d'autres.

3.4.1 Visualisation de l'arborescence du logiciel

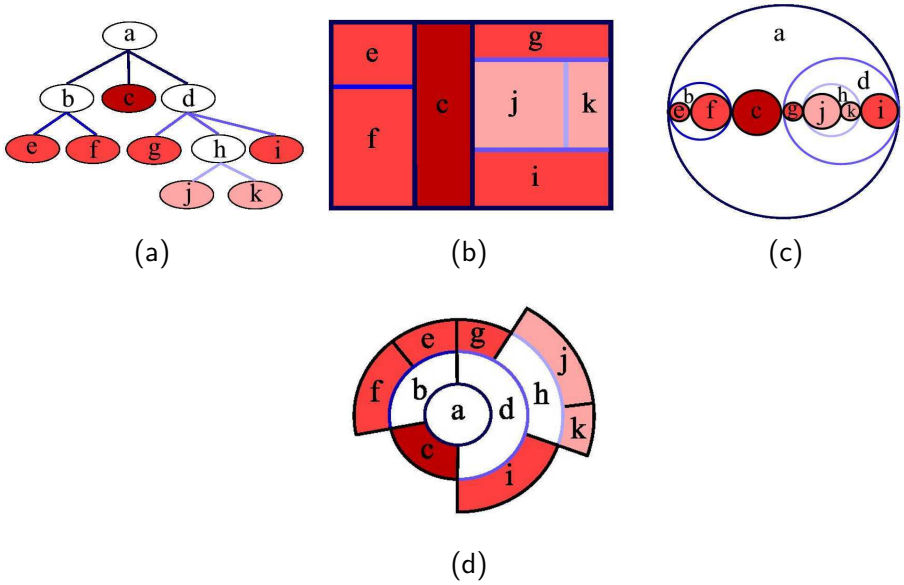


FIGURE 3.3 – (a) Node-Link diagrams (b) Treemap (c) Circular Treemap (d) Sunburst. Toutes ces figures représentent le même système.

L'arborescence du logiciel est l'organisation des répertoires (paquets), des différents fichiers dans les répertoires (classes) et méthodes. Cette arborescence suit une certaine logique d'organisation, regroupant les composants censés réaliser la même fonctionnalité. Il est donc intéressant d'utiliser ce regroupement pour visualiser et comprendre le fonctionnement du logiciel.

L'arbre est la structure de données la plus appropriée pour représenter cette arborescence récursive, mais sa représentation classique avec nœuds et d'arêtes ne semble pas être appropriées dans ce cas précis. En effet, cette représentation devient très vite très large et encombrée car elle n'utilise pas efficacement la place disponible dans la fenêtre de visualisation (Fig. 3.3(a)). De plus, la quantité d'information textuelle présente dans chaque nœud doit être restreinte, sous peine de voir la visualisation devenir illisible à cause d'un trop-plein d'informations [Kos03, KEM07]. Pour cette raison beaucoup de chercheurs dans ce domaine ont proposés des algorithmes d'optimisation de placement de nœuds dans les représentations d'arbres [WS79, LR96, DBETT98, BN01]. Ces visualisations sont principalement des représentations 2D, mais peuvent être adaptées à la 3D. Dans cette section nous présentons plusieurs techniques de visualisation qui montrent l'organisation du code source. La plupart de ces visualisations sont des représentations d'arbre qui peuvent être appliquées à d'autres domaines que la visualisation des logiciels (voir la section 3.4.1 du tableau 3.1 page 17).

Le *Treemap* a été introduit par Johnson et Shneiderman [JS91, Shn92]. Il permet de visualiser l'arborescence complète du logiciel en utilisant la totalité de la fenêtre de visualisation de manière efficace [TJ92] (Fig 3.3(b)). Cette visualisation est générée de façon récursive en divisant (alternativement horizontalement et verticalement) une boite en plusieurs petites boites pour chaque niveau de la hiérarchie. La visualisation permet donc l'affichage de tous les éléments de l'arborescence tout en encodant implicitement le chemin de chaque élément grâce à l'imbrication

des boites. Les figures 3.3(a) et 3.3(b) représentent respectivement une représentation d'arbre sous forme de graphe et sa représentation équivalente sous forme de Treemap (la couleur aide à la compréhension du mécanisme de division des boites). La taille de chaque élément de la représentation est proportionnelle à la taille de ces éléments dans l'arborescence.

Un *Circular Treemap* est une technique de visualisation semblable au Treemap rectangulaire, mais où un paquet est représenté par un cercle. Celui-ci contient d'autres cercles représentant ses sous-paquets directs (Fig. 3.3(c)). Notons que représenter l'arborescence en utilisant des formes circulaires au lieu de formes rectangulaires facilite la visualisation de l'arborescence, car toutes les arêtes et les feuilles de l'arbre restent visibles. Néanmoins, le positionnement des cercles n'est pas optimisé et beaucoup d'espace reste inutilisé. Cette optimisation est un sujet traité dans [WWDW06].

Stasko et Zhang proposent la technique de visualisation nommée *Sunburst* [SZ00]. Celle-ci est une alternative aux techniques de visualisation qui utilisent l'imbrication des éléments pour représenter leurs profondeurs dans l'arborescence. Le Sunburst est une représentation radiale (en anneau) (Fig. 3.3(d)). Les arcs y_i représentés dans le prolongement d'un arc x correspondent aux contributions des fils de l'arc x. Chaque niveau de l'arborescence est représenté par un disque [AH98, Chu98]. Plus le disque est éloigné du centre, plus les éléments représentés sont profonds dans l'arborescence. Le plus petit des disques (centre) représente l'élément *Root*. Les expérimentations dans [SCGM00] ont montré que la réalisation de tâches typiques (telles que la localisation, la comparaison et l'identification de fichiers et paquets dans une grande arborescence) est plus facilement réalisable avec un Sunburst qu'avec un Treemap.

Un inconvénient majeur du Treemap est que tous les éléments (boites) se ressemblent. Ceci ne facilite pas l'identification des classes qu'ils représentent.

Le *Voronoi Treemap* [BDL05] permet de donner des formes irrégulières aux éléments du Treemap. Le but principal est de permettre une meilleure distinction des éléments qu'avec de simples rectangles imbriqués (Fig. 3.4). De cette manière chaque élément a une forme unique.

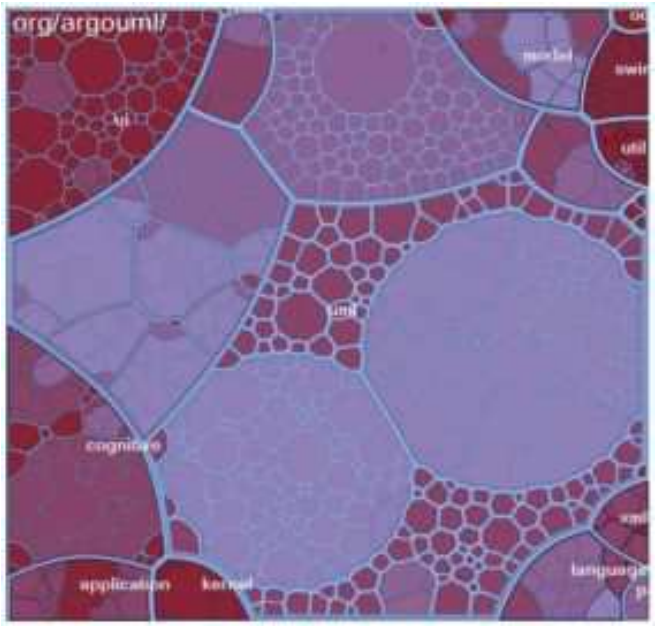


FIGURE 3.4 – Voronoi Treemap [BDL05].

Un autre inconvénient du Treemap est que la structure hiérarchique de l'arborescence est implicite et donc difficile à discerner. Pour faciliter la distinction des niveaux d'imbrication avec

le Voronoi Treemap, les éléments proches de l'élément Root sont assombris alors que les éléments plus profonds sont de plus en plus flous (Fig. 3.4).

Le *Cushion Treemap* [VWVdW99] est également une technique visuelle qui permet d'avoir une meilleure impression de la structure interne du Treemap. Des expérimentations dans [ISS06] ont montré que les utilisateurs préfèrent interagir avec un Cushion Treemap et sont plus rapides pour les tâches d'identification de sous-structure.

Certains travaux ont pour objectif de représenter des Treemap et Circular Treemap en 3D en affichant le sous-arbre de chaque niveau de l'arborescence à un niveau différent d'élévation dans l'espace 3D [CKI99, vHvW03, BCS04, BCK05]. Un Circular Treemap peut être visualisé dans un environnement 3D en transformant les disques en cylindres (Fig. 3.5(d)) [WWDW06]. De cette manière la taille des cylindres donne une meilleure impression de la profondeur des éléments dans l'arborescence (Fig. 3.5). Les expérimentations dans [vHvW03, BCS04] ont montré que les Treemaps 3D permettent effectivement une meilleure visualisation de la profondeur des éléments dans l'arborescence.

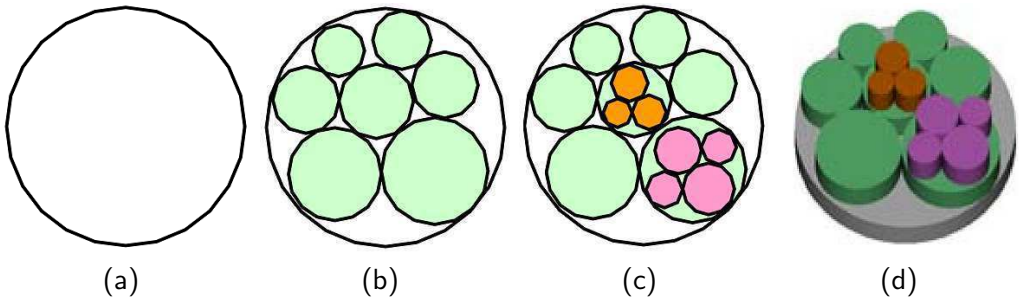


FIGURE 3.5 – Circular Treemap (a) niveau 0 (b) niveau 1 (c) niveau 2 (d) 3D.

Storey, Müller et Wong ont développé l'outil *SHriMP* (Simple Hierarchical Multi-Perspective) [SMK96, WS00, SBM01] qui combine plusieurs techniques de visualisation basées sur les graphes avec des informations textuelles. Le but principal de SHriMP est d'apporter une aide pour naviguer dans le code source des logiciels (Fig. 3.6). En effet, SHriMP regroupe un ensemble de technique de visualisation parmi lesquelles l'utilisateur peut alterner en fonction de ses besoins.

La figure 3.6 montre un Treemap qui représente l'arborescence du logiciel. Les boites colorées représentent les paquets (jaune), classes (vert clair), méthodes (vert et bleu), attributs (rouge). La caractéristique la plus intéressante de SHriMP est que chaque boite peut afficher et éditer son code source. Ceci permet d'avoir à la fois l'architecture du logiciel et une vue miniaturisée du code source dans une même visualisation. La documentation peut également être visualisée de la même manière.

Pour faciliter la navigation du code source, SHriMP intègre un système de liens hypertexte sur les types, les attributs et les méthodes. Lorsqu'un lien est cliqué, une animation s'exécute pour naviguer du lien cliqué vers la cible du lien.

Beaucoup d'attention est portée aux interactions avec la visualisation, car l'interface de SHriMP comprend plusieurs types de zoom : zoom géométrique, sémantique et *Fisheye*. Le zoom géométrique consiste à redimensionner un nœud spécifique dans le graphe imbriqué, tout en cachant l'information du reste du système. Le zoom Fisheye [Fur86] permet à l'utilisateur de zoomer sur une zone, tout en diminuant la taille des autres éléments du graphe qui l'entoure. Ceci aide à préserver le contexte de la partie zommée. Le zoom sémantique offre une vue appropriée d'un nœud en fonction de la tâche en cours.

Grâce à toutes ces fonctionnalités, SHriMP est un outil de visualisation complet, car il

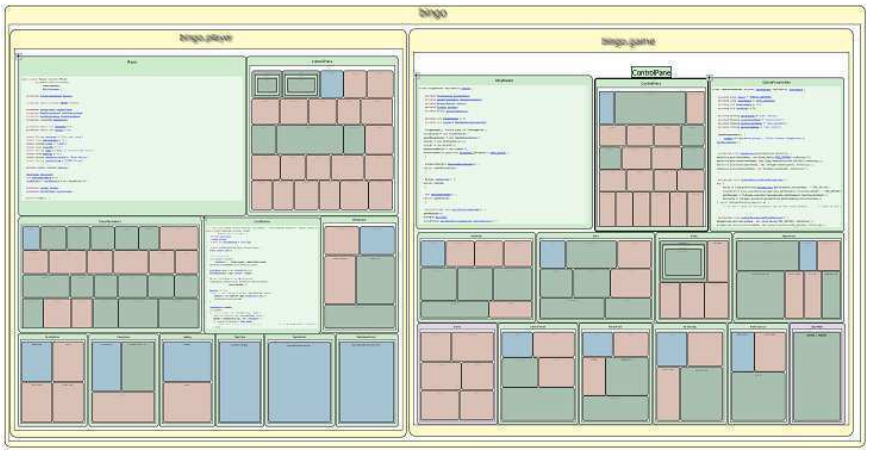


FIGURE 3.6 – The Treemap visualization from the SHriMP application.

permet une exploration efficace de la structure du logiciel et de son code source. En revanche, les métriques logicielles ne sont pas représentées. Les capacités cognitives des utilisateurs sont exploitées au mieux grâce aux animations qui aident à se repérer lors de la navigation d'une partie du code source à une autre [Tud03]. De plus, d'après les travaux de [BS90, RCM93], elles facilitent la compréhension et rendent la visualisation plus agréable.

L'outil SHriMP, avec ses différentes visualisations, a été intégré avec succès dans plusieurs outils tels que l'environnement de développement et reverse engineering Rigi [SM95]. Celui-ci a eu un impact significatif sur le monde de la recherche et le monde industriel [Mül86, SWM97, KM08]. Des expérimentations [SWM00] ont montré que la possibilité de changer de technique de visualisation, tout en gardant la visualisation du code source, sont des caractéristiques qui facilitent grandement la navigation et la compréhension du logiciel. Par conséquent, l'outil SHriMP intégré à Rigi a grandement facilité les tâches de développement et de maintenance du système étudié lors des expérimentations. Une caractéristique qui a également grandement contribué au succès de SHriMP est sa capacité à modifier le code source tout en visualisant le logiciel.

Les visualisations du logiciel qui s'appuient sur une *métaphore du monde réel* représentent le logiciel via un contexte familier à l'observateur. Ceci permet à l'utilisateur de comprendre beaucoup plus rapidement l'information grâce à son rapport à la réalité [GR93, DSGA+00, KM00, Plo02, KvdWVW01]. D'après Dos Santos et al. [RDSGA+00], cette technique permet une reconnaissance plus rapide de la structure globale du logiciel, ce qui évite un des plus gros problèmes des visualisations en 3D, à savoir la désorientation. Les *métaphores du monde réel* favorisent une compréhension "naturelle" des éléments logiciels et donc une meilleure orientation et navigation au sein de la visualisation.

En 1993, Dieberger propose de représenter l'information en utilisant une *métaphore de ville* pour résoudre les problèmes de navigation qui surviennent dans un monde 3D [Die93, Die94, DF98]. En effet, la ville étant un environnement qui nous est familier depuis la naissance, l'orientation au sein d'une ville devrait donc être intuitive. Dans ce chapitre, nous utilisons le terme *métaphore de ville* lorsque le logiciel tout entier est représenté par une seule ville. Nous utiliserons le terme *métaphore de villes* lorsque le logiciel est représenté par plusieurs villes. Habituellement dans les métaphores de villes, les villes représentent :

- les paquets ; dans ce cas, les bâtiments symbolisent les classes.
- les classes ; dans ce cas, les bâtiments symbolisent les méthodes.

La Terre est découpé en pays, les pays en villes, les villes en districts, les districts en rues, bâtiments, jardins, etc. De la même manière, le logiciel est découpé en paquets, les paquets en sous-paquets ou en classes, les classes en classes internes ou en méthodes, les méthodes en lignes de code. Ainsi une analogie avec le découpage artificiel de notre monde et celui du logiciel est possible [Lyn60] en faisant par exemple la correspondance suivante : la Terre représente le logiciel entier, les pays représentent les paquets, les villes représentent les fichiers, les districts représentent les classes et les bâtiments représentent les méthodes. Ce genre de visualisation permet une compréhension à grande échelle du système tout entier grâce à l'analogie qui est faite au monde réel. Knight et Munro [KM99] ont fait partie des premiers à représenter le logiciel en utilisant une métaphore de villes appelée *The Software World* [KM00, CKTM02].

Panas, Berrigan et Grundy [PBG03] pensent que la métaphore doit représenter au mieux la réalité pour permettre une interprétation intuitive du logiciel. Leur visualisation est très réaliste, avec beaucoup de détails (arbres, rues avec trottoirs, lampadaires) (Fig. 3.7).

Tous les chercheurs ne partagent pas leur avis [YG03] et nous sommes convaincus que la métaphore de la ville n'a pas forcément besoin de correspondre exactement à la réalité et que quelques écarts sont permis et même souhaitables. En effet, nous considérons qu'une visualisation simplifiée de la ville aide à davantage à se focaliser sur les informations importantes de la visualisation, sans être distrait par des informations "jolies" et réalistes, mais sans valeur ajoutée pour la visualisation du logiciel.

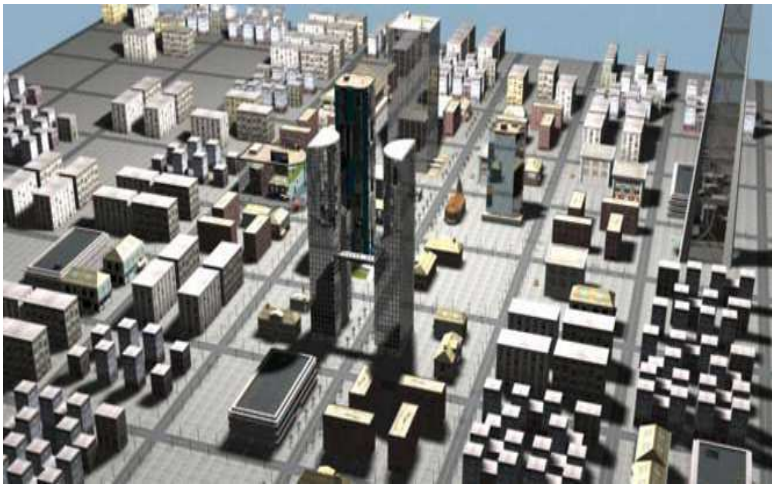


FIGURE 3.7 – Realistic City metaphor representing software from [PBG03].

Toujours en gardant une analogie avec le monde réel, Graham, Yang et Berrigan proposent la métaphore du *système solaire* [YG03, GYB04]. Cette représentation se base sur la manière dont l'univers est organisé. La galaxie représente le logiciel dans son intégralité, puis les différents systèmes solaires représentent les paquets du logiciel (Fig. 3.8). L'étoile centrale de chaque système symbolise le paquet lui-même, alors que les planètes en orbite représentent les classes du paquet. La métrique de profondeur dans l'arbre d'héritage définie dans le chapitre 2 section 2.3 est implicitement représentée par la distance entre l'étoile centrale et la planète qui représente la classe. Ainsi, plus la planète est éloignée de l'étoile centrale, plus la classe est profonde dans l'arbre d'héritage. Les planètes ont deux couleurs différentes : bleu foncé pour les classes et bleu clair pour les interfaces.

La métaphore du système solaire a des propriétés intéressantes, mais elle offre moins de possibilités de représentation de la structure du programme que la métaphore de la ville (avec un

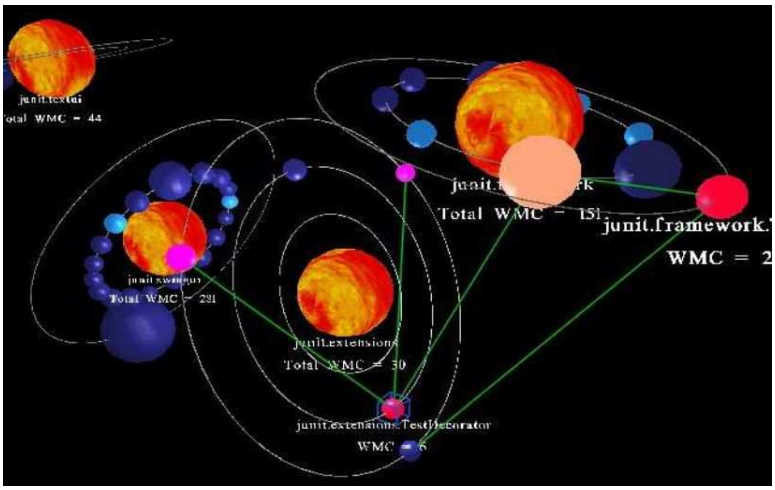


FIGURE 3.8 – La métaphore du système solaire [YG03].

Treemap comme représentation de l'arborescence). Néanmoins, il est possible de gagner plusieurs niveaux de structuration en proposant une métaphore de l'Univers (qui est composé de plusieurs galaxies et ainsi de suite). Avec cette métaphore, nous retrouvons le problème d'optimisation de l'espace que nous signalions pour le Circular Treemap section 3.4.1. En effet, placer des cercles ou des sphère dans un environnement 2D ou 3D est plus volumineux que placer des carrés ou des cubes côtes à côte.

L'avantage important du système solaire est que les systèmes solaires eux-même peuvent être déplacés dans l'espace 3D. L'utilisateur peut donc réorganiser la représentation du logiciel dans l'espace 3D comme il le souhaite pour faciliter sa compréhension. Cette liberté de manipulation et de transformation de la visualisation est très rarement possible avec les autres visualisations du domaine (à l'exception de Sv3D, présenté en section 3.2, qui propose la même fonctionnalité).

3.4.2 Visualisation des relations entre les composants logiciels

Visualiser les relations (statiques) entre les composants du logiciel est une tâche plus complexe que visualiser l'arborescence. En effet, ces relations peuvent être de différentes natures tels que les relations d'héritage, les appels de méthode, les accès aux attributs (voir les détails dans la section 3.4.2 du tableau 3.1 page 17). Pour représenter toutes les relations du logiciel, la représentation en arbre n'est plus adaptée. Ce sont bien les représentations de graphe qui permettent de modéliser au mieux les relations du logiciel. En effet, les *graphes* ont toutes les caractéristiques nécessaires pour représenter les relations entre les composants, en associant les composants aux nœuds du graphe et les relations aux arêtes [Mun97, HMM00, LN03, NL05, SS06]. Visualiser toutes les relations revient donc à visualiser un très grand graphe avec beaucoup d'arêtes de différentes natures.

Sans un placement approprié des nœuds ou une technique efficace de représentation des arêtes, le graphe ne serait pas très compréhensible à cause des arêtes qui se chevaucheraient et qui recouvriraient les nœuds du graphe, le rendant ainsi très confus. De plus, une telle représentation rend presque impossible l'investigation d'une seule relation en particulier.

Une solution qui aide à éviter les amas d'arêtes d'un graphe 2D est de représenter le graphe en 3D [SB99, FBK06]. Ainsi l'ajout d'une dimension supplémentaire offre une liberté supplémentaire au niveau du placement des nœuds. L'utilisateur peut ensuite naviguer dans l'environnement 3D pour trouver une vue sans occlusion qui lui permette de comprendre les relations entre certains

composants.

Un problème récurrent de l'utilisation d'une représentation 3D est que l'utilisateur peut être désorienté. Souvent la solution proposée par l'outil de visualisation consiste simplement à recommencer l'exploration du graphe depuis un même point de départ dans l'espace 3D [RDSGA+00]. Cette solution n'est pas très efficace d'un point de vue cognitif car tout le chemin d'exploration doit être parcouru à nouveau. Une solution proposée par [HWH97] consiste à limiter la liberté de navigation de l'utilisateur pour diminuer les risques de désorientation. Dans le même esprit, les auteurs de [AE05] génèrent automatiquement une animation de la caméra permettant un tour du graphe dans la scène 3D. De cette manière l'utilisateur à beaucoup moins de chance de se perdre dans le graphe car il n'a pas besoin de se concentrer sur ses déplacements.

Un graphe peut être représenté grâce à une matrice carrée avec, pour labels des lignes et colonnes, le nom des composants du logiciel. Le nombre de relations entre la colonne x et la ligne y est écrit dans la matrice, à l'indice m_x, y. Cette représentation est aussi connue sous le nom de *Dependency Structure Matrix (DSM)* [Ste81, BCW01, SJSJ05], et permet de présenter les relations d'un système complexe de manière compacte.

Cette technique de représentation a été appliquée aux logiciels, notamment pour identifier les dépendances entre les paquets et les sous-paquetsdu logiciel. Des travaux pour enrichir les DSM avec des couleurs et des marquages qui permettent l'identification de cycles [BDLP08, LDDB09], et le taux de couplage entre les paquets [AAD+08]. Ainsi les DSM ne sont plus seulement des valeurs numériques dans un tableau mais bel et bien une visualisation qui aide à la compréhension des relations du logiciel.

Les *diagrammes de classes UML* sont probablement le moyen le plus courant pour visualiser le logiciel (au-delà du texte source, bien sûr). Leur but est de représenter certaines relations entre les classes, comme les relations d'héritage, de généralisation, d'association, d'agrégation et de composition. Comme les autres graphes, celui-ci souffre des mêmes contraintes. En effet, plus le diagramme grossit, plus la complexité visuelle croît et donc plus le risque de surcharger la visualisation et de nuire à la compréhension est important (Fig. 3.9(a)).

Un moyen parmi d'autres de réduire la complexité visuelle d'un diagramme de classes UML est de minimiser le nombre de croisements d'arêtes. Des recherches ont révélé que certaines caractéristiques esthétiques pouvaient faciliter la compréhension des diagrammes UML [PAC01, SW05] :

- le placement orthogonal (qui offre une représentation avec peu de chevauchements, peu de superpositions et peu de changements de direction au niveau des arêtes).
- l'écriture du texte sur les liens.
- l'écriture systématique du texte de manière horizontale.
- la fusion des arêtes qui ont la même cible lorsqu'il s'agit de relations d'héritage.

GoVisual est un outil de représentation de diagrammes de classes UML qui prend en compte les caractéristiques esthétiques définies précédemment [GJK+03] (Fig 3.9(b)). En effet, la figure 3.9(b) est indéniablement bien plus compréhensible que la figure 3.9(a) [Eig03].

A l'origine, les diagrammes de classes UML sont des diagrammes 2D, mais comme nous l'avons vu précédemment, il est possible d'ajouter une dimension pour limiter les croisements d'arêtes. Plusieurs travaux de recherche consistent donc à représenter les diagrammes UML en 3D [GK98, GRR99, RG00]. Cependant ces travaux n'ont pas montré un grand succès car les diagramme UML étant déjà complexes en 2D avec beaucoup d'informations textuelles, les représenter en 3D n'augmente pas vraiment leur lisibilité.

Pour palier la complexité visuelle des diagrammes UML en 3D, Irani, Ware et Tingley pro-

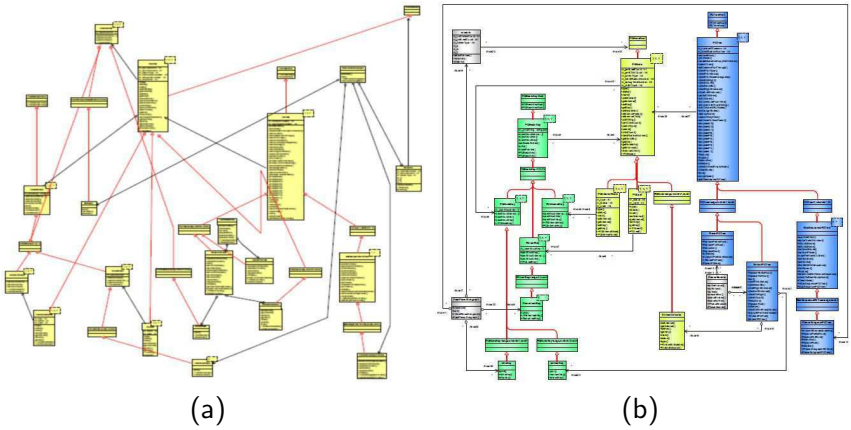


FIGURE 3.9 – (a) Placement classique. (b) Le placement de GoVisual [GJK+03].

posent les *diagrammes de Geon* pour représenter les diagrammes de classes UML en utilisant des formes simples 3.10(a) [IW00, ITW01]. Les Geons sont des objets primitifs en 3D, essentiellement des cônes, cylindres et ellipsoïdes. Cette représentation se base sur une théorie élaborée par Bierderman [Bie87] selon laquelle si une information structurelle peut être mise en correspondance avec la structure d'un objet 3D alors cette information structurelle sera automatiquement perçue [Mar82]. Une caractéristique intéressante des diagrammes de Geon est que la couleur et la texture jouent un rôle secondaire au niveau de la reconnaissance de l'objet, car seule la forme de l'objet a de l'importance. Dans les figures 3.10(a) et 3.10(b), nous pouvons voir un diagramme UML et son équivalent en diagramme de Geon. Tout diagramme UML peut être transformé en diagramme de Geon et vise-et-versa [CE03].

Plusieurs expérimentations [IW03] ont montré que les diagrammes de Geon, grâce à l'utilisation de formes primitives simples, permettent une meilleure compréhension et mémorisation par rapport aux diagrammes UML standards.

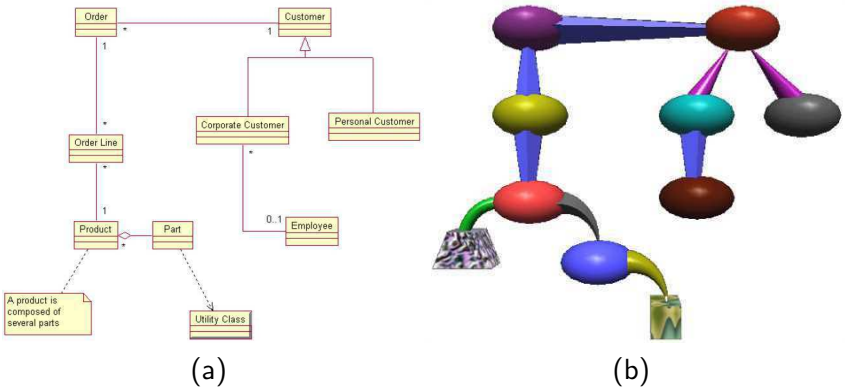


FIGURE 3.10 – (a) UML diagramme (b) le diagramme de Geon équivalent.

Nous avons décrit en section 3.4.1 l'outil *SHriMP* qui permet de visualiser l'arborescence du logiciel mais également d'autres aspects grâces aux autres vues qu'il implante. Dans cette section, nous allons décrire la manière dont SHriMP permet de visualiser des relations.

On retrouve dans SHriMP une visualisation de graphes avec des boites pour représenter les

méthodes et les arêtes pour les relations. La figure 3.11 montre les appels statiques entre les méthodes d'un petit programme Java. Les noeuds sont disposés pour former des arbres d'appels du haut vers le bas. De cette manière, toutes les arêtes montrent plus ou moins la même direction, ce qui facilite la lecture de la représentation graphique.

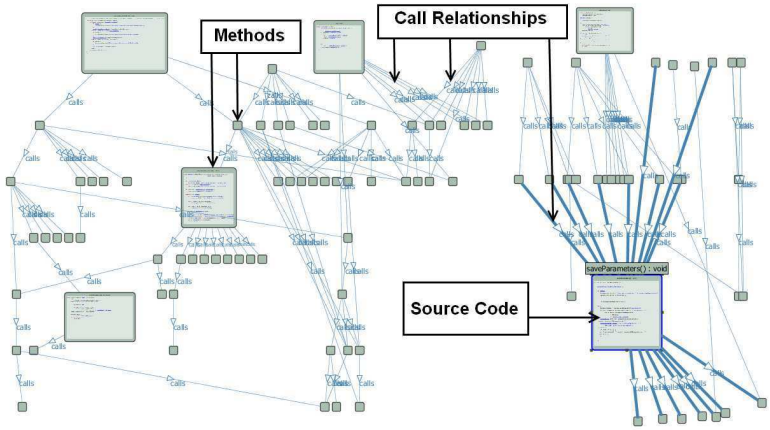


FIGURE 3.11 – Visualisation du graphe d'appel avec l'outil SHriMP.

La figure 3.12 montre une troisième technique de visualisation embarquée avec SHriMP qui a pour but de visualiser à la fois l'arborescence et les relations. Le Treemap permet de visualiser l'arborescence comme expliqué en section 3.4. Les relations sont ensuite représentées par des arêtes colorées (une couleur par type de relation). Les arêtes sont également étiquetées avec le type de la relation : "extends by", "implemented by", "is type of", "calls", "accesses", "creates", "has return type", "has parameter type", "cast to type". Avec cette technique, aucun n'effort n'est fait pour le positionnement des noeuds du graphe. Cette visualisation tiens à garder le découpage en paquets pour visualiser les relations au sein de ce découpage. Par conséquent, les arêtes se croisent et vont dans toutes les directions ce qui peut être très confus. Des travaux ont été réalisés pour réduire la complexité visuelle en utilisant des filtres permettant de visualiser uniquement les arêtes qui répondent à une question spécifique (e. g. afficher seulement les arêtes "calls" qui cible le paquet A) [PGKG08].

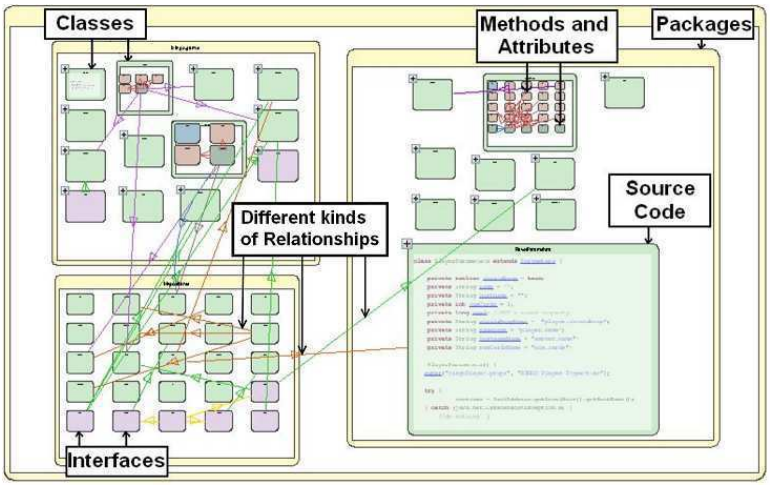


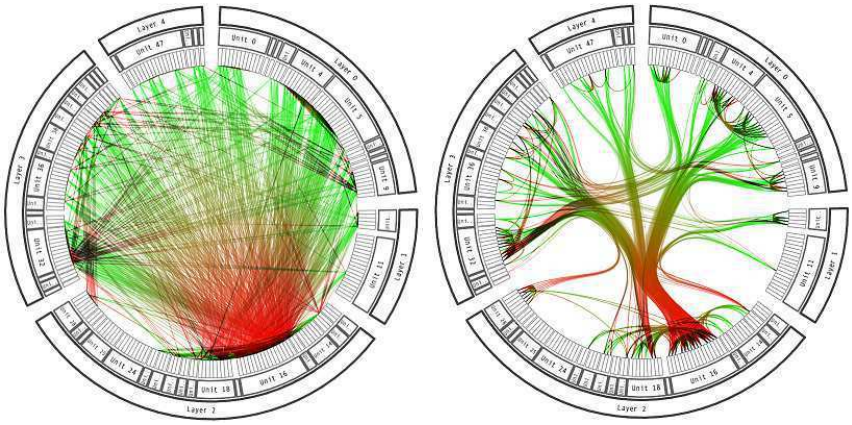
FIGURE 3.12 – Nested boxes visualization from SHriMP.

Comme toujours avec SHriMP, le code source de chaque élément peut être édité directement dans la visualisation. Ainsi le graphe et le code source sont visibles au même moment, ce qui est

très pratique pour comprendre les communications entre les méthodes. Les liens hypertexte et les zooms expliqués en section 3.4.1 sont toujours disponibles avec cette visualisation.

Un autre moyen pour faciliter la compréhension d'un graphe est de modifier la manière dont les arêtes sont représentées. L'outil *Extravis*, créée par Holten [Hol06], utilise la technique du *Hierarchical Edge Bundles* pour visualiser les relations du logiciel. L'idée principale est d'afficher les connexions entre les composants par-dessus la représentation de l'arborescence (quelle qu'elle soit). Comme nous l'avons vu précédemment, visualiser l'arborescence en même temps que les relations permet de comprendre les relations dans le contexte des paquets du logiciel. La figure 3.13 montre un logiciel et son graphe d'appel statique grâce à la méthode des Hierarchical Edge Bundles. Les arêtes sont affichées au milieu du cercle alors que les composants sont placés sur le cercle concentrique de la même manière que décrite en section 3.4.1.

Certains travaux utilisent la courbure des arêtes pour exprimer de manière naturelle la direction de la relation (sans avoir à utiliser de flèches) [FWD+03]. Avec les Hierarchical Edge Bundles, c'est l'interpolation de couleur sur les arêtes qui permet de représenter la direction depuis l'appelant (vert) vers l'appelé (rouge). On peut donc visualiser sur la figure 3.13 un logiciel qui a un paquet (en bas) qui reçoit beaucoup d'appels par d'autres paquets mais n'émet pas d'appel sortant.



(a) $\beta = 0$ (b) $\beta = 0.75$

FIGURE 3.13 – Hierarchical Edge Bundles [Hol06].

Pour réduire le chevauchement des arêtes et le recouvrement de l'arborescence, [WCG03] une technique de courbure et de regroupement des arêtes est employée. Un facteur β permet d'intensifier la courbure des arêtes de la visualisation (Fig. 3.13). Ainsi, les problèmes d'ambiguïté pour distinguer la source et la destination des arêtes sont en partie résolus.

L'outil Extravis, qui implémente les Hierarchical Edge Bundles, permet le filtrage des arêtes et une exploration détaillée des relations.

Les expérimentations de [Hol06] montrent que la majorité des participants ont déclaré que la technique permettait d'avoir un aperçu rapide des relations entre les classes du logiciel. Même si théoriquement, la visualisation peut être utilisée par-dessus plusieurs visualisations d'arborescence, en pratique la plupart des participants ont préféré le placement radial.

Représenter les relations au sein d'une visualisation utilisant une *métaphore de ville et de villes* n'est pas chose facile car tous les éléments reposent sur un même plan 3D. Une solution qui reste cohérente avec la notion de métaphore du monde réel est l'utilisation des voies de

circulation et des allées pour représenter les relations bidirectionnelles [PBG03]. En effet, une vue satellite de la ville permet alors de visualiser les relations. Cependant la gestion du placement des bâtiments afin de minimiser le nombre de croisements est une tâche très complexe [YG03]. En pratique, utiliser les routes dans la ville logicielle comme moyen de représenter les relations devient vite incompréhensible à cause des détours que doivent prendre les arêtes.

Alam et Dugerdil proposent une autre approche pour la visualisation des liens dans la ville logicielle. Leur visualisation, *Evospaces* [AP07, AD07], permet l'affichage des relations grâce à l'utilisation de tubes légèrement courbés d'un bâtiment vers un autre. Un segment coloré en rouge s'anime le long du tube, d'un bâtiment vers l'autre pour symboliser la direction de la relation depuis l'origine vers la destination (Fig. 3.14). Par souci esthétique, des textures sur les bâtiments sont utilisées pour renforcer la sensation de métaphore du monde réel. De la même manière, la courbure que prend le tube ne sert en aucun cas à visualiser la direction de la relation mais seulement pour le côté esthétique. La représentation des relations conduit donc à un écart entre la réalité et la métaphore du monde réel. Toutefois, nous pensons que cet écart n'influence pas vraiment les capacités de l'utilisateur pour comprendre le reste de la visualisation car si quelques objets ne nous sont pas ordinaires, la représentation de la ville quant à elle reste très familière.

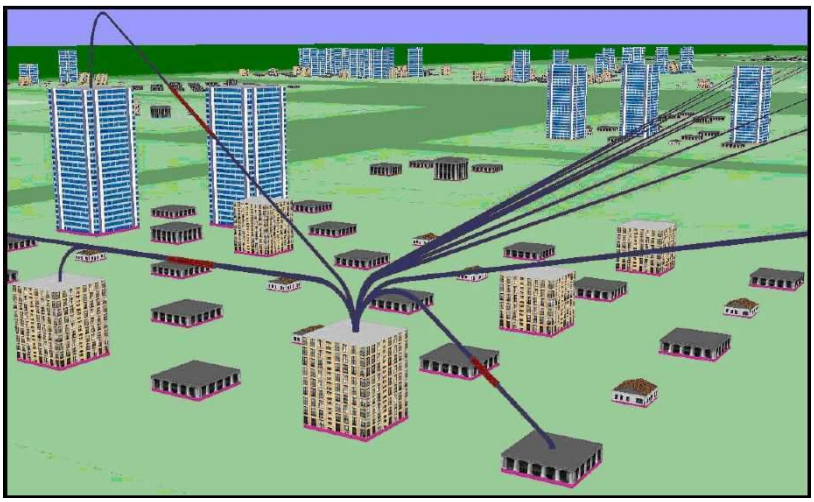


FIGURE 3.14 – Evospaces [AD07].

Une particularité de cette visualisation est qu'elle offre la possibilité de voir le fonctionnement interne des classes, en zoomant à l'intérieur des bâtiments. L'utilisateur peut donc entrer dans un bâtiment et voir des objets représentant les méthodes et les variables locales définies dans ces méthodes. De la même manière que pour les bâtiments, les liens peuvent être visualisés au niveau des méthodes.

Evospaces a plusieurs fonctionnalités telles que l'affichage des valeurs des métriques, le changement d'apparence des objets et l'ouverture du fichier source (dans un éditeur séparé de la représentation graphique). Pour réduire la désorientation de l'utilisateur, un plan 2D en vue de dessus est affiché dans un angle de l'écran pour l'aider à suivre sa position courante dans la ville. Cette combinaison d'une vue 3D et d'un plan en vue de dessus en 2D permet à l'utilisateur de se repérer plus facilement dans l'environnement 3D et de prendre plus rapidement des décisions de navigation [TKAM06].

Panas, Epperly, Quinlan, Sæbjørnsen et Vuduc [PEQ+07] pensent que toutes les informations doivent être visualisées au sein d'une même vue pour permettre une compréhension précise du

logiciel.

Leur visualisation utilise une métaphore de villes pour visualiser beaucoup d'informations sur le logiciel (Fig. 3.15). Les méthodes sont représentées par des bâtiments qui sont placés sur des plateaux bleus symbolisant les classes. Ces plateaux sont eux-mêmes placés sur des plateaux verts représentant les paquets et la hauteur de ces derniers dépend de la profondeur de chacun dans l'arborescence. Grâce à cette élévation, nous avons l'impression que les villes sont placées sur des montagnes. L'arborescence n'est pas visualisée par un Treemap, les plateaux flottent de manière indépendante dans les airs. C'est pour cette raison qu'un ciel et de l'eau (au niveau des plateaux verts) ont été ajoutés à la métaphore pour apporter un meilleur aspect esthétique et pour une meilleure orientation. Nous appellerons donc cette métaphore la *métaphore des îles et des villes*.

Tous les types de relations sont affichés en même temps, ce qui revient à afficher plusieurs graphes simultanément (un graphe par type de relation). Les relations présentées dans la figure 3.15 sont le graphe d'appels de fonctions, le graphe d'appels entre les classes, le graphe d'héritage. Les villes représentent les méthodes du programme, et des arêtes connectent les villes entre elles pour former les graphes des relations. Par rapport aux villes logicielles précédentes qui reposent sur un même plan 3D, la visualisation de Panas *et al.* facilite légèrement la représentation des relations car les villes sont éloignées et à plusieurs niveaux d'élévation.

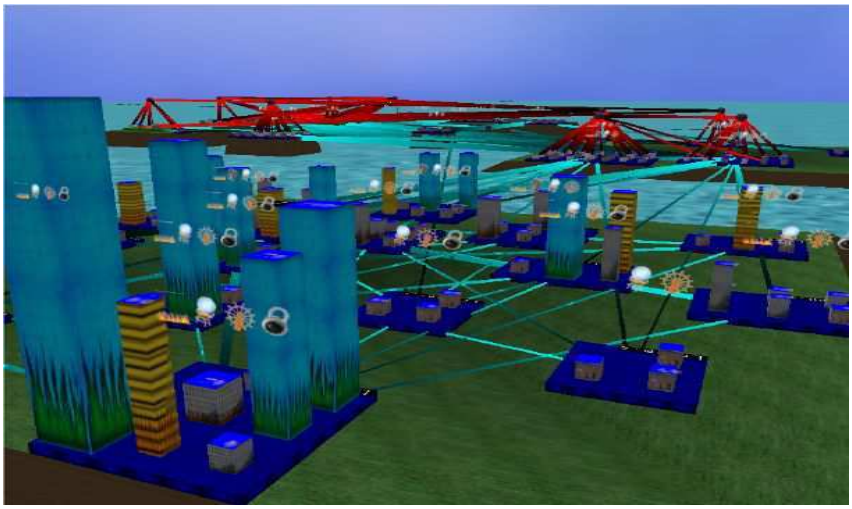


FIGURE 3.15 – La métaphore des îles et des villes [PEQ+07].

La métaphore de villes pour visualiser toutes les informations a été créée avec l'outil Vizz3d (Fig. 3.15) [LP05, PLL05] qui permet de créer des représentations en définissant seulement :

- les objets graphiques,
- le placement
- les associations entre les métriques et les caractéristiques graphiques,

Il n'est donc pas utile de coder la visualisation à la main, ce qui fait de Vizz3D un outil intéressant pour développer des prototypes de visualisation rapidement. L'article [PLL05] montre que Vizz3D est capable de donner des résultats de visualisation très différents.

Nous avons vu précédemment, dans la section 3.4.1, qu'avec la métaphore du système solaire, les éléments peuvent être déplacés librement dans l'espace 3D. Ceci permet de représenter des relations plus efficacement (Fig. 3.8) car il est en effet possible de déplacer les objets pour trouver une vue avec moins de chevauchements. En pratique, il n'est pas possible de visualiser lisiblement tous les relations logicielles de cette manière. Les auteurs ont donc envisagé d'utiliser

la gravité pour représenter les relations de couplage entre les composants mais cette solution ne nous semble pas être judicieuse car les planètes seraient constamment en mouvement ce qui risque de perturber la lecture des autres informations telles que les métriques.

Balzer et Deussen introduisent la visualisation *Software Landscape)* (en français la métaphore du paysage) ([BD04, BNDL04], qui est une métaphore hybride couplant la métaphore du système solaire et la métaphore des îles et des villes. Ces auteurs utilisent des sphères imbriquées pour représenter l'arborescence (Fig. 3.16). Ainsi, chaque sphère contient les sphères correspondants aux sous-paquets directs. Les classes sont représentées par des cercles posés sur un plateau au centre de chaque sphère. Les méthodes et les attributs sont représentés par des boites (ou des bâtiments) au milieu des cercles (Fig. 3.16(a)).

Une propriété intéressante de la métaphore du paysage est que les relations sont représentées de manière hiérarchique. L'idée est de guider les relations pour que chaque arête passe par son élément parent dans l'arborescence (Fig. 3.16(b)). Les arêtes des composants d'un même niveau d'arborescence sont regroupées pour former une seule arête qui est reliée à son tour à la sphère représentant le niveau supérieur. La taille des arêtes est proportionnelle au nombre de relations et la couleur permet de différencier les communications sortantes et entrantes. Ceci permet d'estimer la quantité de relations sortante et entrante pour chaque composant du logiciel. Ainsi nous avons bien une représentation de graphe mais les arêtes forment un réseau hiérarchique qui s'apparente à une représentation d'arbre en 3D. Cette technique permet donc de n'avoir aucun chevauchement d'arête, mais rend difficile le suivi d'une arête en particulier. Evidement, il est possible de visualiser plusieurs types de relations simultanément en utilisant différentes couleurs.

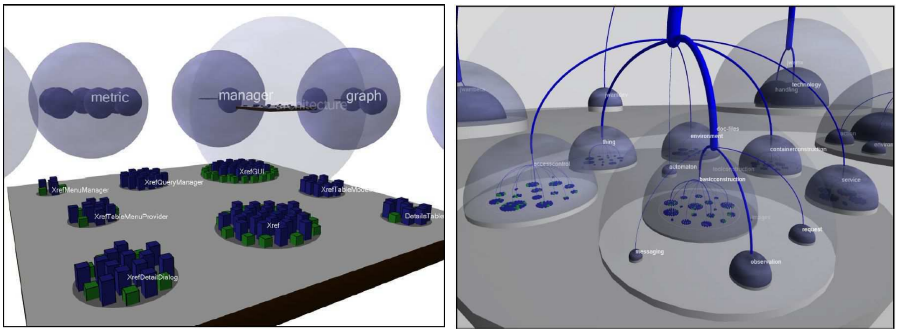


FIGURE 3.16 – (a) spheres imbriqué [BNDL04] (b) réseau hierarchique [BD04].

Peu de temps après avoir proposé la métaphore du paysage, Balzer et Deussen [BD07] proposent une visualisation nommée *Clustered Graph Layouts* qui permet la visualisation 3D de *graphe avec plusieurs niveaux de granularités*, pour permettre une visualisation efficace de très gros graphes (Fig. 3.17). Son principe est de regrouper les nœuds distants en un seul nœud et les arêtes distantes en une seule arête, sans modifier la structure du graphe. Le contenu des nœuds englobants devient visible lorsque l'utilisateur se focalise sur une sous-partie du graphe, sans être perturbé par les détails du reste du graphe. Le taux de transparence s'adapte dynamiquement en fonction de la position de l'utilisateur dans l'espace 3D [YM98]. Plus l'utilisateur est près d'un nœud englobant, plus celui-ci révèle son contenu. Cette caractéristique est très intéressante, car elle permet la compréhension du graphe de manière interactive tout en accentuant les détails sur les zones d'intérêts (proches de l'utilisateur). De plus, les arêtes avec une source et une destination identiques sont courbées et regroupées ensemble, on ne distingue alors qu'une seule arête,

ce qui est plus lisible.

La visibilité des nœuds et des arêtes change donc constamment grâce à des animations qui permettent des transitions fluides entre les différents niveaux de détails. Le graphe avec plusieurs niveaux de granularité produit ainsi une visualisation plus compréhensible pour de très gros graphes car l'information apparait petit à petit. La figure 3.17 montre un graphe avec 1500 nœuds et 1800 arêtes dans 126 groupes d'objets (ce graphe représente les relations d'héritage d'un très gros logiciel).

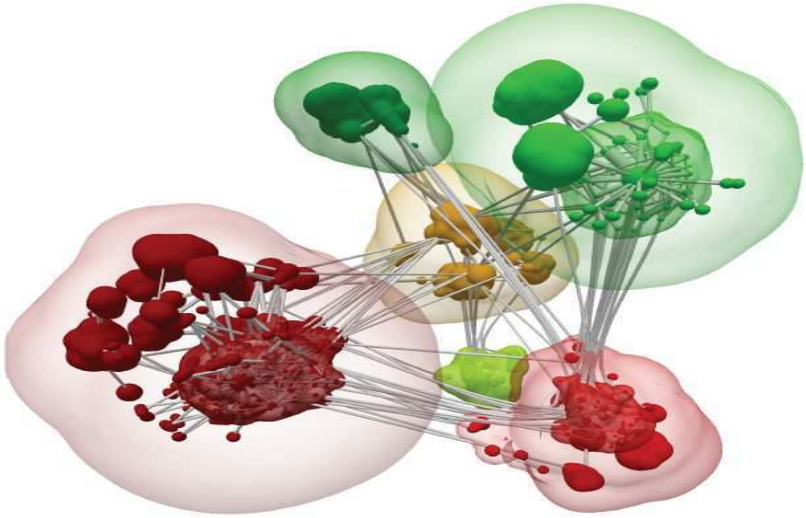


FIGURE 3.17 – Clustered graph layout from [BD07].

3.4.3 Visualisation orientée : métrique

Une métrique logicielle est une mesure d'une propriété d'une partie du logiciel [Fen91, LK94, FP97, ME98, BME+07]. Son but est de quantifier une caractéristique particulière du logiciel telle que la complexité, la structure, le nombre de ressources utilisées ou la stabilité du système. Les métriques logicielles sont intéressantes parce qu'elles apportent des informations sur la qualité de la conception du logiciel [CG90, Kan02, TZ93] et contribuent ainsi à la gestion de cette qualité durant le processus de développement [Bro93].

Les visualisations de logiciel permettent de transformer les valeurs numériques des métriques en caractéristiques visuelles. Ceci permet à l'utilisateur d'avoir une meilleure perception et donc une compréhension plus rapide de l'information [CMS99, Che04, War04]. Le principal défi est de trouver une association métrique-caractéristique efficace pour faciliter le processus cognitif [IC03].

Dans cette section nous décrivons plusieurs techniques pour la visualisation de métriques statiques du logiciel. La plupart de ces visualisations utilisent des concepts définis précédemment dans ce chapitre (voir la section 3.4.3 du tableau 3.1 page 17).

Termeer, Lange, Telea et Chaudron [TLTC05] proposent *MetricView* qui combine métriques et diagrammes de classes UML. Cette technique de visualisation ajoute, par-dessus les diagrammes de classes UML, des diagrammes à barres ou circulaires représentant les valeurs de métriques (Fig 3.18) [DDL99]. Une version en 3D permet de transformer les diagrammes à barres 2D en diagramme à barres 3D pour mieux visualiser la valeur de métrique représentée par chaque élément.

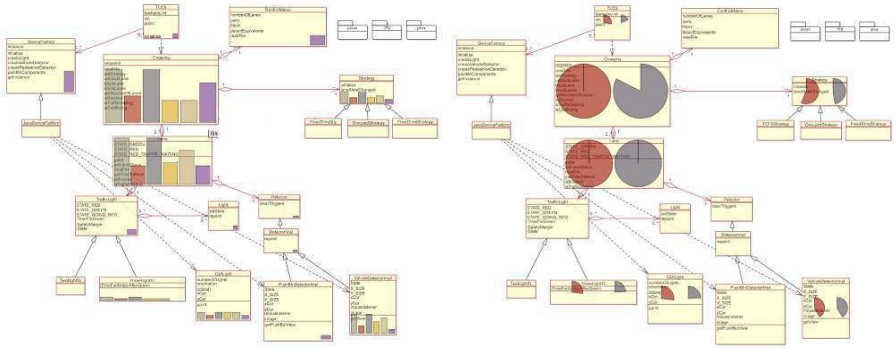


FIGURE 3.18 – (a) MetricView avec diagrammes à barres (b) MetricView avec diagrammes circulaires

Le principal défaut de cette technique est le recouvrement du diagramme UML par les diagrammes à barres et circulaires, tout particulièrement en affichage 3D. Ce problème peut être partiellement résolu en déterminant un taux de transparence pour les diagrammes à barres et circulaires.

Un autre défaut que nous remarquons est que la taille des diagrammes de métriques est liée à la taille de l'élément du diagramme UML. Ainsi les classes qui prennent plus de place dans la représentation ont également les plus gros diagrammes de métrique. Ce choix est très étrange, et même trompeur, car il donne l'impression que les valeurs de métrique de ces classes sont plus importantes que les autres classes alors que ce n'est pas forcement lié.

Pour pallier le problème de recouvrement des diagrammes UML par les diagrammes de métriques, Byelas et Telea introduisent une technique nommée *Area Of Interest (AOI)* (en français zones d'intérêts) [BT06, BT09]. Chaque zone d'intérêt regroupe les éléments du diagramme UML qui partagent une caractéristique commune (Fig. 3.19). Elle est représentée par un contour qui englobe les éléments du diagramme UML, sans changer leurs positionnements. De cette manière, Byelas et Telea évitent d'encombrer la visualisation avec des icônes qui recouvriraient le diagramme. Chaque zone d'intérêt a sa propre texture (lignes horizontales, lignes verticales, lignes diagonales et cercles). Les textures sont simples et assez faciles à différencier lorsqu'elles se chevauchent parce qu'un nouveau motif est créé au niveau du chevauchement. De la transparence et des effets d'ombre sont utilisés pour mieux percevoir la délimitation des zones d'intérêt et minimiser l'effort cognitif. L'intérieur des zones peut ensuite être coloré grâce à un dégradé de couleur allant du rouge vers le bleu, utilisé pour représenter respectivement de grandes et petites valeurs de métrique. La couleur et la texture permettent donc de visualiser la valeur de métrique représenté dans chaque AOI.

Comme les diagrammes de classes UML se focalisent sur les classes du logiciel, les métriques présentées grâce aux AOI seront plutôt orientées classes. Par exemple, le système de la figure 3.19 représente plus de 50 classes avec 7 zones d'intérêt. La zone (A1 : GUI) regroupe les classes qui définissent l'interface graphique du programme. Les classes correspondant au point d'entrée du programme sont en (A5 : main) et les classes réalisant les fonctionnalités principales du programme sont dans la zone (A6 : core).

Byelas et Telea définissent le taux de participation $p_{i,j}$ de la classe C_j dans la zone A_i (avec C_j appartenant à A_i) comme le pourcentage de code de la classe C_j qui est spécifique à la zone A_i. Ainsi une classe d'affichage graphique en OpenGL a $p = 0.5$ si la moitié de son code n'est pas

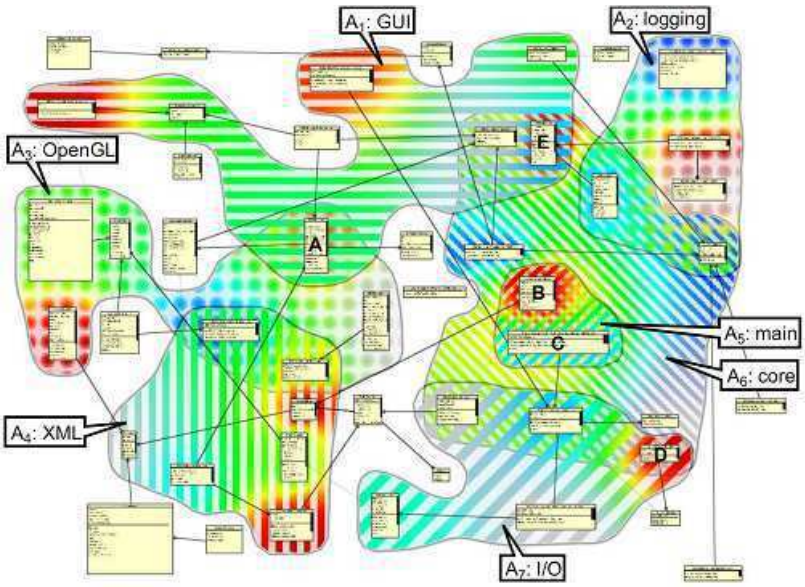


FIGURE 3.19 – Zones d'intérêt [BT09].

spécifique à OpenGL. Les auteurs cherchent à visualiser le cohésion en regardant si les classes du logiciel font bien les tâches qui leurs sont demandées ou si le logiciel à des problèmes de modularité.

Plusieurs remarques peuvent être faites grâce à la visualisation du logiciel de la figure 3.19 :

- 12 classes sur 29 appartiennent à deux zones et participent donc à deux caractéristiques et aucune classe n'appartient à trois zones d'intérêt, ceci montre une bonne modularité du système.
- La classe B est en rouge dans la zone A5 et A6, elle est donc fortement impliquée dans les deux zones.
- La classe E est en rouge dans la zone A6 et en bleue dans la zone A1 : c'est une classe avec beaucoup de fonctionnalités et qui, dans une moindre mesure, participe en tant que point d'entrée du programme.

Les auteurs mentionnent que si plus de 3 zones se chevauchent, les textures et les couleurs deviennent difficiles à comprendre. Néanmoins, cette technique permet de définir jusqu'à 10 zones d'intérêts, chacune pouvant visualiser une métrique, sur un diagramme qui peut avoir jusqu'à 80 classes. Nous pensons néanmoins que cette visualisation peut très rapidement causer de la fatigue visuelle à cause des textures et du grand panel de couleurs. Sa montée en charge semble donc problématique.

La *Polymetric View* [Lan03b, DDL99, LD03], créée par Lanza, Ducasse et Demeyer, est une visualisation qui utilise des formes géométriques simples dont la taille et la couleur peuvent être modifiées en fonction des valeurs de métrique ou d'une caractéristique particulière (Fig. 3.20). Cette technique de visualisation est implantée dans l'outil *CodeCrawler* [Lan03a] qui utilise le *Moose reengineering framework* [DLT01, DGN05]. L'outil est également disponible en plug-in pour l'IDE Eclipse, ce qui lui procure une plus grande visibilité auprès des communautés de développeurs. La Polymetric View est une représentation de graphes où la forme et la taille de chaque élément est configurable afin de pouvoir créer plusieurs vues différentes en fonction des associations entre métriques et caractéristiques visuelles. Avec la Polymetric View, les auteurs utilisent uniquement des métriques logicielles *directes* (qui ne dépendent pas d'autres métriques) pour avoir des métriques faciles à interpréter, précisément définies et qui quantifient une seule

caractéristique du logiciel.

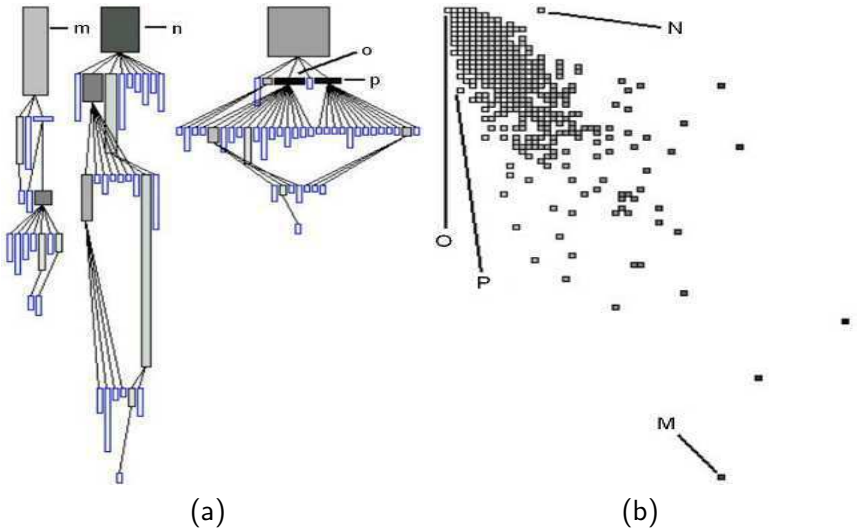


FIGURE 3.20 – Polymetric View [DDL99] : (a) arbre d'héritage et (b) graphe de corrélation.

Toutes les caractéristiques visuelles du graphe dépendent des valeurs des métriques : la largeur, la taille, la position horizontale et verticale, la couleur des nœuds, la largeur et la couleur des arêtes. Cette liberté au niveau de l'association métrique-représentation permet de créer une vue appropriée en fonction des besoins de l'utilisateur. Par exemple, la figure 3.20(a) montre l'arbre d'héritage où les nœuds représentent les classes et les arêtes les relations d'héritage. Le but de cette visualisation est de montrer comment les méthodes sont réparties dans l'arbre d'héritage. Ainsi, la largeur et la hauteur des nœuds représentent respectivement le nombre de descendants de la classe et le nombre de méthodes définies dans la classe. La couleur représente le nombre d'enfants immédiats de la classe. Nous constatons que la figure 3.20(a) montre que les classes m et n semblent être bien conçues car toutes les sous-classes sont plus hautes que larges, ce qui signifie qu'elle héritent des méthodes définies dans les classes parents et que les fils deviennent de plus en plus petits ce qui signifie qu'ils définissent moins de nouvelles méthodes. L'architecture de la figure 3.20(a) met bien à profit l'héritage de méthodes Au contraire, les classes o et p ne définissent pas de fonctionnalités qui sont partagées par leurs sous-classes. En effet, o et p sont petites, larges et noires ce qui signifie qu'elles ont peu de méthodes et beaucoup d'enfants immédiats qui sont des feuilles de l'arbre d'héritage.

Dans la figure 3.20(b), les nœuds représentent les méthodes, les couleurs et les coordonnées horizontales de ces nœuds symbolisent le nombres de lignes de code dans la classe, les coordonnés verticales des nœuds sont déterminées par le nombre d'instructions. Cette visualisation montre la taille de toutes les méthodes et permet de détecter aisément les méthodes vides et les très grosses méthodes. La figure 3.20(b) permet de visualiser que la méthode O n'inclut aucune instruction et que la méthode N contient plusieurs lignes de code mais aucune instruction. Ces méthodes sont peut-être des méthodes inutiles qu'il faudrait supprimer du logiciel pour plus de clarté dans le code. D'un autre côté, la méthode P a plus d'instructions que de lignes de code, cela indique que le code est probablement mal formaté. La méthode M a, quant à elle, beaucoup d'instructions et beaucoup de lignes de codes : elle semble être un bonne candidate pour être ré-implémentée sous forme de plusieurs petites méthodes.

D'autres techniques de visualisation comme l'histogramme, le vérifieur, la confrontation et le cercle sont des modèles de correspondance métriques-visualisation qui sont décrits dans [Lan03b,

DDL99, LD03].

Holten, Vliegen et Wijk utilisent la texture et la couleur pour visualiser deux métriques logicielles avec un Treemap (Fig. 3.21) [HVvW05]. Ce choix est basé sur le fait que le système de perception de l'être humain est capable de différencier très rapidement des textures et des couleurs différentes [HE98]. Un dégradé de couleurs est utilisé pour représenter une métrique et une distorsion de texture pour en visualiser une autre. L'utilisation de ces deux caractéristiques visuelles permet d'avoir une forte densité d'information puisque la texture peut être appliquée sur la couleur. Cette visualisation permet donc de faire directement des corrélations entre les métriques, le but étant de trouver des motifs qui révèlent des problèmes de conception.

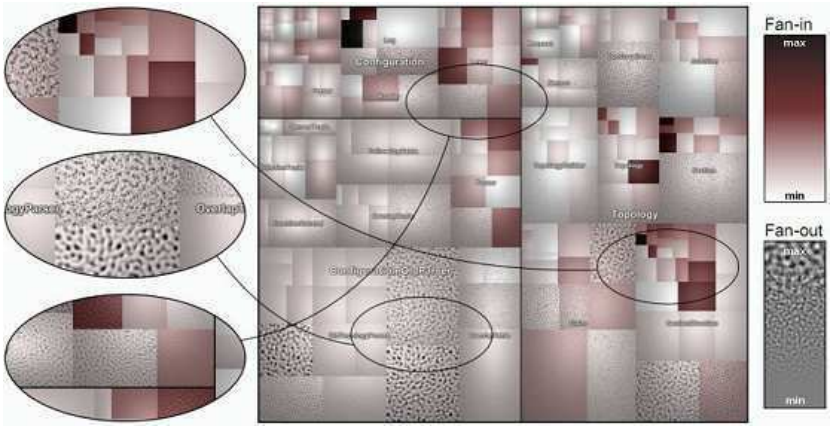


FIGURE 3.21 – Treemap avec couleurs et textures [HVvW05].

Dans le Treemap de la figure 3.21 les méthodes sont représentées par les boites les plus imbriquées. Les classes et paquets sont donc représentés implicitement par l'imbrication du Treemap (voir section 3.4.1). Les deux métriques sont :

- le nombre d'appelants (FANIN) est visualisé grâce à un dégradé de couleurs allant du blanc vers le rose puis le rouge et enfin le noir.
- le nombre d'appelés (FANOUT) est visualisé grâce à une distorsion de la texture. Plus la valeur est grande, plus la texture est distordue.

Dans les langages à objets, il est préférable d'avoir des méthodes avec un FANIN élevé et un FANOUT bas : cela montre respectivement qu'une méthode est importante (utilisée) et qu'elle dépend faiblement d'autres méthodes. La première ellipse en haut de la figure 3.21 montre les méthodes avec un FANIN élevé et un FANOUT bas. Ce premier motif correspond donc aux méthodes qui semblent bien conçues. En revanche, la seconde ellipse de la figure 3.21 montre un motif où des méthodes ont un FANIN bas et un FANOUT élevé, ce qui signifie que la méthode est peu appelée et fait appel à beaucoup d'autres méthodes. Cette deuxième ellipse révèle donc une méthode qui pourrait faire l'objet d'une re-conception pour limiter le FANOUT. Néanmoins, le FANIN étant bas, cette méthode n'est pas très sollicitée et n'est donc pas forcément prioritaire pour une re-conception. Enfin, la dernière ellipse de la figure 3.21 présente des méthodes avec un FANIN élevé ou un FANOUT moyen, ce qui peut indiquer un problème de conception.

Langelier, Sahraoui et Poulin [LSP05] pensent qu'une visualisation simple est cruciale pour optimiser la cognition. Leur visualisation *VERSO* utilise de simples boites pour représenter les classes et des cylindres pour les interfaces (Fig. 3.22). Le système visuel humain étant très efficace pour repérer des motifs [War04], ces formes simples facilitent ainsi leur détection. De cette manière, le risque d'avoir un excédent d'informations est limité.

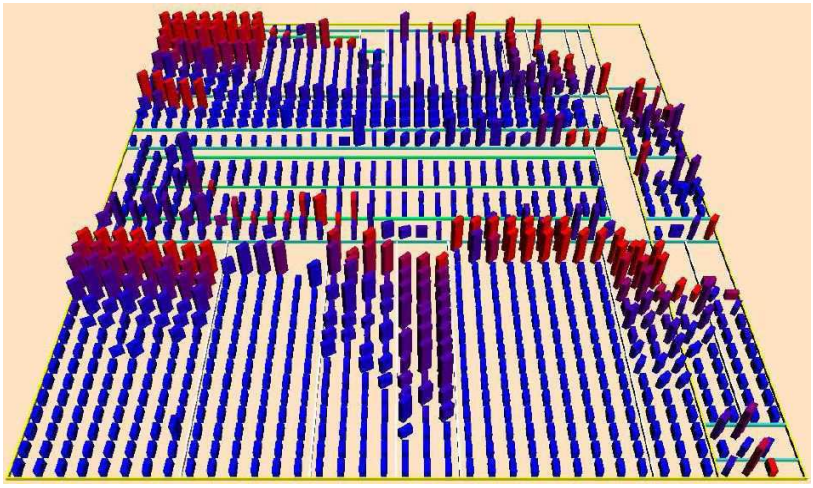


FIGURE 3.22 – VERSO [LSP05].

Les boites ont trois caractéristiques graphiques qui peuvent être associées à des valeurs de métriques. Les auteurs proposent une association métrique - représentation visuelle avec un sens sémantique proche du monde réel :

- La hauteur des boites est naturellement associée à la taille du code source.
- La couleur est associée au couplage. Le rouge indique un couplage élevé, ce qui est considéré comme inquiétant pour les langages à objets. Or la couleur rouge est généralement associée à la notion de danger, dans la vie courante occidentale.
- L'angle de positionnement des boites symbolise un manque de cohésion car des boites qui ne sont pas dans la même direction que celles alignées correctement donnent l'impression de ne pas faire partie du même groupe.

L'arborescence du logiciel est visualisée grâce à un placement de type Treemap ou Sunburst (Section 3.4) et les classes sont représentées par des boites placées dans leurs paquets respectifs. Les relations d'héritage ne sont pas représentées directement mais peuvent être visualisées avec l'utilisation d'un filtre qui met en évidence les boites qui correspondent aux enfants de cette classe. Les autres classes deviennent transparente pour permettre de se concentrer uniquement sur le sous-ensemble d'éléments filtrés. D'autres relations UML telles que : "in", "out", "in/out", "agrégation , "généralisation", "implémentation" et "invocation" peuvent être filtrées de la même manière.

Une caractéristique importante de VERSO est qu'il place l'utilisateur au centre de toutes les décisions : c'est l'utilisateur qui va détecter les anomalies de manière semi-automatique, grâce à la détection de motifs [DSP08]. Par exemple un *Blob Anti-Pattern* [BMMIM98] est une anomalie qui dépend de plusieurs critères. En effet une classe Blob est une classe avec une complexité élevée, une faible cohésion et dont les classes qui utilisent ce Blob sont peu complexes, très cohérentes et profondes dans l'arbre d'héritage [DSP08]. Pour permettre la détection de cette anomalie, la complexité est représentée par la taille des boites, le manque de cohésion par l'angle et la profondeur dans l'arbre d'héritage par la couleur. Le filtre "association out" est ensuite utilisé sur les classes qui montrent une complexité élevée et un manque de cohésion (l'espace de recherche est ainsi restreint). Si les classes filtrées sont petites (peu complexes), droites (cohérentes) et bleues (profondes dans l'arbre d'héritage), alors la classe est probablement un Blob.

Le processus pour détecter les anomalies de conception peut sembler complexe, mais les auteurs font remarquer que certains problèmes existent avec la détection automatique des anomalies [DSP08]. Premièrement, il n'existe aucun consensus qui permet de déclarer avec certitude

que certains éléments font baisser la qualité globale du système. Deuxièmement, une détection automatique de beaucoup de candidats potentiels n'aide pas vraiment si ces candidats sont des faux positifs. Les auteurs affirment que la détection est bien plus efficace et plus utile lorsqu'elle est faite après une inspection humaine assistée par la visualisation. Les expérimentations [DSP08] confirment qu'utiliser VERSO pour trouver des anomalies est plus rapide et plus efficace que d'inspecter le code source manuellement. La détection des anomalies est également plus précise avec VERSO, car le nombre de faux positifs découverts est moins important que sans utiliser VERSO.

CodeCity est une visualisation créée par Wettel et Lanza. Comme son nom l'indique, cette visualisation s'appuie sur la *métaphore de la ville* [WL07b]. Dans CodeCity, les paquets sont des districts représentés grâce à un Treemap et les classes sont représentées par des bâtiments placés dans le district correspondant à leur paquet (Fig. 3.23).

Les auteurs de [PBG03] proposent également de regrouper les classes en districts mais aussi que la métrique de couplage entre les classes détermine la position des bâtiments. Ainsi, plus les bâtiments sont proches, plus le couplage entre ces bâtiments est grand. Ce type de placement est intéressant parce qu'il permet de visualiser de manière intuitive le couplage entre les classes sans utiliser la représentation traditionnelle à base de liens. Par contre, le placement des bâtiments peut changer de manière significative d'une version à une autre, ce qui peut compliquer et troubler la compréhension de la visualisation [HDM98].

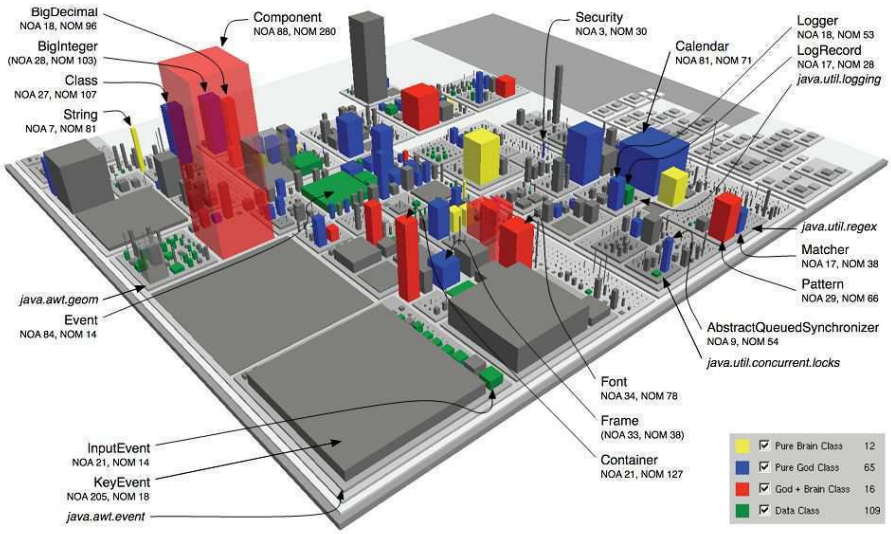


FIGURE 3.23 – Carte des problèmes de conception avec CodeCity [WL08b].

La forme des bâtiments dépend des métriques associées aux classes, l'association métrique-représentation par défaut permet de faire une correspondance intuitive et sémantique avec le monde réel. Le nombre de méthodes par classe (NOM) est représenté par la taille du bâtiment et le nombre d'attributs par classe (NOA) est symbolisé par la largeur du bâtiment, ce qui permet donc de visualiser la quantité de fonctionnalités en fonction du volume du bâtiment. Une classe avec beaucoup de méthodes mais peu d'attributs est représentée par un grand bâtiment très fin (ressemblant à une antenne). Au contraire, une classe avec beaucoup d'attributs et peu de méthodes est représentée par un bâtiment assez petit et large (ressemblant à un parking). Les auteurs affirment que cette visualisation permet de révéler les grands écarts qui existent entre les classes et aide ainsi à bien comprendre la distribution des fonctionnalités du système.

Avec une telle correspondance métrique-représentation, presque tous les bâtiments ont des formes différentes. Ceci peut être problématique car les recherches de gestalt [Tod] ont montrées que l'être humain est capable de différencier de manière efficace jusqu'à six tailles différentes d'un même objet. Les auteurs de CodeCity [WL07a] ont donc décidé de limiter le nombre de tailles différentes des bâtiment à 5 pour réduire la charge cognitive. Ainsi l'utilisateur peut reconnaitre efficacement les bâtiments similaires et mieux catégoriser les classes [Few04]. De plus, l'utilisation de seulement 5 tailles différentes réduit grandement la complexité visuelle et rend la visualisation plus réaliste et familière et donc plus facile à naviguer. Dans la visualisation EvoSpace (section 3.4.2), les auteurs ont accentué un peu plus l'idée d'avoir uniquement quelques tailles de bâtiments différents en associant une texture différente pour chaque taille de bâtiments. De cette manière, les textures utilisées renforcent la métrique associée à la taille du bâtiment en utilisant une texture représentant une simple maison pour les bâtiments les plus petit, puis des textures rappelant (de manière croisante) une mairie, des appartements et des bureaux d'entreprise (Fig. 3.14).

Dans CodeCity, les possibilités de filtrage et de navigation sont classiques par rapport aux autres techniques de visualisation. La couleur et la transparence sont utilisées pour faire ressortir certains bâtiments. La camera peut être bougée librement dans l'espace 3D mais les bâtiments ne peuvent pas être traversés pour limiter les problèmes de désorientation.

CodeCity implémente également une analyse automatique des métriques du logiciel afin de détecter les problèmes de conception [WL08b]. Les classes pour lesquelles un problème de conception est détecté sont identifiées par une couleur correspondant au problème. Ainsi les auteurs de CodeCity proposent une technique de visualisation nommée *Disharmony Maps* (en français la carte des problèmes de conception) de la ville procure un moyen simple pour se focaliser sur les classes potentiellement problématiques (Fig. 3.23). Néanmoins, comme indiqué par les auteurs de VERSO, nous pensons que l'automatisation complète de ce processus n'est pas toujours fiable et qu'une recherche semi-automatique peut aider à mieux comprendre quelles sont les raisons des problèmes de conception trouvés.

CodeCity permet également de représenter le logiciel au niveau de la méthode [WL08b] en utilisant les briques qui composent chaque bâtiment pour représenter les méthodes. De cette manière la carte des problèmes de conception peut être visualisée au niveau de la méthode et permet donc de localiser précisément les éventuels problèmes de conception (Fig. 3.24).

D'autres techniques de visualisation proposent de visualiser directement les problèmes de conception calculés de manière transparente par l'outil de visualisation. Avec la technique de visualisation présentée dans la section 3.4.1, les auteurs de [PBG03] associent la texture des bâtiments à une métrique de qualité du code source. De cette manière, les classes de mauvaise qualité apparaissent avec la texture d'un vieux bâtiment usé, ce qui semble assez intuitif.

La technique de visualisation *CocoViz* introduite par Boccuzzo et Gall [BG07, BG08], est une technique qui se focalise essentiellement sur la visualisation des problèmes de conception. L'idée principale est d'attirer le regard de l'utilisateur en représentant des objets disproportionnés au milieu d'objets correctement proportionnés. Pour cela, les auteurs choisissent des objets simples du monde réel et utilisent des métriques logicielles pour déterminer la taille de ces objets. Par exemple, la *métaphore de la maison* est en fait une simple représentation 3D de maison grâce à un cône pour le toit et un cylindre pour le corps de la maison. La taille et la largeur des cônes et cylindres sont associées aux métriques du logiciel après avoir appliqué une normalisation des métriques. Les métriques doivent être soigneusement choisies afin de pouvoir détecter d'éventuels problèmes de conception. Si l'objet apparait disproportionné alors il existe

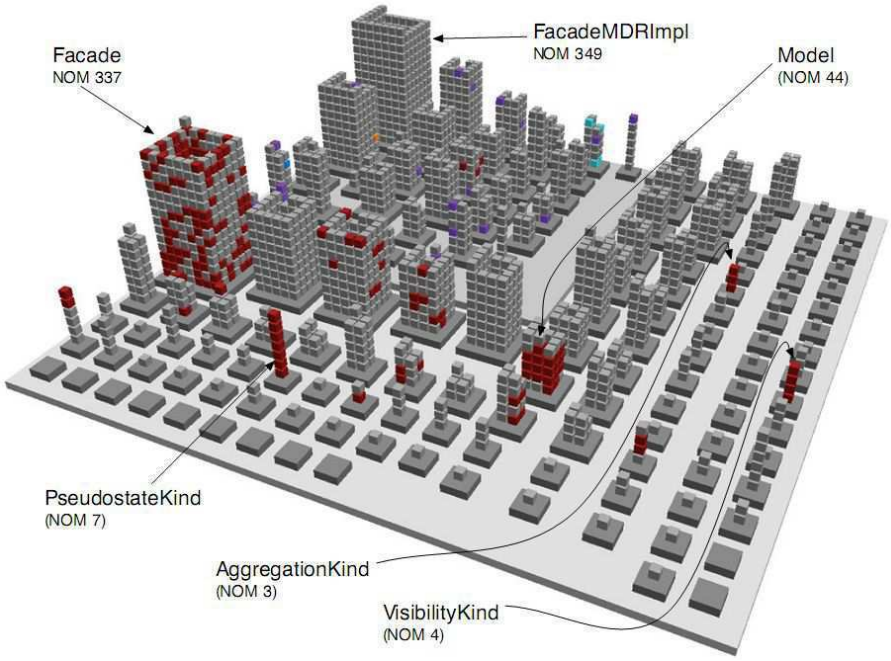


FIGURE 3.24 – La carte des problèmes de conception avec CodeCity au niveau de la méthode[WL08b].

probablement un problème de conception de la classe représentée. Par exemple, la métrique : NOM (Nombre De Méthodes) peut être associée à la largeur du cône et la métrique : LOC (Nombre de Lignes De Codes) associée à la hauteur du cône. Ainsi les classes avec beaucoup de lignes de code et peu de méthodes (ce qui est habituellement considéré comme une mauvaise conception) seront représentées avec un toit très grand, mais très fin. Cette erreur de proportion sera alors interprétée comme un éventuel problème de conception.

Dans la métaphore des villes et iles présentée en section 3.4.2, les auteurs déclarent que montrer beaucoup d'informations sur une même et unique vue est plus efficace que de disperser l'information sur plusieurs vues [PBG98]. Ainsi, leur visualisation présente l'architecture, les relations, les métriques sur une seule et unique vue [PEQ+07] (Fig. 3.15). Cette vision est opposée à celle des auteurs de SHriMP (voir section 3.4.1) qui proposent au contraire plusieurs vues pour visualiser le logiciel.

L'argument invoqué par les auteurs de la visualisation unique est que l'utilisateur doit se familiariser une bonne fois pour toutes avec la visualisation et qu'après cela il est capable de mieux la maitriser cet environnement. De cette façon, l'utilisateur serait plus à l'aise pour naviguer dans l'espace 3D. Puisque toutes les métriques sont affichées simultanément sur la même vue, les auteurs ont ajouté des icônes 2D par-dessus les bâtiments pour coder des informations sur les métriques. La taille, largeur, profondeur et texture de tous les objets graphiques sont également associés à des valeurs de métriques.

En résumé cette technique de visualisation apporte une représentation très complète du système, mais le problème sous-jacent est bien entendu l'excès d'informations [PBG98]. L'utilisation de la métaphore de la ville permet de pallier en partie ce problème grâce au côté intuitif d'une métaphore du monde réel. Cependant, aucune étude empirique n'a été conduite sur la visualisation de la ville très réaliste donc il est difficile d'affirmer l'efficacité cette technique de visualisation. Néanmoins, nous sommes d'accord pour dire que la métaphore de la ville permet de représenter beaucoup d'informations implicitement sans altérer la cognition.

Au contraire, la métaphore du système solaire présentée en section 3.4 permet seulement de représenter une seule métrique grâce au diamètre des planètes. Les auteurs du système solaire proposent donc 5 modes où chaque mode permet la visualisation d'une métrique différente. Cette conception conduit l'utilisateur a constamment changer de vue pour consulter toutes les informations, ce qui peut être fatiguant et risque d'induire en erreur. Les métriques disponibles sont celles définies dans le fameux ensemble CK qui a été validé empiriquement à plusieurs reprises comme étant un bon indicateur de la qualité du système et un bon prédicteur d'erreurs dans le logiciel [BBM96, SK03, OEGQ07].

3.5 Visualisation de l'évolution du logiciel

Visualiser l'évolution du logiciel semble être plus complexe que de visualiser le logiciel à l'instant t car la dimension temps ajoute une grande quantité d'informations. Néanmoins, étudier l'évolution du logiciel permet de mieux expliquer son état actuel en analysant chaque version et chaque changement qu'a subi le logiciel [EGK+02, HJK+08]. Étudier l'évolution apporte donc beaucoup d'informations utiles tels que les modules qui ont été le centre d'intérêt principal ou au contraire ceux qui n'ont pas beaucoup évolué [GJR99].

Le domaine de la visualisation de l'évolution du logiciel inclut également la visualisation au niveau du projet, c'est-à-dire la manière dont les développeurs interagissent avec les différentes parties du logiciel [CKN+03, GKSD05, XPM06]. Ce type de visualisation au niveau projet n'est pas traité dans cette thèse, car elle n'est pas en relation avec le logiciel lui-même.

La plupart des techniques de visualisation de l'évolution s'appuient sur des représentations à l'instant $t-2$, $t-1$ ou t et des animations sont utilisées pour faire les transitions d'un instant à l'autre. D'autres techniques plus spécifiques à la visualisation de l'évolution proposent de visualiser l'évolution à l'aide d'une seule vue.

Cette section suit la même structure que la section précédente. Dans un premier temps, nous présentons des techniques de visualisation qui permettent de visualiser l'évolution au niveau de la ligne de code 3.5.1. La section 3.5.2 traite ensuite de l'évolution d'une classe. Enfin, nous nous intéressons à l'évolution de l'architecture complète en section 3.5.3.

3.5.1 Visualisation du changement des lignes du code source

Cette section présente une technique de visualisation qui facilite la compréhension de l'évolution du code source et des différents changements opérés sur le code source (voir détails dans la section 3.5.1 du tableau 3.1 page 17).

Telea, Auber et Chevalier proposent la technique de visualisation appelée *Code flows* qui est une *métaphore de câblage* (Fig. 3.25). Celle-ci permet de visualiser les lignes de code source à travers les différentes versions du logiciel [CAT07, TA08]. Il est ainsi possible de voir un fragment de code spécifique et de suivre son évolution. Cela permet donc de mettre en évidence les évènements tels que le découpage ou la fusion de méthodes (S (split) et M (merge) dans la figure 3.25). La figure 3.25 montre 4 versions d'un code source d'une classe (de la gauche vers la droite) grâce à un placement *Icicle* [BN01]. Les arêtes courbées représentent le parcours d'une ligne de code entre 2 versions de la classe. Pour plus de lisibilité, les lignes du code source qui demeurent identiques sont colorées en noir.

Cette visualisation *Code flows* apporte une vue globale intéressante des différents changements qui s'opèrent au niveau des lignes de code d'une même classe. Elle permet de voir si le code source change de manière significative ou très peu.

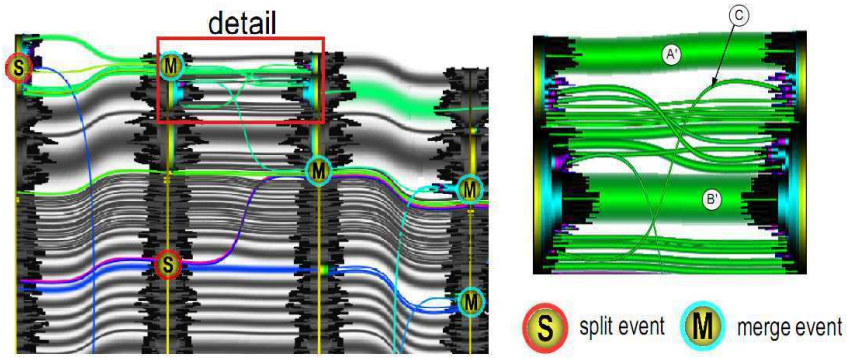


FIGURE 3.25 – Code flow from [TA08] (modified)

3.5.2 Visualisation de l'évolution des classes

Cette section présente une technique de visualisation pour faciliter la compréhension de l'évolution des méthodes d'une classe à travers plusieurs versions d'un logiciel (voir détails dans la section 3.5.2 du tableau 3.1 page 17).

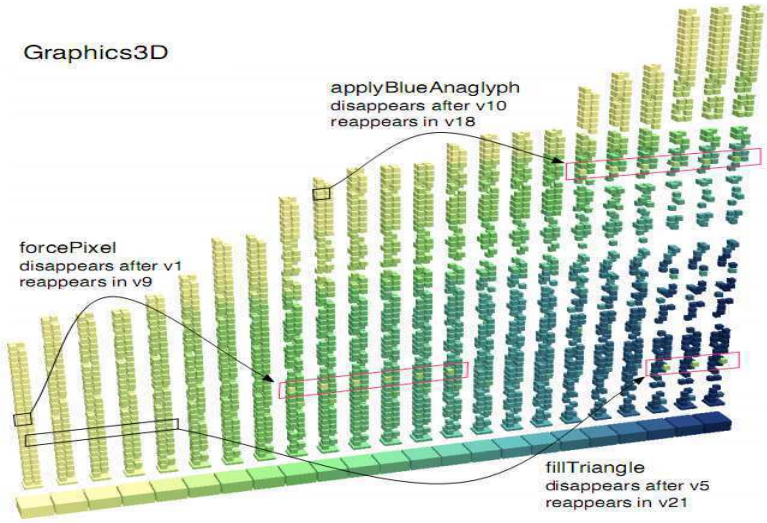


FIGURE 3.26 – Timeline [WL08a] pour visualiser l'évolution de la classe : *Graphics3D* du logiciel *Jmol*

La technique de visualisation *Timeline* par Wettel et Lanza montre l'évolution d'une classe (Fig. 3.26). La technique utilisée repose sur la *métaphore des bâtiments* qui représente les classes comme des bâtiments et les méthodes comme les briques qui le composent. Chaque méthode garde la même place au sein du bâtiment quelle que soit la version du logiciel visualisée : lorsqu'une méthode disparait, la place que la brique occupait dans la visualisation 3D reste vide. La couleur est utilisée pour repérer l'âge des méthodes (de la même manière qu'expliqué en section 3.2), de jaune clair pour les méthodes récentes au bleu foncé pour les anciennes méthodes. La métrique associée à l'âge de la méthode est très simplement calculée en comptant le nombre de versions pour lesquels la méthode est présente.

La visualisation timeline est très efficace pour montrer combien de méthodes sont définies dans les classes et combien sont ajoutées ou supprimées au cours des versions. Il est possible de détecter des motifs ; par exemple, un bâtiment (une classe) qui évolue et perd un nombre de plus en plus important de briques (méthodes) au cours des versions est une classe instable. Un

autre exemple est quand un grand nombre de briques est soudainement ajouté d'une version la suivante, cela signifie que la classe prend plus d'importance au sein du système. Visualiser le timeline de plusieurs classes en même temps permet de détecter les périodes de re-conception du logiciel [FG06].

3.5.3 Visualisation de l'évolution de l'architecture

Visualiser l'évolution de l'architecture du logiciel est probablement le sujet le plus important du domaine de la visualisation de l'évolution du logiciel. Avoir une vue globale de l'évolution du système est crucial pour expliquer l'état actuel de la conception du système. En effet, en visualisant l'historique de conception et re-conception du logociel, il est possible de déduire les choix qui ont menés à l'architecture actuelle.

Dans cette section, nous présentons des techniques de visualisation qui se focalisent sur trois différents aspects de l'évolution du logiciel. D'abord nous verrons comment représenter l'évolution de l'architecture du logiciel et de l'organisation du code source. Puis nous étudierons comment les métriques évoluent à travers les versions du logiciel. Certaines représentations sont plus efficaces que d'autres pour visualiser certains aspects du logiciel. Nous allons donc les décrire en détails et donner les points positifs et les points négatifs que chaque visualisation.

3.5.3.1 Visualisation des changements de l'arborescence

Dans cette section nous présentons une technique de visualisation qui permet de représenter les changements d'organisation du code source (voir section 3.5.3.1 du tableau 3.1 page 17).

Holten et Wijk se sont inspirés de leur précédente visualisation, les Hierarchical Edge Bundles présentée en section 3.5.1, pour créer une nouvelle technique qui permet de comparer les arborescences de deux versions du logiciel [HvW08]. Les arborescences des deux versions du logiciel sont visualisées en même temps. Les arêtes permettent d'identifier comment les deux versions sont apparentées. Pour faciliter la comparaison, les deux arborescences sont organisées de telle manière que les nœuds identiques des deux arborescences soient l'un en face de l'autre (lorsque c'est possible).

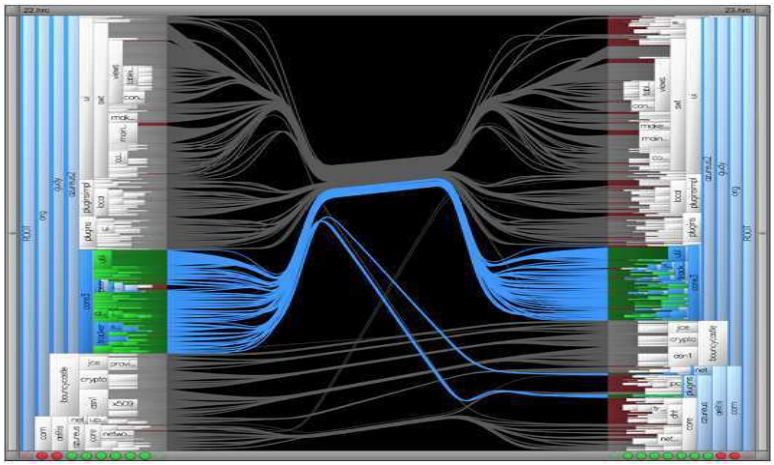


FIGURE 3.27 – Comparaison de l'arborescence de code source [HvW08].

La figure 3.27 présente l'arborescence du code source du logiciel *Azureus v2.2* à gauche (2,283 nœuds), et v2.3 (3,179 nœuds) à droite. Afin de réduire les croisements des arêtes, celles-ci sont courbé de la même manière que pour la technique des Hierarchical Edge Bundles présentée en

section 3.4.2. Afin de permettre une compréhension pus fine des changements de l'arborescence, cette technique permet de sélectionner une ou plusieurs arêtes facilement pour avoir des informations supplémentaires (Fig. 3.27). Les nœuds colorés en rouge dans la partie gauche de la visualisation représentent les éléments supprimés dans la version suivante. Les nœuds colorés en rouge dans la partie droite de la visualisation représentent les éléments qui ont été ajoutés depuis la version précédente.

3.5.3.2 Visualisation de l'évolution des métriques

Utiliser les métriques logicielles permet de quantifier certains aspects du logiciel. Ces métriques sont un moyen de mieux comprendre, contrôler, gérer, prévoir et améliorer le logiciel et son processus de développement.

Dans cette section, nous allons présenter plusieurs techniques montrant l'évolution des métriques logicielles (voir section 3.5.3.2 du tableau 3.1 page 17).

La *matrice d'évolution* est une visualisation simple qui permet d'afficher l'évolution du système grâce à une seule vue (Fig. 3.28) [Lan01, LD02]. Les créateurs, Lanza et Ducasse proposent de représenter les métriques en associant les valeurs aux dimensions de rectangles. Le nombre de méthodes dans la classe est associé à la largeur du rectangle et le nombre d'attributs à la hauteur. Chaque version du logiciel est visualisée en colonne et chaque ligne représente une classe différente.

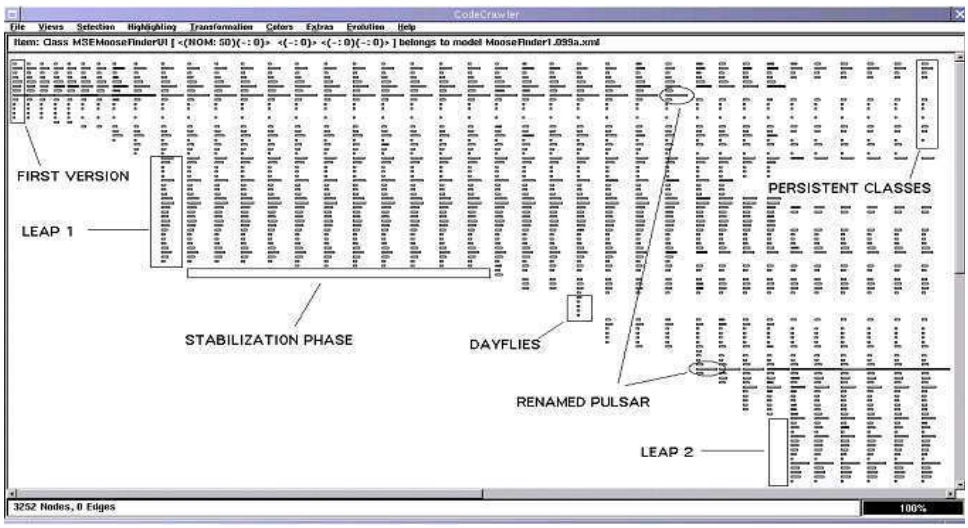


FIGURE 3.28 – Evolution Matrix from [LD02]

Cette technique de visualisation permet surtout de visualiser l'évolution de la taille du système, les ajouts et suppression de méthodes dans les classes et les périodes de croissance du logiciel. D'autres motifs peuvent être visualisés. Par exemple, les classes qui grossissent et décroissent de manière répétitive durant leur durée de vie sont des classes importantes au sein du système. Les classes dont le nombre de lignes de codes augmentent soudainement d'une version à une autre peuvent avoir (eu) un problème de conception.

La matrice d'évolution est donc un moyen simple de visualiser deux valeurs de métriques qui nous renseignent sur l'évolution du logiciel. Ses créateurs considèrent que l'héritage est un aspect important des logiciels à objets, mais la matrice d'évolution ne permet pas actuellement de le visualiser. Ils pensent qu'introduire la visualisation des relations d'héritage entre les classes permettrait d'augmenter l'utilité de leur visualisation. Nous avons vu en section 3.4.3 que pour

pallier ce problème, les auteurs de VERSO ont défini une métrique qui mesure la profondeur d'héritage. Représenter cette métrique grâce à la couleur pourrait être un indicateur supplémentaire pour comprendre l'évolution avec la matrice.

La visualisation *VERSO* par Langelier, Sahraoui et Poulin, présentée en section 3.4.3, permet également de mettre en évidence l'évolution du logiciel [LSP08]. Les figures 3.29(a) et 3.29(b) montrent respectivement la visualisation avec VERSO d'un logiciel en version 1 et version 2. Une interpolation linéaire qui modifie simultanément les trois caractéristiques visuelles (la couleur, la taille et l'angle) est appliquée aux objets de la représentation graphique afin de créer une animation qui permet de passer de la version 1 à la version 2. Cette animation ne dure qu'une seconde, mais elle attire l'attention de l'utilisateur sur les changements et permet une meilleure compréhension qu'en sautant brutalement à la visualisation de la version suivante [SIG07].

Au cours de l'évolution du logiciel, des changements concernant l'arborescence du logiciel peuvent être effectués, modifiant ainsi le placement des classes sur le Treemap [TS07]. Ceci est problématique pour le placement, car avoir une classe en version 1 qui prend la place d'une autre classe en version 2 peut créer de la confusion. Les auteurs ont donc décidé de maintenir la place de chaque classe au sein du Treemap quel que soit le numéro de version du logiciel. De cette manière les classes sont plus facilement repérables. Néanmoins, beaucoup d'espace peut être perdu, car les places de toutes les classes supprimées dans les versions précédentes ou ajoutées dans les versions futures sont gardées vides (Fig. 3.29). Pour pallier ce problème, les auteurs de VERSO proposent simplement de diminuer la taille du Treemap lorsque c'est possible, ce qui permet de réduire la perte d'espace lors des premières version du logiciel (quand seulement peu de classes sont présentes). Cette technique permet donc de garder une place unique pour chaque classe.

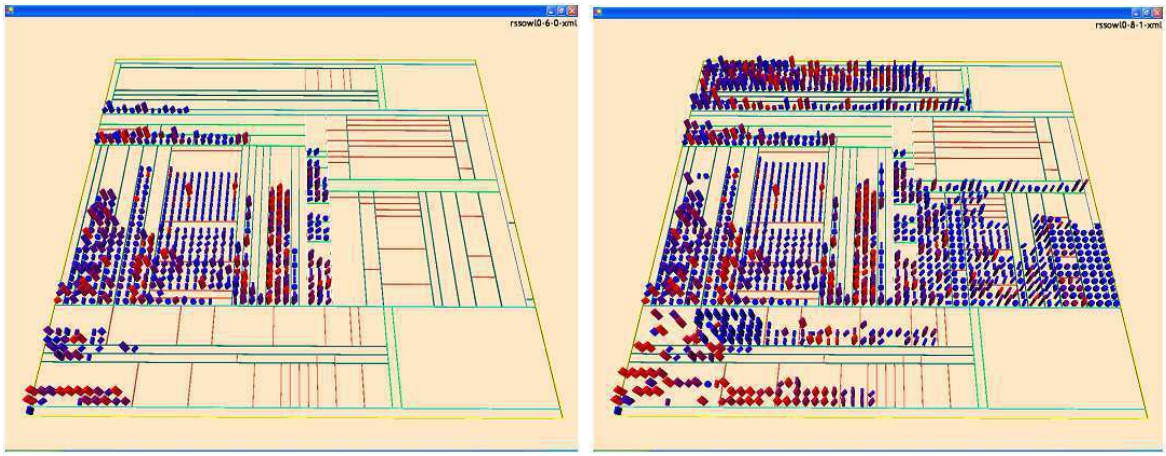


FIGURE 3.29 – Visualisation de l'évolution avec VERSO [LSP08].

Des motifs intéressants peuvent être décelés grâce à la visualisation de 3 différentes métriques qui évoluent pour chaque version du logiciel. Des anomalies de conception telles que le *God Class Anti-Pattern* (classe qui augmente en termes de complexité et couplage) ou le *Shotgun Surgery* (même chose que le premier sauf que ce phénomène apparait et disparait plusieurs fois) peuvent être détectées. De plus la visualisation permet de voir leur évolution et surtout ce qui a mené à leur apparition au sein du système.

L'outil *CodeCity* de Wettel et Lanza, présenté en section 3.4.3, permet de visualiser l'évolu-

tion du logiciel grâce à l'association métrique-représentation définie en section 3.2) et qui permet de voir rapidement l'âge de chacune des classes depuis sa dernière modification. [WL08a]. L'évolution n'est pas vraiment visualisée mais l'âge de chaque composant nous renseigne quand même sur le dernier changement qu'a subi chaque élément du logiciel. La figure 3.30 montre le système au niveau de la méthode.

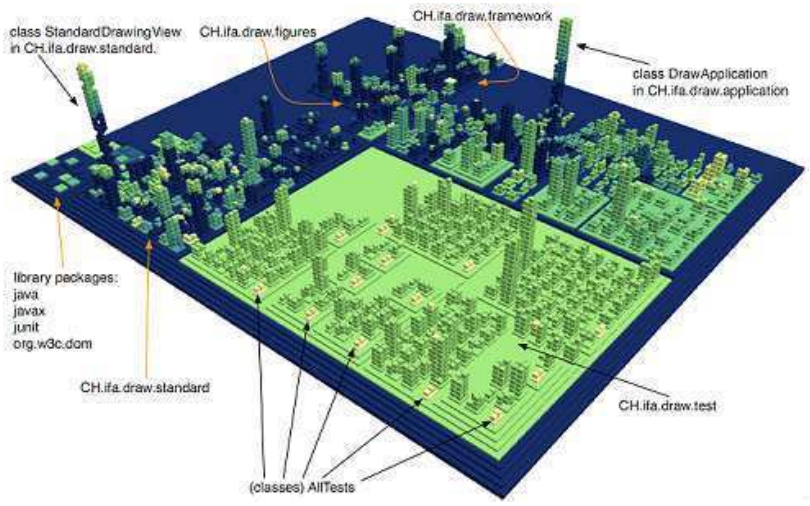


FIGURE 3.30 – Fine-grained Age Map of JHotDraw from [WL08a]

L'inconvénient de cette visualisation est qu'il est impossible de savoir quand une classe ou une méthode particulière a disparu. En effet, l'espace occupé par le bâtiment ou la brique est laissée vide, mais on ne peut pas déterminer à quel moment ceci est arrivé. De plus, les méthodes sont difficiles à discerner lorsque le système est représenté au complet. Pour apporter une solution permettant d'utiliser la métaphore des bâtiments et des briques et visualiser toute l'évolution d'une classe, Wettel et Lanza proposent le *Timeline* (voir section 3.5.2). Celui-ci permet de se focaliser sur l'évolution d'une seule classe et de visionner l'évolution de ses méthodes en une seule image (section 3.5.2).

Pinzger, Gall, Fischer et Lanza proposent la visualisation *RelVis* qui montre l'évolution de plusieurs métriques liées aux modules et aux relations [PGFL05, PGJ05]. Cette technique de visualisation est basée sur les graphes et les *diagrammes Kiviat* pour représenter graphiquement plusieurs valeurs de métrique en les associant chacune à une ligne sur le diagramme Kiviat (Fig. 3.31).

Le graphe est simplement utilisé pour montrer le couplage entre les modules (plus l'arête est large, plus le couplage est grand). La transparence est utilisée pour atténuer l'affichage du graphe pour ne pas surcharger la visualisation.

Les Kiviats permettent de voir l'évolution des métriques. Les valeurs sont placées sur les branches du Kiviat. Les petites valeurs de métriques sont placées proches du centre du Kiviat, les grandes valeurs sont placées loin du centre. RelVis encode le temps en utilisant un dégradé de couleur ; les différentes couleurs indiquent donc le temps entre deux versions consécutives.

Nous remarquons que les métriques logicielles on tendance à augmenter en même temps que le système devient plus complexe et implémente plus de fonctionnalités. Ainsi les bandes de couleurs ne se chevauchent que très rarement, ce qui signifie que le logiciel se développe normalement et qu'aucune reconception du logiciel n'a été entreprise. Lors d'importantes reconception, le logiciel subit de grosses modifications qui peuvent faire baisser les valeurs de certaines métriques. Ainsi les répercussions sur la RelVis visualisation se manifeste par des zones de couleur entièrement

recouvertes par la couleur de la version après reconception (Fig. 3.31(a)).

Une caractéristique intéressante de RelVis est qu'il peut mettre en évidence l'évolution des métriques des modules (Fig. 3.31(a)) ainsi que l'évolution des métriques des relations (Fig. 3.31(b)). La figure 3.31(a) montre 3 diagrammes Kiviats représentant 3 modules de Mozilla. Chaque diagramme montre 20 métriques à travers 7 différentes versions numérotée de 1 à 7 [PGFL05], soit un total de 140 métriques par diagramme Kiviat.

Les métriques qui caractérisent les mêmes propriétés du logiciel (e.g. les métriques de taille) sont regroupées ensemble sur les branches de l'étoile du Kiviat. La figure 3.31(a) met en évidence le module *DOM*, qui est le plus grand de tous les modules de Mozilla. De plus, on voit que ce module grossit de plus en plus à chaque version, la plus grosse évolution étant de la version 2 à la version 3 (cyan).

On remarque aussi que le nombre de fonctions a dramatiquement augmenté de entre les versions 1 et 2 (bleu) mais décroit à la version suivante (cyan). De même, le nombre d'appels de méthode entrants augmente constamment de la version 1 (bleu) à la version 7 (rouge) avec un pic intéressant entre les versions 5 et 6 (orange). Le nombre d'appels de méthode sortants augmente constamment jusqu'à la version 6 (orange), en revanche il décroit beaucoup de la version 6 à la version 7 (rouge). Apparemment les programmeurs ont fait une re-conception d'une partie du logiciel qui a permis de réduire le couplage du module DOM en diminuant le nombre d'appels de méthodes sortant.

La figure 3.31(b) présente des métriques en rapport avec les relations du logiciel. Les diagrammes sont donc placés sur les arêtes du graphe. Chaque moitié de diagramme de Kiviat correspond aux relations avec le module qui lui est le plus proche. La figure 3.31(b) montre quelques points intéressants. Par exemple, les relations entre le module en bas à droite et le module en bas au milieu diminuent de la version 6 à la 7 (rouge) ; on peut donc conclure que les développeurs ont découplés ces deux modules dans la version 7.

On remarque que cette technique de visualisation est capable de montrer énormément d'informations. Elle permet de déceler les moments importants de l'évolution du logiciel grâce à une seule image, notamment au niveau du nombre de métriques visualisées. Le réel avantage de cette visualisation est que toute l'évolution est montrée grâce à une seule et même image, contrairement à l'utilisation d'animations qui montre successivement les valeurs pour chaque version du logiciel.

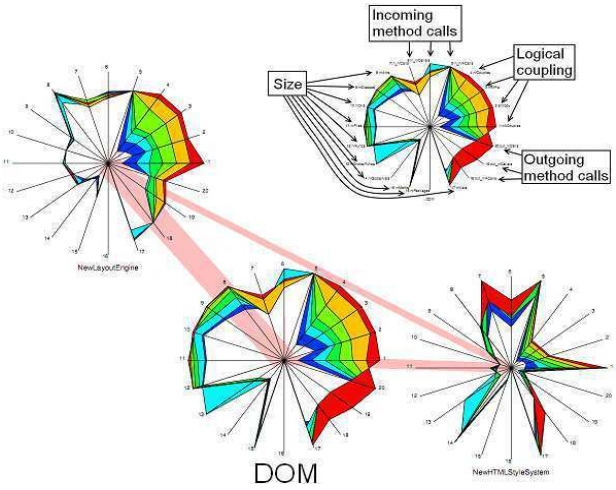
Le désavantage de cette approche est que lorsqu'une métrique baisse certaines bandes de couleur des versions précédentes peuvent être recouvertes. Ceci a pour conséquence de cacher certaines valeurs de métriques pour les versions précédentes.

Les travaux présentés dans [KJ09] implémentent des diagrammes de Kiviat en 3D pour afficher chaque version du logiciel à un niveau d'élévation différent, résolvant ainsi le problème de recouvrement.

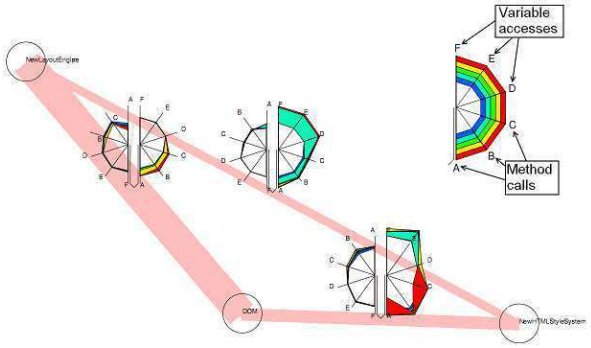
3.6 Conclusion

Dans ce chapitre, nous apportons un état de l'art complet et à jour de ce qui se fait au niveau de la visualisation de l'aspect statique du logiciel et son évolution. Le grand nombre de conférences et workshops dans ce domaine montre bien qu'il est très actif et qu'il existe un réel engouement de la part de la recherche mais également de l'industrie concernant les techniques et outils de visualisation des logiciels [BPK03].

Dans le tableau 3.1 page 17, nous avons résumé les techniques de visualisation et nous les



(a) Métriques sur les modules



(b) Métriques sur les relations

FIGURE 3.31 – RelVis [PGFL05] (modifié)

avons catégorisées de manière à aider le lecteur à s'y retrouver et à parcourir ce chapitre, qu'il vienne du milieu académique ou non.

Comme indiqué dans ce chapitre, le domaine de la visualisation du logiciel a beaucoup évolué ces dix dernières années (voir la colonne *Année* du tableau 3.1 page 17), ce qui a permis d'apporter aux développeurs de nouveaux outils pour analyser, comprendre et développer les logiciels.

Même si de nouveaux produits de visualisation ont atteint le niveau d'un produit commercial comme *SolidFX*, qui est un outil permettant la visualisation de programmes C et C++ pour aider la re-conception, la visualisation du logiciel n'est pas encore largement répandue dans l'industrie.

Pourtant ces outils permettent aux chefs de projet de mieux percevoir la qualité et l'avancement du projet et donc de mieux gérer le projet, en faisant des choix plus cohérents et constructifs concernant la re-conception de certaines parties. En effet, ces outils apportent un moyen fiable de visualiser l'état du système et donc de mieux comprendre son code.

Souvent les prototypes de visualisations de recherche ont un problème de passage à l'échelle lorsqu'il s'agit de conquérir de monde industriel. Des recherches substantielles sont souvent nécessaires pour transformer les prototypes de recherche en produits commerciaux. En effets les prototypes de visualisations utilisent très souvent les produits open-sources disponibles sur internet pour tester leur technique de visualisation. Alors qu'en pratique, les produits commerciaux peuvent être bien plus complexe et donc bien plus difficile à visualiser.

Ces dernières années, une tendance se dégage (au niveau de la recherche) pour les techniques de visualisation en 3D. Même si la navigation en 3D reste difficile, des résultats prometteurs ont été obtenus avec l'utilisation de ces visualisations 3D pour faciliter la compréhension des logiciels. Notamment au niveau de l'utilisation des métaphores du monde réel pour faciliter l'orientation et donc la navigation. Avec l'apparition de nouvelles technologies permettant de faciliter la navigation ou l'interaction avec la visualisation (telles que les écrans très larges, la stéréo vision, les techniques de suivi de l'œil ou les détecteurs de mouvement, etc.), les possibilités techniques de base ont nettement progressé, facilitant du coup la réalisation d'outils avancés de visualisation de logiciels. Il y a donc de grandes opportunités de recherche, et d'industrialisation, dans ce domaine.

4

Analyses et métriques dynamiques du logiciel

Sommaire

4.1 Introduction

L'avantage des métriques dynamiques par rapport aux métriques statiques est l'absence de la prise en compte du code mort dans les mesures. En effet, l'analyse statique mesure tous les cas d'exécution "possibles", la notion de possible dépendant de (la précision de) l'analyse statique. Par conséquent, les résultats des métriques sont imprécis parce qu'ils sont basés sur un sur-ensemble des cas qui arrivent réellement à l'exécution.

En revanche, un des désavantages à utiliser l'analyse dynamique est le choix des jeux de tests pour les mesures. En effet, les métriques calculées dynamiquement peuvent dépendre (parfois fortement) des données en entrée du programme. De plus, la couverture de la totalité du code (non mort) du programme n'est pas toujours facile avec les jeux de tests disponibles. Pour cette raison, il est souvent très judicieux d'effectuer un test de couverture, afin de fournir les informations suffisantes pour pouvoir interpréter convenablement les résultats (métriques) obtenus.

Dans la section 4.2 nous donnons les définitions et exemples des différents types de métriques dynamiques (i.e. d'exécution) du logiciel. Cette section est divisée en plusieurs sous-sections décrivant chacune un type de métrique particulier. Nous discutons ainsi des métriques dynamiques de taille d'exécution (section 4.2.1), de complexité (section 4.2.2), d'utilisation de structure de données et de types primitifs (section 4.2.3), de couplage (section 4.2.4) et polymorphisme (section 4.2.5) et liées aux fils d'exécutions (section 4.2.6). Puis nous concluons en section 4.3.

4.2 Les types de métriques d'exécution du logiciel

Dans cette section nous décrivons la plupart des métriques dynamiques de la littérature. De la même manière que pour l'aspect statique, nous cherchons à déterminer les aspects essentiels de l'exécution tels que : la taille de l'exécution, la complexité, le couplage, le polymorphisme, les fils d'exécution, etc. Même si tout ces critères dynamiques peuvent être évalués statiquement, le seul moyen d'avoir des mesures dynamiques précises et réelles est d'analyser l'exécution du programme. Nous définissons tous ces aspects dans ce chapitre.

Dufour, Driesen, Hendren et Verbrugge ont notamment proposés un cinquantaine de métriques dynamiques pour Java dans leur article [DDHV03]. L'intérêt de ces métriques est qu'elles présentent certaines caractéristiques désirables telles que la non-ambiguïté, la robustesse, la discrimination, l'aspect dynamique et l'indépendance par rapport à la machine qui exécute le programme. Ces caractéristiques sont utiles pour permettre la comparaison entre différentes études de l'exécution des programmes :

- La caractéristique de non-ambiguïté recouvre l'utilisabilité de la métrique, en capturant un aspect précis de l'exécution.
- La robustesse garantit qu'un petit changement au niveau du scénario d'exécution engendrera une petite variation de la métrique d'exécution. En effet, l'exécution d'un programme dépend des données en entrée, et celles-ci influencent parfois grandement l'exécution du programme. Une métrique robuste est une métrique qui varie proportionnellement aux données d'entrée.
- Sur le même principe, la discrimination assure que de grandes différences au niveau des données d'entrée engendrent de grandes variations au niveau des métriques.
- Ces métriques dynamiques doivent capturer un aspect dynamique du programme qui serait incalculable précisément de manière statique.
- Enfin, ces métriques doivent être indépendantes de la machine.

Dans cette thèse, nous nous focaliserons principalement sur ces métriques car nous pensons qu'elles fournissent la base la plus précise pour comprendre l'exécution du logiciel. Par conséquent, nous avons implanté le calcul de la plupart de ces métriques dans notre analyseur *VITRAIL JBInsTrace Analyzer* qui sera présenté dans le chapitre 6 section 6.4 page 85.

Une question importante que se posent la plupart des chercheurs du domaine de l'analyse de l'exécution du logiciel est la prise en compte ou non des bibliothèques utilisées par le programme. En effet, prendre en compte ces bibliothèques engendre une grande quantité d'information supplémentaires à collecter et interpréter. Nous pensons que l'analyse des bibliothèques a longtemps été ignorée simplement parce qu'elle est très lourde et pose de nombreux problèmes techniques. Néanmoins, nous souhaitions vivement disposer des informations d'exécution de ces bibliothèques pour nos analyses pour avoir un niveau de détail très fin. Nous avons donc spécialement conçu notre outil VITRAIL JBInsTrace Tracer pour récupérer les informations sur l'exécution du programme *et* des bibliothèques utilisées.

Pour permettre une analyse plus fine, les métriques calculées sont séparées en deux groupes : celles spécifiques aux classes du programme, et celles calculées sur les bibliothèques. De grande différences peuvent être obtenues si cette distinction n'est pas prise en compte. Par exemple, le simple programme Java "Hello World" exécute environ 7800 bytecodes mais seulement 4 bytecodes concernent la classe du programme "Hello World" [DDHV03].

Dans les sous-sections 4.2.1 et 4.2.2 nous décrivons d'abord respectivement les métriques de taille puis les métriques de complexité de l'exécution. Dans la sous-section 4.2.3 nous définissons brièvement les métriques qui s'intéressent à mesurer l'intensité d'utilisation des structures de

données. La sous-section 4.2.4 aborde les métriques de couplage à l'exécution. Nous y verrons que certains travaux s'intéressent particulièrement aux métriques de couplage dynamique. Dans la sous-section 4.2.5 nous nous intéressons à définir les métriques liées au polymorphisme, qui est un aspect essentiel des programmes à objets. Enfin la sous-section 4.2.6 traite des métriques en relation avec les différents fils d'exécution (*threads*).

4.2.1 Métriques de taille

De la même manière que les métriques de taille statiques, les métriques de taille dynamiques sont les plus simples à définir, comprendre et calculer. Les métriques de taille permettent d'avoir une idée générale de la taille du programme [DDHV03] en calculant le nombre de classes chargées, le nombre de bytecodes chargés, etc. D'un point de vue plus dynamique, les métriques de taille calculent également le nombre de bytecodes touchés par l'exécution, le nombre total de bytecode exécutés, etc.

Les métriques de taille permettent donc de déterminer l'ordre de grandeur du logiciel et également de cibler les classes qui jouent un rôle prédominant au niveau de l'exécution du programme.

A notre connaissance, très peu d'articles traitent des métriques de taille dynamique, contrairement au métriques de taille statique qui ont fait l'objet de beaucoup de publications [AGJ83, CS00, Fen91].

D'un point de vue de l'ingénierie logiciel il est très compliqué de lier le nombre d'instructions machines exécutées avec une propriété du codage. En effet, l'analyse de l'exécution doit nous permettre de déceler des anomalies de codage ou plus simplement nous permettre de comprendre le fonctionnement du logiciel et donc du code. En pratique, les métriques simples telles que le nombre de classes chargées ou le nombre d'appels de méthodes total, etc. renseignent bien plus sur la manière dont le logiciel est codé plutôt que le nombre d'instructions machines exécutées.

4.2.2 Métriques de complexité

Ces métriques mesurent le nombre de bytecodes de contrôle et le nombre de sauts (jumps) exécutés au niveau du bytecode [DDHV03]. Les appels de méthodes sont eux aussi considérés comme des sauts dans le bytecode car ce sont littéralement des sauts vers le bytecode d'une autre méthode.

La métrique la plus évidente consiste à compter le nombre de bytecodes de contrôle touchés lors de l'exécution. Grâce à celle-ci, la densité de bytecode de contrôle touché par rapport au nombre total de bytecode touché permet d'avoir la moyenne de la complexité (du flot) de l'exécution. Nous pensons que cette dernière mesure reflète bien la complexité de l'exécution. Pour estimer plus précisément la complexité dynamique d'un programme, au lieu de compter tous les bytecodes de contrôle rencontrés, nous mesurons le nombre de bytecodes de contrôle qui ont effectivement modifié le flux d'exécution du programme avec un saut. La densité de cette dernière mesure par rapport au nombre total de bytecodes permet d'avoir une moyenne du nombre de sauts par bytecode.

Les métriques de complexité de l'exécution (en terme du nombre de sauts d'instructions) permettent donc de localiser les classes complexes à l'exécution.

4.2.3 Métriques d'utilisation des structures de données et types primitifs

Les programmes à objets utilisent les structures de données et les types primitifs de manière intensive [DDHV03]. Par conséquent, calculer le taux d'utilisation dynamique des structures de données est une information intéressante, car elle peut par exemple aider à cibler les éventuelles baisses de performance. L'idée principale des ces métriques est de déterminer si le programme exécute beaucoup d'opérations sur les tableaux et les références d'objet ou au contraire sur les types primitifs tels que les nombres flottants, les autres types primitifs. Pour cela il faut compter les bytecodes exécutés qui manipulent ces types de données.

4.2.4 Métriques de couplage

Les métriques dynamiques de couplage sont probablement les métriques dynamiques les plus étudiées. Les langages à objets sont basés sur les principes d'encapsulation, de création d'objet et d'envoi de message. Ainsi, les accès aux attributs et les appels de méthodes sont des éléments essentiels qui peuvent être quantifiés par les métriques de couplage.

L'article d'Arisholm, Briand and Føyen [ABF04] définit rigoureusement 12 métriques de couplage dynamique que nous allons détailler ci-dessous. Les métriques de couplage peuvent être calculées à différents niveaux de granularité et de différents points de vue. Ainsi les mesures peuvent être faites au niveau des objets ou de la classe parce que les mesures sont faites lors de l'exécution du programme. De plus, les mesures au niveau des objets peuvent être agrégés au niveau de la classe instanciée. Concernant les points de vue, on peut différencier l'appelant de l'appelé.

Arisholm, Briand and Føyen proposent donc les 3 métriques suivantes, qui chacune ont deux niveaux de granularité (objet, classe) et deux points de vue (appelant, appelé) :

- le nombre total de messages envoyés (ou reçus) au niveau de chaque objet (ou classe).
- le nombre de méthodes différentes qui invoquent (qui sont invoquées) par d'autres méthodes au niveau de chaque objet (ou classe).
- le nombre de classes différentes qui invoquent (qui sont invoquées) par d'autres classes au niveau de chaque objet (ou classe).

Nous constatons que la plupart des articles de la littérature du domaine définissent plus ou moins ces mêmes métriques [YAR99, Ari02].

4.2.5 Les métriques liées au polymorphisme

Le polymorphisme est une caractéristique importante des langages à objets. Celle-ci utilise l'héritage pour permettre la résolution dynamique des receveurs sur les sites d'appels des méthodes virtuelles (ou polymorphes). Cette résolution dynamique peut être estimée statiquement mais une exécution du programme est nécessaire pour capturer précisément les informations de résolution réelles.

L'héritage étant une propriété très importante des langages à objets, mesurer le taux de polymorphisme permet d'étudier le comportement des sites d'appels virtuels. Ainsi il est possible de déterminer si le code fait une bonne utilisation de l'héritage dans les langages à objets. Dans leurs travaux, Dufour, Driesen, Hendren et Verbrugge proposent des métriques mesurant le taux de polymorphisme d'un programme Java [DDHV03].

De la même manière que le couplage, le polymorphisme est bidirectionnel. Des métriques s'intéressent aux types des différents receveurs sur les sites d'appels, tandis que d'autres mé-

triques s'intéressent aux méthodes ciblées par les sites d'appels. Ainsi chaque métrique à deux versions :

- le nombre de sites d'appels qui ont 1, 2 ou plus de 2 receveurs (respectivement, méthodes cibles) différent(e)s.
- le nombre d'appels qui ont lieu sur des sites d'appels avec 1, 2 ou plus de 2 receveurs (respectivement, méthodes cibles) différent(e)s.
- le nombre de changements de types de receveurs sur les sites d'appels ou le nombre de changements de méthodes appelées.

Ces métriques permettent de :

- savoir si les sites d'appels sont plutôt monomorphes ou polymorphes.
- connaitre la quantité d'appels polymorphiques.
- estimer le taux d'erreur d'appel de méthode en cas de dé-virtualisation des sites d'appels. En effet est-il possible de fixer la méthode ciblée par un site d'appel ou est-ce que la méthode ciblée varie constamment ?

Une autre métrique intéressante même si elle ne mesure pas le polymorphisme à proprement parler est celle qui compte le nombre de sites d'appels virtuels. Elle permet d'estimer la quantité d'efforts que fait la JVM pour les résolutions dynamiques. En effet, le choix d'une méthode parmi toutes celles qui ont le même nom s'effectue sur la base des informations de typage. L'effort qui est demandé à la JVM consiste à retrouver les méthodes cibles à partir du type des objets et de l'arbre d'héritage.

Ces métriques liées au polymorphisme nous intéressent particulièrement et nous avons des travaux en cours traitant de ce sujet (voir la partie III section 8.2).

4.2.6 Les métriques liées aux fils d'exécutions

Le dernier aspect de l'exécution que nous allons traiter ici concerne les fils d'exécution. En effet, les fils d'exécution sont intéressants car ils offrent des fonctionnalités de concurrence et synchronisation, extrêmement utiles dans de nombreux programmes, améliorant ainsi les performances lorsqu'elles sont bien utilisées et les dégradant dans le cas contraire.

Deux aspects essentiels liés aux fils d'exécutions peuvent être clairement identifiés : le nombre de fils d'exécution concurrents et la synchronisation entre ces fils. La métrique calculant le nombre maximum de fils d'exécution actifs permet d'estimer grossièrement le taux de concurrence d'un programme. Par contre, mesurer la quantité de code exécuté alors qu'un autre fil d'exécution est actif donne plus de précision sur la concurrence réelle entre les fils d'exécution.

Pour mesurer la synchronisation, nous nous intéressons au nombre d'opérations de blocage (*lock*). En Java, il existe deux bytecodes spécifiquement réservés pour cette opération de blocage (*monitorenter, monitorexit*) donc le comptage est relativement simple. Une métrique plus intéressante et plus dynamique est le nombre de fois qu'un fil d'exécution s'est fait mettre en attente à cause d'un autre fil détenant l'autorisation d'exécuter ce code.

Nous pouvons souligner que le suivi des fils d'exécution n'est pas trivial parce que l'analyseur s'exécute en même temps que le programme et risque donc d'influencer l'ordonnancement du programme analysé (voir chapitre 6, section 6.3.3, page 83).

4.3 Conclusion

Les métriques dynamiques du logiciel reflètent "réellement" ce qui arrive lors de l'exécution du logiciel. Ainsi elles sont le seul moyen d'obtenir des résultats précis sur l'exécution. Si nous

comparons le domaine des métriques dynamiques et des métriques statiques, nous constatons qu'il existe moins de métriques dynamiques que de métriques statiques 2.

Nous constatons également qu'aucun travail ne regroupe des métriques dynamiques qui capturent différents aspects de l'exécution pour ensuite estimer la qualité du logiciel (de la même manière qu'avec les métriques statiques chapitre 2 section 2.3.1). Les résultats avec les métriques statiques étant positifs, un travail de recherche intéressant serait d'explorer cette option de créer un ensemble de métriques dynamiques permettant la détection d'anomalies d'exécution et donc de codage.

Notons les travaux [ABF04] où les métriques de couplage dynamique sont utilisées pour estimer avec succès le changement dans les logiciels. En effet, ces travaux ont montré une corrélation entre métriques de couplage dynamique et les changements qui interviennent entre deux versions du logiciel.

Il semblerait que l'attention des chercheurs du domaine de l'analyse dynamique des logiciels se porte particulièrement sur les métriques de couplage dynamique [Ari02, ABF04]. En effet, les connexions entre les objets à l'exécution nous renseignent beaucoup sur l'architecture des classes et donc sur la conception du logiciel, ce qui est le sujet le plus intéressant d'un point de vue ingénierie logicielle.

5

Visualisation de l'aspect dynamique du logiciel

Sommaire

5.1 Introduction

Le domaine de la visualisation de l'exécution des logiciels est moins étudié que celui de la visualisation de l'aspect statique des logiciels (voir chapitre 3). Nous pensons que ceci est dû au fait que les outils permettant d'analyser les données d'exécution sont moins répandus et moins aboutis que les outils permettant les analyses statiques du code source [Moc03]. Par conséquent, si les données d'analyse ne sont pas disponibles, peu de visualisations s'intéressent à visualiser l'exécution des logiciels. Néanmoins il existe un certain nombre de travaux et publications dans le domaine de la visualisation de l'exécution dont nous traiterons dans ce chapitre.

Certaines visualisations sont des adaptations des visualisations qui traitent de l'aspect statique des logiciels. Ce chapitre est organisé de la manière suivante : La section 5.2 présente les visualisations de traces d'exécution du logiciel à trois niveaux de granularités : au niveau de la ligne de code (section 5.2.1), au niveau de la méthode (setion 5.2.2), au niveau de la classe (section 5.2.3) et au niveau de l'instance (section 5.2.4). Puis la section 5.3 traite des visualisations qui utilisent les métriques d'exécution. Enfin, nous ferons une conclusion sur ce chapitre dans la section 5.4.

5.2 Visualisation de trace d'exécution

Dans cette section nous présentons des visualisations de l'exécution des logiciels qui se focalisent sur la visualisation des traces d'exécution. De la même manière que dans le chapitre 3,

nous garderons la même structuration en découpant chaque sous-section pour qu'elle présente un niveau de granularité différent. La section 5.2.1 présente d'abord une visualisation qui se focalise sur la visualisation de l'exécution au niveau de la ligne de code. Puis, dans la section 5.2.2 nous décrivons une visualisation qui se concentre sur la méthode. Dans la section 5.2.3 nous étudions ensuite une visualisation qui se place au niveau de la classe pour représenter des informations liées à l'exécution du logiciel. Enfin la section 5.2.4 traite des visualisations au niveau des instances crées à l'exécution.

5.2.1 Visualisation orientée ligne de code

Les visualisations de traces d'exécution ayant pour niveau de granularité la ligne de code sont rares, car le très grand nombre de lignes parcourues par l'exécution d'un logiciel est difficilement représentable. Néanmoins, les auteurs de [OJH03] proposent une visualisation appelée *Execution Bar*. La première étape de cette visualisation consiste à colorer le code source. La figure 5.1(a) présente un exemple de coloration de code source avec les appels de méthode en vert et les conditionnelles en rouge. La deuxième étape consiste à visualiser l'exécution grâce à une barre composée de plusieurs bandes colorées représentant chacune une ligne dans le code source du logiciel 5.1(b). La visualisation de l'exécution consiste donc à voir défiler l'exécution de ces lignes grâce à une animation ou via une touche qui permet le défilement.

Un exemple d'utilisation de cette visualisation en pratique est l'observation des levées d'exception en utilisant la métaphore des feux de signalisation du trafic routier, la couleur verte signifiant l'absence de risque, l'orange signifiant attirant l'attention, et le rouge signifiant un danger. Lors de la génération de la trace d'exécution du logiciel, toutes les informations concernant le déclenchement des exceptions sont enregistrées. Puis, lors de l'analyse et la visualisation, les lignes n'ayant levé aucune exception sont colorées en vert, celles qui en ont levé occasionnellement en orange et celles qui en ont levé beaucoup en rouge.

Un autre exemple de coloration utilisant le même principe consiste à calculer le nombre de fois qu'une même ligne est exécutée puis de visualiser grâce à l'Execution Bar les lignes qui sont les plus exécutées. Dans ce dernier cas, l'important est de visualiser si ces lignes sont concentrées autour d'un ou plusieurs "points chauds" et d'évaluer leur nombre. Ceci permet de repérer des points de congestion à l'exécution.

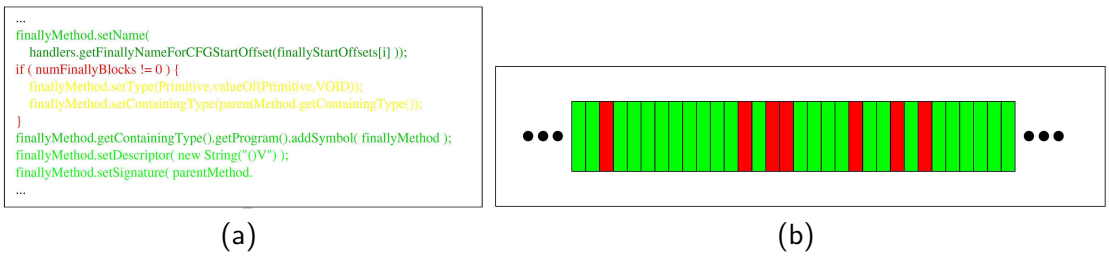


FIGURE 5.1 – (a) Code source coloré, (b) Execution Bar provenant de [OJH03].

Cette technique de visualisation est exploitable en combinaison avec d'autres représentations qui montrent déjà l'ensemble du système. Cette combinaison permet donc de visualiser l'impact de l'exécution des lignes de codes sur le système entier [OJH03].

JOVE [RR05] est également un outil d'analyse et visualisation qui permet de voir des données d'exécution en temps réel au niveau de la ligne de code. Le but principal de JOVE est d'aider à comprendre les problèmes de performance et des comportements anormaux du programme

lorsqu'ils surviennent, à l'exécution. Néanmoins, suivre tout ce qu'il se passe à l'exécution en temps réel n'est pas facile car le code s'exécute tellement rapidement qu'il est impossible de suivre l'exécution de chaque ligne de manière efficace. Pour cette raison, la visualisation se rafraichit toutes les 10 millisecondes ou plus (suivant la configuration de l'utilisateur). La représentation graphique est en fait une variation de SeeSoft (voir chapitre 3 section 3.2), car l'espace est découpé verticalement en régions qui représentent chacune un fichier source exécutés lors du scénario d'exécution (fig. 5.2). La largeur de chaque région correspond au temps d'exécution de chaque fichier. Le cercle en haut de la visualisation représente le temps total passé à exécuter le fichier. Ce cercle est découpé en quartiers qui montrent le temps d'exécution des différents fils d'exécution (une couleur par fil).

Une des particularités de JOVE est qu'il analyse l'exécution au niveau du bloc de base, permettant ainsi une analyse fine. Au niveau de la visualisation, l'exécution des blocs de base se reflète par des rectangles colorés dans la partie inférieure de chaque région. Chaque bloc de base est représenté sur une ligne dont la hauteur représente le nombre d'instructions dans le bloc de base. La largeur quant à elle correspond au nombre de fois que le bloc de base est exécuté.

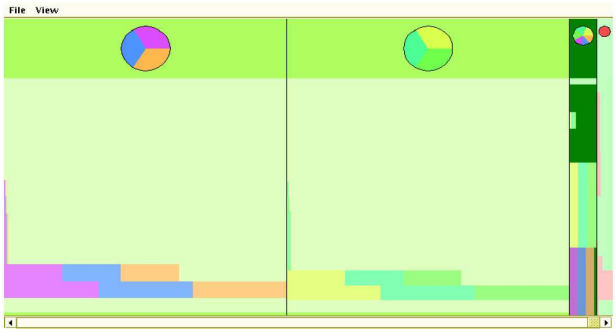


FIGURE 5.2 – Visualisation de l'exécution d'un programme Java avec JOVE [RR05].

JOVE permet donc de visualiser des données d'exécution au niveau bloc de base en temps réel, ce qui offre à l'utilisateur une vue d'ensemble qui facilite l'identification des problèmes. Néanmoins, les auteurs [RR05] précisent que cette visualisation permet de se focaliser sur certaines classes en particulier, mais pas sur le système entier. En effet, la visualisation occupant beaucoup d'espace horizontalement d'une part et le système d'analyse en temps réel ne permettent pas une mise à l'échelle au niveau de toutes les classes du système d'autre part.

5.2.2 Visualisation orientée méthode

Visualiser les appels de méthodes, sans considérer le flot d'exécution intra-méthode, permet de se focaliser sur les envois de message qui sont une caractéristique essentielle des langages à objets. Bien évidement, l'agrégation au niveau des classes et des paquets est possible.

Dans [MBAV09] les auteurs présentent une technique pour visualiser une trace d'exécution à l'aide d'un *Calling Context Tree*[1] *(CCT)*. La représentation du CCT se fait grâce à une représentation d'arbre dont les nœuds sont composés d'un identifiant de méthode et d'un ensemble de métriques 5.3(a). Le parent de chaque nœud correspond à l'appelant de ce nœud dans ce contexte précis. Les fils de chaque nœud correspondent aux méthodes appelées par ce nœud. Les métriques de chaque parent correspondent logiquement à ses métriques, plus l'agrégation des métriques de tous ses enfants. Nous avons vu dans le paragraphe 3.3 qu'un arbre pouvait être représenté de plusieurs manières. Dans leurs articles [MBAV09], les auteurs utilisent une

1. est une structure de données représentant toutes les informations d'exécution de chaque fil du programme.

représentation radiale (en anneau) pour représenter le CCT 5.3(b). Les métriques de chaque noeud servent à déterminer la taille du segment de l'anneau. Les parties représentées dans le prolongement d'une portion d'un anneau correspondent aux contributions des fils de cet anneau. Par exemple, la figure 5.3(b) utilise la métrique *Nombre de Bytecodes Exécutés*; la partie droite de l'anneau central (`main`) n'a pas d'anneau dans son prolongement, elle montre donc que 33% des bytecodes sont exécutés par la méthode `main` elle-même.

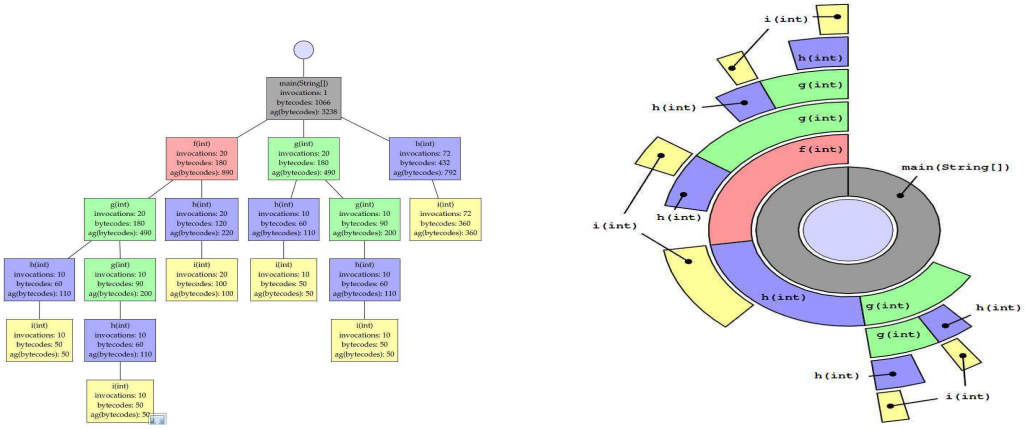


FIGURE 5.3 – (a) Calling Contex Tree, (b) Calling Context Ring Charts [MBAV09].

Le problème principal quand on veut visualiser l'exécution est la très grande taille de la trace. Trouver les informations pertinentes pour répondre à certaines tâches précises est donc, très difficile. *Jinsight EX* [SDPK01] (figure 5.4) est une visualisation de trace d'exécution au niveau de la méthode, qui permet de se focaliser sur le comportement de certaines méthodes lors de l'exécution du programme. Le temps d'exécution est représenté de haut en bas et les appels de méthode sont représentés par des rectangles placés de gauche (méthodes appelantes) vers la droite (méthodes appelées). L'utilisateur définit plusieurs groupes de méthodes qui permettent l'identification de certaines fonctionnalités du programme. Par exemple, la figure 5.4 visualise deux fils d'exécution qui exécutent des opérations sur une base de données. D'abord, les méthodes qui servent à établir la connexion avec la base de données forment le premier groupe et sont colorée en bleue. Puis, les méthodes exécutant les requêtes forment le deuxième groupe qui sont colorée en vert. Enfin, les méthodes qui font des traitements sur l'ensemble des résultats d'une requête forment le troisième groupe et sont colorée en rose.

Le résultat est donc un condensé du graphe d'appel des méthodes avec la couleur permettant de faire ressortir certaines méthodes qui réalisent certaines fonctionnalités bien précises. Cette visualisation ne donne pas de renseignement précis sur le flot d'exécution mais montre comment les différents fils d'exécution se répartissent la réalisation de certaines tâches.

5.2.3 Visualisation orientée classe

La notion de classe est essentielle en programmation à objets : elle définit une abstraction, un type abstrait qui permet d'instancier les objets. Représenter le système en visualisant des informations agrégées au niveau des classes présente un intérêt double pour la représentation graphique : ce niveau de granularité permet rapidement de faire le rapprochement entre instances et classes d'une part et les problèmes de mise à l'échelle sont moindres au niveau de la classe

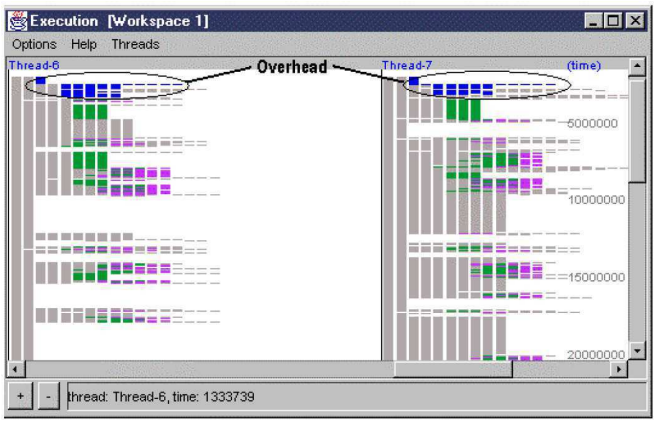


FIGURE 5.4 – Détails d'exécution de deux fils d'exécution avec les couleurs représentant des groupes de méthode[GLW05].

comparativement avec le niveau plus fin des méthodes.

Les auteurs de [HCvW07a, CHZ+07, CZH+08] ont adapté la technique de visualisation des *Hierarchical Edge Bundles (HEBs)* décrite dans le chapitre 3 paragraphe 3.4.2 pour visualiser des traces d'exécution au niveau de la classe. Leur technique de visualisation se divise en 2 parties :

- La vue : *Massive Sequence* 5.5(a) représente l'arborescence du logiciel grâce à un placement *Icicle* (partie haute) et les relations entre les différents éléments grâce à un trait (partie basse). Le sens de la relation est représenté par la couleur, du vert vers le rouge. Les relations sont ordonnées par ordre chronologique, du haut vers le bas. Dans la figure 5.5(a) nous visualisons 7 relations, mais la taille de la fenêtre est variable et permet de visualiser un très grand nombre de relations.
- La vue : Hierarchical Edge Bundles 5.5(b) représente les mêmes relations que celles affichées dans la vue Massive Sequence, mais en utilisant la technique éponyme décrite dans le paragraphe 3.4.2. Avec cette visualisation il est possible de modifier la signification de la couleur pour visualiser l'ordre chronologique des relations ou la direction.

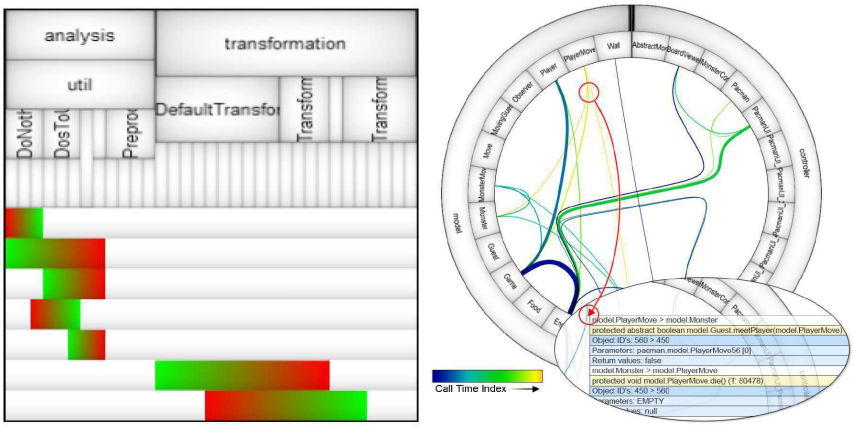


FIGURE 5.5 – (a) Massive Sequence, (b) HEBs avec vue temporelle [HCvW07a, CHZ+07].

Visualiser l'ordre d'exécution des appels de méthodes est important pour comprendre le

fonctionnement du logiciel et son exécution. Dans la figure 5.5(b) les auteurs utilisent un dégradé de couleur du bleu vers le vert puis le jaune pour visualiser l'ordre chronologique d'exécution des appels. Cet ordre est également visualisable grâce à la Massive Sequence qui se lit de haut vers le bas.

5.2.4 Visualisation orientée instance

Les langages à objets sont basés sur un principe simple qui consiste à utiliser une classe comme modèle pour générer des objets (ou instances). Ces objets contiennent les données du programme et la classe définit les opérations de manipulation de ces données. Une fois ces objets créés, ils communiquent entre eux grâce à l'envoi de message. Ces messages permettent aux objets de réaliser les fonctionnalités définies dans les classes.

Greevy, Lanza et Wysseier proposent une technique de visualisation qui est basée sur la *Polymetric Views* (voir chapitre 3, section 3.4.3) étendue en 3D [GLW05] qui permet de visualiser une trace d'exécution d'un logiciel à objets. De la même manière qu'avec la Polymetric Views en 2D, la version 3D utilise des formes géométriques simples pour représenter les composants du logiciel. Les classes sont représentées par des boites grises placées sur un même niveau d'élévation et les instances de chaque classe sont représentées par des boites bleues positionnées au-dessus de la classe qui lui correspond. Les relations d'héritage sont représentées par de simples arêtes noires.

La figure 5.6 présente l'exécution d'un programme objet. Chaque fois qu'une nouvelle instance est créée, elle vient s'ajouter à la colonne qui lui correspond. Les communications entre les instances sont représentées par des arêtes rouges. Ainsi, la trace peut être parcourue en mode pas-à-pas et la coloration est utilisée pour identifier les deux instances concernées par un message un instant donné.

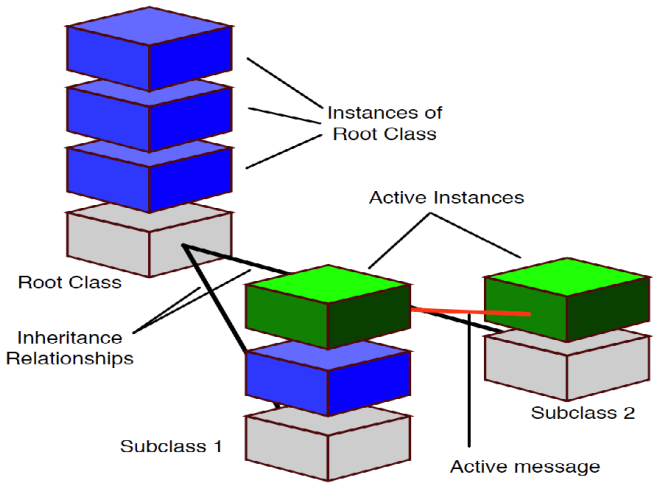


FIGURE 5.6 – Une vue schématique de la vue polymétrique en 3D [GLW05].

La figure 5.7 montre les relations entre les instances d'un module qui permet de lire un modèle avec *Moose* [DGN05]. Les deux points principaux qui peuvent être déduits de cette visualisation sont :

- le très grand nombre de cubes à droite, ce qui signifie que ces classes ont un très grand nombre d'instances.
- le très grand nombre d'arêtes vers la classe `CCEntityTypeFactory`, ce qui signifie que cette classe reçoit ou envoie beaucoup de messages.

Ces informations n'auraient pas forcement été évidentes à identifier sans l'utilisation d'une telle représentation graphique.

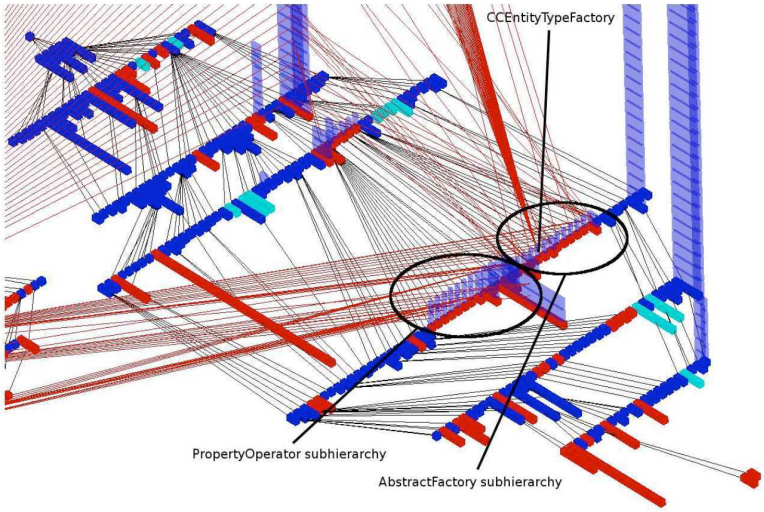


FIGURE 5.7 – Visualisation des relations entre les instances d'un module de Moose[DGN05].

Nous remarquons que toutes les classes qui n'ont pas d'instance à l'exécution occupent des places inutiles dans l'espace 3D. De plus, le placement des éléments sur des plans 3D n'est pas optimisé pour minimiser les croisements d'arêtes. La visualisation offre néanmoins un niveau de détail permettant l'identification du comportement des objets à l'exécution.

Malloy et Power proposent une autre technique de visualisation d'instances basé sur *la métaphore des molécules* (ou graphes) [MP05]. Les instances sont représentés par des sphères colorées et numérotées par rapport à la classe qu'elle instancie. Les cônes entre les objets permettent de visualiser la composition de chaque objet, la base du cône indiquant l'objet qui contient une instance de l'objet à la pointe du cône. Les communications entre les instances ne sont pas représentées mais simplement les relations de composition.

La figure 5.8 illustre cette technique de visualisation. Le placement de chaque nœud dans l'espace 3D se fait d'abord aléatoirement, puis un algorithme itératif calcule le taux d'attirance et de rejet de chaque nœud par rapport aux autres, en calculant le nombre de relations de composition entre les objets. Ainsi plus les objets sont couplés, plus ils s'attirent.

Le but principal de cet outil est de visualiser les objets créés en mémoire et d'identifier concrètement le rôle et l'important de chaque classe. Les expérimentations faites dans [MP05] montrent que le diagramme UML seul permet d'identifier les classes centrales du programme, mais qu'il est utile de voir les instances créées à l'exécution et les relations entre les objets pour réellement comprendre la place et le fonctionnement de chaque classe au sein du programme.

5.3 Visualisation de métriques d'exécution

MetaViz [RWM03, RM05] est une visualisation à base de graphe qui utilise des métriques logicielles en conjonction avec un découpage du programme pour représenter des données statiques ou dynamiques. Nous présentons cette visualisation dans ce chapitre des visualisations dynamiques car nous estimons que celle-ci est particulièrement adaptée pour montrer les relations entre les composants à l'exécution. Le découpage du programme se fait en filtrant les éléments qui ne sont pas pertinents pour la réalisation d'une tâche précise. Par exemple, si l'uti-

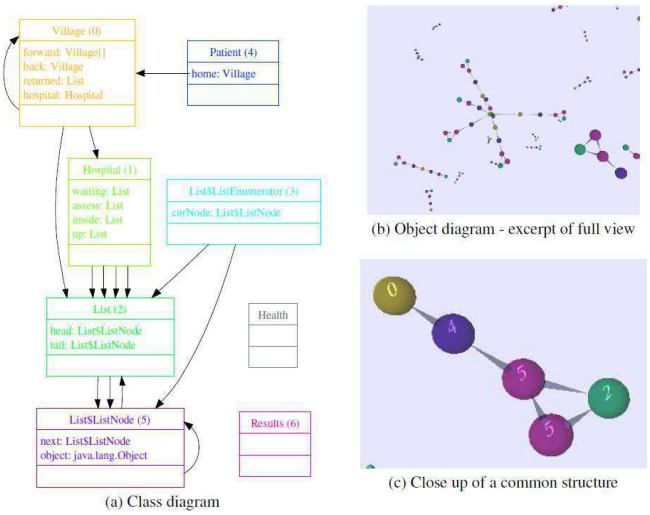


FIGURE 5.8 – Le diagramme de classes et deux visualisations des objets lors de l'exécution de ce programme [MP05].

lisateur s'intéresse à visualiser une classe en particulier, toutes les classes ayant une relation avec cette classe seront montrées, alors que les autres seront filtrées (non affichées).

La figure 5.9 présente un système avec 19 classes, chaque couleur regroupant les classes d'un même paquet. La taille des arêtes représente le couplage entre les deux classes aux extrémités. Par exemple, en haut de la visualisation de la figure 5.9 on remarque un fort couplage entre la classe `CText2D` et la classe `Text` juste à droite.

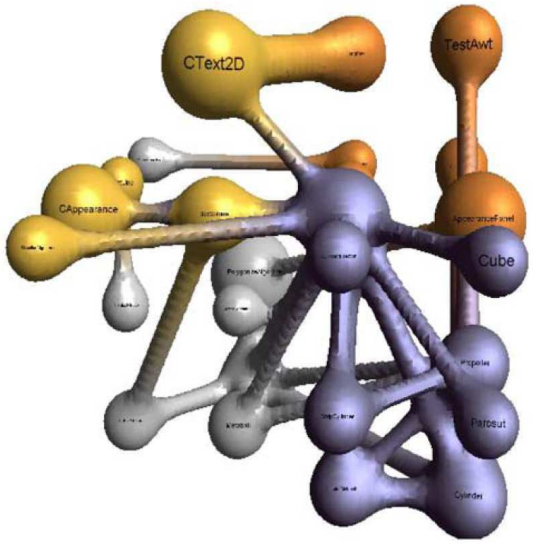


FIGURE 5.9 – Visualisation des relations entre les instances d'un module de Moose[DGN05] avec MetaViz [RM05].

Un problème avec la représentation de très gros graphes en 3D, est le positionnement des nœuds dans l'espace 3D pour minimiser le nombre de croisements et la taille des arêtes. Par conséquent, cette visualisation n'est probablement pas adaptée à la visualisation de toute l'exécution d'un gros programme.

Les auteurs de MetaViz proposent quelques critères de placements pour améliorer le rendu

de leur visualisation. La symétrie, l'optimisation de l'espace 3D, l'optimisation de la densité de distribution, la minimisation de la taille des arêtes, la minimisation des croisements d'arêtes sont des critères pris en compte pour le placement des noeuds. Néanmoins, l'algorithme prenant en compte ces critères ne permet pas le placement efficace d'un très grand nombre de noeuds avec beaucoup de relations.

5.4 Conclusion

Même si de nombreux outils existent, nous constatons que les outils de visualisation de l'exécution des logiciels ne sont pas utilisés dans l'industrie, mais plutôt dans les laboratoires de recherche. Les travaux présentés dans [PRW03] et qui font l'étude empirique de 5 outils de visualisation d'exécution pour la réalisation de taches de compréhension ont révélé que:

- aucun outil n'était capable de réaliser toutes les tâches de compréhension.
- le niveau d'abstraction de chaque visualisation affecte grandement le succès de la réalisation des tâches de compréhension.
- certains outils étaient plus efficaces pour effectuer des tâches de reverse engineering que des tâches de compréhension générale.

Il n'existe pas de visualisation de l'exécution efficace qui couvre tous les aspects de l'exécution et permette ainsi une compréhension totale de l'exécution. Néanmoins certains outils de visualisation de l'exécution facilitent la compréhension de certains aspects de l'exécution. Comme par exemple, le Calling Context Ring Charts présentée en section 5.2.2 qui montre les appels de méthodes. Ou encore la technique Hierarchical Edge Bundles discutée en section 5.2.3 qui met l'accent sur la visualisation des relations entre les classes. Egalement avec MetaViz présentée en section 5.3 qui utilise les métriques pour montrer le couplage entre les classes.

Le gros problème de la plupart de ces techniques de visualisation de l'aspect dynamique est qu'elles ne passent pas bien à l'échelle pour gros logiciels commerciaux. En effet, la représentation de l'exécution grâce aux graphes n'est pas chose facile car ces graphes deviennent très rapidement illisibles, à cause du grand nombre d'arêtes et de nœuds. Certaines techniques de visualisation proposent des filtrages par suppression de noeud [RM05] (voir section 5.3). D'autres proposent de suivre uniquement certaines fonctionnalités déterminés à l'avance par l'utilisateur [SDPK01] (voir section 5.2.2). Enfin, certaines proposent d'agréger au niveau de la classe les informations recueillies au niveau des méthodes [HCvW07a] (voir section 5.2.3). En effet, la mise à l'échelle au niveau de la classe est bien plus simple qu'aux niveau des instances ou des méthodes.

Par conséquent, des travaux restent à faire dans le domaine de la visualisation de l'exécution des logiciels pour représenter l'information avec un niveau de détail fin.

Notons néanmoins la Polymetric View [GLW05] qui est une visualisation au niveau instance (section 5.2.4) qui permet une mise à l'échelle pour des systèmes relativement grands (environ 1000 classes). Ceci est en partie dû aux formes géométriques simples qui sont utilisées pour représenter l'information.

Deuxième partie

Comprendre l'exécution du logiciel

Introduction

Cette partie II traite des contributions scientifiques de cette thèse dont l'objectif est d'étudier et d'apporter des techniques facilitant efficacement la compréhension des logiciels. Seule une analyse approfondie du logiciel permet de recueillir des informations difficiles, voire impossibles, à obtenir autrement. Ces informations sont nombreuses et complexes. Nous travaillons donc sur des techniques qui en permettent une meilleure assimilation. Ces contributions se focalisent principalement sur ces deux aspects: l'analyse du logiciel et les techniques de visualisation facilitant le processus cognitif.

Nous nous focalisons sur l'aspect dynamique des logiciels (l'exécution) plutôt que sur l'aspect statique (le code source). En effet, de nombreux travaux dans la littérature se focalisent exclusivement sur l'analyse du code source mais nous pensons que l'analyse de l'exécution du logiciel doit également être prise en compte pour étudier le logiciel de façon précise. Le chapitre 6 traite donc de l'analyse dynamique des programmes. Plus précisément, il aborde une technique de suivi de l'exécution de programme avec un niveau de granularité très fin nous permettant un suivi intra-méthode allant jusqu'au suivi du typage des instances sur les sites d'appels. Ce type d'information n'est pas forcement disponible avec d'autres analyseurs dynamiques. De plus, notre technique de traçage (et l'outil VITRAIL JBInsTrace correspondant) suit non seulement le programme, mais également les bibliothèques utilisées par le programme. Ceci nous procure donc un suivi très complet, prenant en compte toute l'exécution du logiciel. Nous utilisons ces données sur l'exécution précise des logiciels pour nos travaux d'analyses et également pour nos travaux de visualisation.

En effet, nous pensons que la visualisation des logiciels est un moyen efficace pour faciliter la compréhension du logiciel. Or visualiser un logiciel c'est en dessiner une image et donc réduire la charge cognitive nécessaire pour le comprendre, car l'être humain est plus efficace pour comprendre l'information lorsque celle-ci est représentée de manière concrète (graphiquement) plutôt que virtuelle ou théorique [Mar82, Bie87, Spe90]. Certaines techniques telles que la visualisation grâce à l'utilisation d'une *métaphore du monde réel* consistent à représenter le logiciel dans un contexte familier que l'utilisateur reconnait rapidement. La métaphore de la ville représente un logiciel par une ville, avec des bâtiments représentant les éléments du logiciel, les quartiers représentent des paquets d'éléments, etc. Nous pensons que ce type de représentation est très efficace mais qu'il est très difficile de représenter les relations entre les éléments dans ce genre de visualisation. En effet, tous les bâtiments reposent sur un même plan 3D, ce qui ne facilite pas la représentation des relations. Le chapitre 7 introduit donc notre nouvelle technique de visualisation (et l'outil VITRAIL Visualizer correspondant) qui permet de représenter les relations logicielles au sein de la ville logicielle. Notre représentation regroupe les arêtes en paquets sur plusieurs niveaux d'élévations pour diminuer la complexité visuelle. Ainsi, elle procure un moyen efficace de visualiser ces relations et aide le développeur à comprendre le fonctionnement du logiciel, lui permettant ainsi d'être plus efficace pour la réalisation de certaines tâches.

6

Nouvelle technique d'analyse de l'exécution du logiciel au niveau du bloc de base

Sommaire

6.1 Motivation

L'analyse dynamique de l'exécution des logiciels apporte une très grande quantité d'informations sur le fonctionnement du logiciel ([Bal99, CZvD+09]) et permet de cibler les modules qui jouent un rôle déterminant. En plus des informations sur le chemin d'exécution du code source d'un logiciel, l'analyse dynamique permet de se renseigner sur la manière dont les objets sont créés et communiquent entre eux. Les informations statiques telles que l'héritage des classes ou le couplage d'appels de méthodes statiques ne nous renseignent pas sur la manière dont le programme s'exécute, sur le nombre d'objet en mémoire, ni sur leurs relations à l'exécution. Les informations strictement dynamiques telles que la réflexion et le chargement des classes peuvent être approximées par excès grâce à l'analyse statique ([CMS03, SBC00]). Au contraire, l'analyse dynamique apporte une approximation par défaut des informations de l'exécution ([HDH04]). Néanmoins, l'analyse statique a des propriétés intéressantes car elle renseigne sur le code source

lui-même et permet de contrôler la qualité du logiciel. C'est pourquoi nous pensons qu'utiliser *conjointement* les analyses statiques et les analyses dynamiques permet d'avoir une vision bien plus précise des programmes qu'en s'appuyant simplement sur un seul type d'analyse.

Comme nous l'avons déjà indiqué, les outils d'analyse statique de logiciels à partir du code source sont beaucoup plus courants que les outils d'analyse dynamique. Nous pensons que les difficultés liées au développement d'outils d'analyse dynamique constituent un problème qui freine la recherche dans ce domaine.

Un des objectifs principaux des recherches que nous présentons dans ce chapitre, ainsi que de notre outil correspondant, VITRAIL JBInsTrace [1], est d'apporter une solution au problème exposé précédemment en proposant une technique de traçage de l'exécution du logiciel efficace, qui donnera les informations nécessaires à une visualisation avancée du logiciel.

La trace du logiciel est l'ensemble des instructions que le programme parcourt à l'exécution. Elle est créée en instrumentant le logiciel observé afin d'y insérer du code (un traceur), qui suit (ou trace) les instructions réellement exécutées.

Une gageure consiste à avoir une trace précise permettant des *analyses très fines* de l'exécution d'un logiciel, tout en ayant un ralentissement acceptable de l'exécution du logiciel instrumenté. Pour satisfaire cette gageure, nous considérons l'analyse dynamique de l'exécution selon deux étapes. La première étape consiste à *tracer* [2] le flux d'exécution du logiciel et à enregistrer des informations statiques sur le code source. La seconde étape consiste à *analyser post-mortem* la trace d'exécution et les informations statiques obtenues lors de l'étape précédente. Lors de cette deuxième étape est créé un graphe d'appel, qui sera ensuite utilisé pour calculer nos métriques dynamiques ([DDHV03, ABF04]), et sera exploité par un système de visualisation du logiciel. L'avantage principal de cette technique en deux étapes est d'alléger l'analyse de l'exécution du programme en reportant le calcul des métriques après la fin de cette exécution. Ainsi, l'impact en terme de performance à l'exécution sur le programme instrumenté est diminué. De plus, cette technique séparant la génération de la trace et son exploitation nous permet de ne pas modifier le code (d'instrumentation) du programme quand de nouvelles métriques doivent être calculées.

La plupart des outils dans le domaine de l'analyse dynamique se contentent de tracer l'exécution du logiciel au niveau des méthodes. Même si les nombreux travaux réalisés au niveau du flux d'exécution des méthodes sont très intéressants et apportent beaucoup d'informations, nous voulions avoir un niveau de précision encore plus fin pour nos travaux. Nous souhaitons notamment avoir le flux d'exécution *intra*-méthode pour étudier le polymorphisme sur les sites d'appels, les accès aux attributs, les sauts dus aux boucles ou conditionnelles, les accès aux ressources critiques, etc. Notre analyse de l'exécution des programmes Java a donc été conçue pour être appliquée au niveau du *bloc de base*. Sachant qu'un bloc de base est une séquence d'instructions machine qui se termine par un saut.

Notre technique de traçage (et l'outil VITRAIL JBInsTrace correspondant) instrumente non seulement les classes du programme, mais également les classes du JRE et des bibliothèques, car nous voulons une analyse la plus complète possible, prenant en compte toute l'exécution du logiciel. Nous pensons en effet que ceci est essentiel pour pouvoir comprendre plus complètement le fonctionnement du programme analysé.

En effet, les relations entre les classes, que ce soit les classes du programme ou des bibliothèques, et les classes du JRE sont nombreuses et complexes. Un appel à une simple méthode du

1. Disponible à http://www.loria.fr/~casertap/jbinstrace.html
2. Tracer est le verbe qui exprime l'action de suivre les instructions parcourues à l'exécution.

JRE génère parfois un flux d'exécution très volumineux, ce qui rend difficile la compréhension du déroulement de l'exécution de cette méthode. Par exemple, le premier programme que la plupart des développeurs écrivent est le programme *"hello_world"* qui se programme en Java par l'appel de méthode suivant: `System.out.println("Hello world ...");`. L'exécution du programme *"hello_world"* interprète environ 42550 bytecodes, mais une majeure partie de ces bytecodes sont liés à l'initialisation de la JVM et du programme. La méthode `println(...)` génère à elle seule près de 160 appels de méthode et exécute environ 2660 bytecodes. Ces nombres semblent étonnamment importants, même si les méthodes d'entrées-sorties sont connues pour être complexes. Cet exemple illustre donc l'impact des classes du JRE et l'importance de les analyser en plus des classes du programme.

Avoir une analyse du logiciel exécuté, mais également du JRE et des bibliothèques, offre justement un niveau de complétude aidant la détection de ces problèmes de performance dus aux utilisations des structures de données. Bien évidemment, VITRAIL JBInsTrace dispose également d'une option qui permet de ne pas tracer les classes du JRE et des bibliothèques.

Cependant, instrumenter les classes du JRE est problématique car les classes du JRE elles-mêmes sont utilisées par le traceur (comme des classes de base, par exemple la classe `String`). Nous devons donc introduire un mécanisme qui permette d'éviter la pollution de la trace du programme par la trace du traceur lui-même. Nous détaillons ce mécanisme en section 6.3.3.

L'émergence de nouveaux paradigmes comme la programmation orientée aspects rend plus facile le prototypage de *profileurs*, de *débogueurs*, de *traceurs*, et d'outils de *reverse engineering* ([VBM09]). En effet, ajouter dynamiquement (c'est-à-dire pendant l'exécution) des *bytecodes* à une classe pour suivre son exécution est une manière pratique d'aborder le problème. Notre outil instrumente le bytecode des classes Java (c'est-à-dire ajoute des morceaux de code) à des endroits précis dans le but d'obtenir une trace d'exécution. La difficulté principale de cette instrumentation est dans la recherche d'une conception efficace et transparente de la technique de suivi des programmes ([Moc03, Bru04]).

Dans ce chapitre, nous présentons comment nous avons conçu et implanté notre technique. La section 6.2, introductive, donne une présentation rapide et globale de la technique et l'outil VITRAIL JBInsTrace que nous avons créé pour instrumenter dynamiquement les classes Java. Puis la section 6.3 fournit les détails sur les aspects liés à l'instrumentation du programme et à la production de la trace d'exécution. La section 6.4 se focalise ensuite sur l'analyse, l'exploitation de la trace, expliquant comment les sorties statiques et dynamiques que procure notre traceur peuvent servir à calculer des métriques intéressantes. Enfin, la section 6.5 présente des résultats de performance de l'outil afin de valider notre approche en pratique, et la section 6.6 conclut et présente des perspectives.

6.2 Synthèse des principales caractéristiques de notre technique d'analyse dynamique

Pour notre technique d'analyse dynamique, nous voulions une instrumentation fiable, complète, prenant en compte toutes les classes, y compris les classes créées dynamiquement. Nous avons donc pris grand soin d'avoir une instrumentation complètement dynamique sur les classes Java. Pour ce faire, nous l'avons implantée à l'aide d'un agent Java.

En effet, depuis sa version 1.5, Java fournit un service qui permet à un agent Java d'instrumenter à la volée le programme exécuté par la JVM. L'agent est exécuté par la même JVM, ses classes sont lues par le même chargeur de classes et ont les même règles de sécurité et le

même contexte que le programme exécuté. De cette manière, les classes Java du programme, les classes JRE et des bibliothèques sont interceptées à la volée lorsqu'elles sont lues par le chargeur de classes, et cela même si le programme utilise son propre chargeur de classe (plutôt que celui fourni par défaut par la JVM). Ainsi, même les classes créées ou chargées dynamiquement via la mécanisme de reflexivité de Java (par exemple par l'utilisation de `Class.forName("X")`) sont instrumentées de la même manière que les autres. Plus de précisions sont données sur ce point en section 6.3.1.

Dans notre technique, le flux d'exécution est tracé au niveau du bloc de base. Ce niveau de granularité très fin nous permet notamment d'avoir des informations sur le flux intra-procédural. Avec notre outil VITRAIL JBInsTrace, nous sommes capables de compter combien de fois chaque bytecode est exécuté, mais également d'avoir des informations sur les sites d'appels. Ainsi les informations liées au polymorphisme sont disponibles, ce qui nous permet de faire une analyse *post-mortem* détaillée des types dynamiques des sites d'appels, en comparant le type statique de chaque site d'appel et le type des instances receveurs de ces mêmes sites d'appels. Les changements de contexte des fils d'exécution et le chargement de nouvelles classes Java sont également détectés directement par le traceur. Ils apportent des informations supplémentaires rendant possible une analyse extrêmement détaillée de l'exécution du logiciel. Les appels aux méthodes natives sont quant à eux aussi détectés, au niveau de l'analyse *post-mortem* (voir section 6.3).

La *trace* de VITRAIL JBInsTrace nous procure donc assez d'informations pour reconstruire précisément la pile d'appel du programme analysé et donc réaliser une analyse dynamique très fine.

L'instrumentation de code implique l'injection de nouveaux bytecodes dans les classes Java. Par conséquent, l'exécution du programme originel se trouve modifiée, et le temps passé à exécuter le code instrumenté impacte l'ordonnancement des fils d'exécution. Néanmoins, il faut se rappeler que même sans la présence de notre traceur, l'ordre d'exécution des fils d'exécution du programme n'est pas forcément déterministe. En effet, le temps est de toute façon éminemment variable, car il dépend de très nombreux facteurs extérieurs au programme (JVM, système d'exploitation, autres tâches présentes, etc.). En ajoutant des bytecodes de traçage à ceux du programme, du JRE et des bibliothèques, notre traceur ralentit l'exécution du programme et n'est donc pas neutre sur ce plan.

Ainsi l'analyse de l'exécution de programme avec plusieurs fils d'exécution reste possible car le traceur n'est pas intrusif et ne risque donc pas d'accéder à une ressource critique en cours d'utilisation. Les règles de synchronisation et de restriction d'accès aux ressources partagées définies par le programme restent les mêmes en présence du traceur.

La figure 6.1 résume le fonctionnement global de notre technique de traçage. Lorsqu'une classe est chargée par le `ClassLoader`, l'agent Java intercepte le bytecode de cette classe à la volée (voir section 6.3). Notre `Instrumentor` parcourt alors le bytecode de cette classe et injecte le nouveau bytecode qui va servir au traçage. L'`Instrumentor` sauvegarde également les informations statiques sur les blocs de base, les méthodes et les classes (voir section 6.3.2). Évidemment, le code ajouté n'altère pas la sémantique du programme original mais il influence le déroulement des fils d'exécution (voir section 6.3.2). À l'exécution, le bytecode injecté appelle notre classe `Tracer` (voir 6.3.2.3), qui prend soin de ne pas polluer la trace par des événements générés par le traceur lui-même (voir section 6.3.3). À la fin de l'exécution, toutes les informations

statiques et dynamiques extraites sont écrites dans des fichiers qui sont ensuite étudiés par un analyseur et éventuellement un outil de visualisation (voir section 6.4).

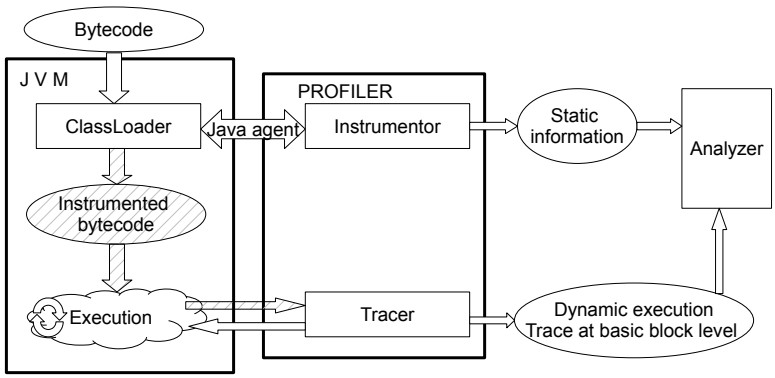


FIGURE 6.1 – Le fonctionnement de notre profileur

6.3 Instrumentation dynamique des classes

Notre instrumentation dynamique transforme uniquement les classes qui sont effectivement chargées lors de l'exécution du scénario, sans altérer le fichier `.class` original sur le disque.

Nous avons fait ce choix car une instrumentation statique aurait impliqué de transformer toutes les classes qui auraient pu être utilisées par le logiciel, quand bien même seul un petit pourcentage de ces classes auraient effectivement été utilisées. Choisir une instrumentation statique aurait également imposé de doubler le nombre de fichiers `.class` (JRE et bibliothèques inclus), l'un servant pour la version instrumentée et l'autre pour la version non instrumentée de la classe. De plus, comme les classes sont souvent disponible sous forme de fichiers JAR, nous aurions été contraints d'extraire les classes, de les instrumenter et de les re-empaqueter sous forme de JAR, en devant répéter cette opération pour chaque nouvelle version de chaque classe. Un autre avantage de l'instrumentation dynamique est qu'elle permet l'instrumentation des classes créées dynamiquement à l'exécution. Pour toutes ces raisons, nous avons décidé d'implémenter une technique d'instrumentation dynamique.

Les trois sections suivantes détaillent notre technique et son implantation. La section 6.3.1 explique tout d'abord le fonctionnement de l'agent Java. La section 6.3.2 détaille ensuite comment l'instrumentation des classes est réalisée. Enfin, la section 6.3.3 présente notre technique de résolution du problème de la pollution de la trace par le traceur.

6.3.1 Agent Java d'instrumentation des classes

Un agent Java est déployé via fichier JAR. C'est un service qui est disponible depuis la version 5 de Java et qui permet d'exécuter un deuxième programme en même temps que le programme principal grâce à l'utilisation de l'option `-javaagent` de la commande Java. Cette commande fonctionne sur toutes les JVMs qui implémentent le service de l'agent Java. VITRAIL JBInsTrace a d'ailleurs été testé sur la VM Java HotSpot v19.0 et la VM IBM v6.1.

Le fonctionnement de l'agent Java est le suivant. La méthode `premain(...)` est appelée par la JVM avant même l'exécution de la méthode `main(...)`, mais après l'initialisation de la JVM. VITRAIL JBInsTrace utilise donc cette méthode `premain(...)` pour enregistrer le transformateur de classe (`Instrumentor`) qui sera ensuite appliqué à toutes les nouvelles classes

qui seront lues par la JVM. Un problème vient des nombreuses classes qui sont lues durant la phase d'initialisation de la JVM (nous appellerons ces classes les classes *"Core JRE"*). En effet, notre transformateur de classe n'est pas encore fonctionnel à ce moment-là, il ne peut donc pas les instrumenter. En revanche, par la suite, toutes les autres classes (non Core JRE) qui seront lues par la JVM seront automatiquement instrumentées avant d'être chargée en mémoire.

De plus, nous savons que la JVM applique un système de chargement paresseux. En d'autres termes, les classes sont lues uniquement lorsqu'elles sont utilisées par le code source qui est exécuté. Ainsi lorsque la JVM démarre sans l'agent Java, seules les classes utiles pour l'exécution de la méthode `main(String[])` sont chargées, puis la méthode `main(...)` est invoquée ([LY99]). Or le processus d'initialisation change lors de l'utilisation de l'agent Java puisque c'est la méthode `premain(...)` qui est appelée en premier. Les classes d'initialisation chargées en mémoire sont donc celles utiles à l'exécution de la méthode `premain(...)`. Dans nos expérimentations, nous comptons environ 350 classes Core JRE qui proviennent principalement du paquet `java.lang`.

Sun-Oracle implémente la méthode `retransform(...)` du paquet `java.lang.instrument` pour permettre l'instrumentation de ces classes Core JRE, mais cette méthode impose des contraintes: l'instrumentation via `retransform(...)` ne peut pas ajouter, supprimer, renommer des champs ou méthodes, changer les signatures des méthodes, ni changer l'héritage ([SM08]). Par conséquent, pour appliquer une transformation dynamique sur les classes chargées lors de l'initialisation de la JVM, notre technique doit prendre en compte ces limitations. Ce qui implique le développement d'une technique de traçage très différente de celles qui existent déjà.

Habituellement les traceurs ajoutent de nouvelles méthodes relais, ou *Wrappers*[1], lors du chargement des classes en mémoire; ils ajoutent également des arguments supplémentaires dans les signatures des méthodes pour suivre le contexte d'appel ([VBM09]). C'est pourquoi peu de travaux incluent les classes du JRE lors du traçage du flux d'exécution.

Néanmoins, Binder *et al.* ([BHM07]) utilisent une instrumentation dynamique pour les classes du programme, avec des classes du JRE instrumentées statiquement (donc avant l'exécution du traceur). L'outil de Binder *et al.* a été réimplanté avec une technique d'instrumentation plus avancée nommée *JP2* ([SMB+11]). Dans cette nouvelle version les auteurs ont également évité de changer la structure des classes dans leur instrumentation. Seule la classe `Thread` a été remplacée dans le JRE par une classe instrumentée statiquement afin d'inclure des attributs `public` supplémentaires. Ces attributs `public` sont donc accessibles par toutes les classes du programme ; de cette manière il n'est pas nécessaire de passer le contexte d'appel par paramètres des méthodes, puisqu'il suffit d'accéder au nouvel attribut de la classe `Thread` pour connaitre le contexte d'appel au sein de ce fil d'exécution. Grâce à cette technique, les métriques et l'arbre d'appel sont calculés au fur et à mesure de l'exécution du programme, ce qui entraîne un certain ralentissement et une limitation au niveau de la complexité des métriques calculées. En effet, plus les métriques ajoutées sont complexes à calculer, plus elles induisent un ralentissement important de l'exécution. JP2 implante donc des métriques simples telles que le nombre de bytecodes exécutés, le nombre d'appels de méthode, le temps d'exécution, etc. Le problème de la détection des méthodes natives persiste avec JP2 car le retour à une instrumentation purement statique est imposé pour détecter l'appel de ces méthodes.

1. Une méthode Wrapper (de l'anglais *Wrapper method*) est une fonction visant principalement en l'appel d'une seconde fonction.

6.3.2 Instrumentation du bytecode des classes

L'instrumentation injecte de nouveaux bytecodes dans les classes. Plusieurs outils et bibliothèques permettent de faire ce genre d'opération.

Nous avons choisi le *framework* ASM ([BLC02, Bru07, Kul07]) qui fournit une bibliothèque pour parcourir les classes Java, ajouter des bytecodes aux classes et réaliser des transformations complexes du bytecode. En effet, ASM peut entièrement modifier le bytecode d'une classe existante ou peut créer dynamiquement de nouvelles classes. Selon l'étude ([Kul07]), ASM est beaucoup plus rapide que les autres bibliothèques d'instrumentation telles que BCEL ([DvZH02, D$^+$01]), SERP ([Whi02]) et Javassist ([Chi04]) et le coût mémoire d'ASM très faible.

Dans les sous-sections qui suivent nous montrons comment nous obtenons les informations statiques des classes chargées, le niveau de granularités dans le traçage du code, comment les événements sont codés et comment le programme est instrumenté. Enfin dans la dernière sous-section nous expliquons comment nous récupérons l'informations de typage des instances sur les sites d'appels.

6.3.2.1 Extraction d'informations statiques

Lorsque les classes sont lues, VITRAIL JBInsTrace les parcourt et effectue trois opérations:
- un identifiant unique est assigné respectivement à chaque classe, méthode et bloc de base;
- les informations statiques sont extraites du code source des classes;
- les classes sont instrumentées (injection de bytecode).

Pour des raisons de performances, il est préférable de manipuler les identifiants de chaque élément en les codant sous forme d'entiers plutôt qu'en utilisant le nom de chaque élément. En effet, la chaine de caractère qui permet d'identifier les éléments est composé du nom complet de la classe, du nom et de la signature de la méthode et du numéro de bloc de base. Ainsi cette chaine comporte en moyenne une centaine de caractères qu'il faut enregistrer pour chaque événement survenant à l'exécution (quelques millions). Enregistrer cette chaine ralentirait donc sensiblement le traceur. Un dictionnaire liant les identifiants au nom de chaque classe, méthode ou bloc de base est donc créé lors de l'initialisation du programme, et les nouveaux identifiants sont créés et ajoutés au dictionnaire lors du chargement de ces classes en mémoire.

Lors du traçage, ces identifiants sont enregistrés dans le fichier de sortie et composent la séquence d'exécution du programme analysé. Le dictionnaire est alors utilisé en grande partie durant l'analyse *post-mortem* pour retrouver la correspondance entre les identifiants enregistrés dans la trace et les éléments exécutés lors du scénario. Les informations statiques et les détails de conception du logiciel sont également sauvegardés au format XML ou CSV pour en faciliter l'exploitation.

Pour l'écriture de ce chapitre, nous utiliserons le format XML dans nos exemples.

Voici ci-dessous un exemple d'enregistrement des informations d'un bloc de base :

```
<basic_block id="5095440"
            metrics="4 0 0 0 1 1 0 1 0 0 0 0 0 0 0 1 0 1">
       <call_site_list>
              <call_site call="java/nio/CharBuffer:hasRemaining()Z"
                        type="INVOKEVIRTUAL"/>
       </call_site_list>
</basic_block>
```

Le champ `metrics` est une liste de métriques au niveau du bloc de base. Dans nos expérimentatiosn actuelles, cette liste est composée des métriques suivantes :

- le nombre de bytecodes,
- le nombre d'opérations en lecture sur un tableau,
- le nombre d'opérations en écriture sur un tableau,
- le nombre d'opérations sur des nombres à virgule,
- le nombre de lectures d'une référence vers un objet,
- le nombre de bytecodes qui peuvent rediriger le flux d'exécution,
- le nombre de bytecodes qui changent le flux d'exécution sans condition (jump, return, etc.),
- le nombre d'opérations de lecture sur un attribut de type référence qui est défini dans la classe,
- le nombre d'opérations d'écriture sur un attribut de type référence qui est défini dans la classe,
- le nombre d'opérations de lecture sur un attribut de type primitif qui est défini dans la classe,
- le nombre d'opérations d'écriture sur un attribut de type primitif qui est défini dans la classe,
- le nombre d'opérations de lecture sur un attribut de type référence qui est défini dans une autre classe,
- le nombre d'opérations d'écriture sur un attribut de type référence qui est défini dans une autre classe,
- le nombre d'opérations de lecture sur un attribut de type primitif qui est défini dans une autre classe,
- le nombre d'opérations d'écriture sur un attribut de type primitif qui est défini dans une autre classe,
- le nombre de bytecodes qui invoquent des méthodes d'une autre classe,
- le nombre d'appels à l'instruction `new`,
- les nombres d'appels virtuels et d'appels statiques.

Le champ `call_site_list` est la liste des sites d'appels présents dans le bloc de base, dans l'ordre d'apparition. Cette information statique est utile pour détecter les appels aux méthodes natives et pour étudier le polymorphisme lors de l'analyse *post-mortem* (voir 6.4). Nous avons choisi de synthétiser les informations sur les blocs de base grâce à une liste de métriques mais nous pourrions sauvegarder tout le bytecode afin de garder toutes les informations. De cette manière, les métriques déjà calculées au niveau des blocs de base seraient calculées au moment de l'analyse *post-mortem*. Cependant, nous avons estimé que les métriques définies ci-dessus décrivent suffisamment les actions effectuées lors de l'exécution des blocs de base pour ne pas avoir à sauvegarder la totalité du bytecode. De plus, sauvegarder tout le bytecode serait coûteux en terme de performances car l'instrumentation serait plus longue. Plus de détails techniques sur la manière dont sont générées les informations statiques sont disponibles sur la page internet de VITRAIL JBInsTrace (http://www.loria.fr/~casertap/jbinstrace.html).

6.3.2.2 Evénements tracés

VITRAIL JBInsTrace surveille quatre types d'évènements, dont deux sont très classiques dans le domaine des traceurs : lorsqu'une méthode commence, et lorsqu'une méthode termine. Dans notre technique, nous traçons également lorsqu'un bloc de base est exécuté et lorsqu'une

classe est chargée. Il n'est pas utile de tracer le début et la fin d'un bloc de base car un bloc de base se termine par un saut dans le bytecode. Lorsqu'un bloc de base est exécuté toute la séquence de bytecodes qui le composent sera donc exécutée, sauf si une exception est levée. La section 6.4.2 explique comment VITRAIL JBInsTrace suit le flux d'exécution dans le cas particulier de la présence d'exceptions.

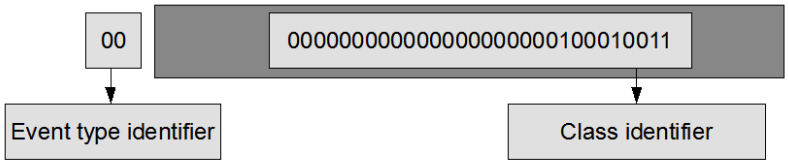


FIGURE 6.2 – Structure de l'entier de 32 bits qui code un évènement: « une classe est chargée ».

Chaque évènement est codé de façon compacte sur un entier de 32 bits (figures 6.3 et 6.2) pour optimiser au maximum les performances du traceur. Ce *nombre codant l'évènement* est composé ainsi:

- les 2 bits de poids forts encodent *le type de l'évènement* : « une méthode commence », « une méthode termine », « un bloc de base est exécuté » ou « une classe est chargée ».
- si l'événement est « une classe est chargée » (figure 6.2), les 30 bits de poids faible encodent le numéro d'identifiant de la classe chargée.
- sinon pour les événements « une méthode commence », « une méthode termine », « un bloc de base est exécuté » (figure 6.3), les 30 bits de poids faibles sont utilisés ainsi:
  - les 19 bits du milieu encodent *l'identifiant unique de chaque méthode.*
  - Si le type de l'évènement est « un bloc de base est exécuté », alors les 11 bits de poids faible encodent *l'identifiant unique du bloc de base dans sa méthode.* Ces bits ne sont pas utilisés pour les évènements: « une méthode commence » et « une méthode termine ».

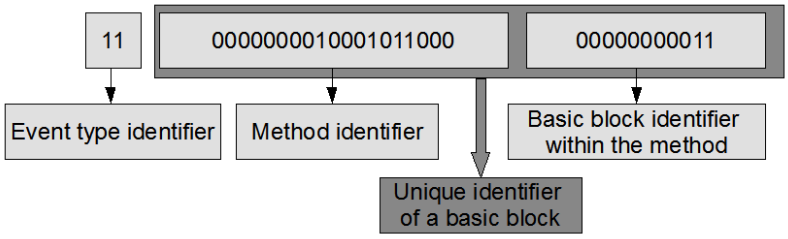


FIGURE 6.3 – Structure de l'entier de 32 bits qui code les évènements: « une méthode commence », « une méthode termine », « un bloc de base est exécuté ».

Cet encodage permet donc d'identifier 524288 méthodes différentes, qui peuvent donc avoir au plus 2048 blocs de base chacune. Nous pensons que ces limitations sont raisonnables. En effet, les résultats obtenus lors de l'analyse du logiciel ArgoUML, qui est un logiciel open source permettant de modéliser des diagrammes UML et qui est assez volumineux, nous montrent que l'exécution d'un scénario banal implique le chargement de 5100 classes, avec environ 12000 méthodes instrumentées, avec au plus 180 blocs de base dans une méthode. De plus, en cumulant l'exécution de cent exécutions de scénarios variés, nous atteignons un total global de 50000 méthodes différentes, loin de la limite des 524288. S'il était besoin, nous pourrions dépasser ces limites en encodant l'événement sur un entier de 64 bits au lieu de 32, mais cela engendrerait

```
public void foo() {

    int i=0;
    while(i<100){

        if(i<50){

            a();
            b(i);
        }

        i++;
    }

}
```

(a) Avant l'instrumentation

```
public void foo() {
    Tracer.eventNotifyWithInstanceType(1084653569,
            (int)Thread.currentThread().getId(),
             this.getClass());
    Tracer.eventNotify(-1062830080,
            (int)Thread.currentThread().getId());
    int i=0;
    while(i<100){
       Tracer.eventNotify(-1062830079,
                (int)Thread.currentThread().getId());
       if(i<50){
          Tracer.eventNotify(-1062830078,
                   (int)Thread.currentThread().getId());
          a();
          b(i);
       }
       Tracer.eventNotify(-1062830077,
                (int)Thread.currentThread().getId());
       i++;
    }
    Tracer.eventNotify(-1062830076,
            (int)Thread.currentThread().getId());
    Tracer.eventNotify(-2136571904,
            (int)Thread.currentThread().getId());
}
```

(b) Après l'instrumentation

FIGURE 6.4 – Notre instrumentation du bytecode Java

une augmentation de l'utilisation de la mémoire et potentiellement un ralentissement.

Soulignons que, ensemble, *l'identifiant de la méthode* et *l'identifiant du bloc de base* forment un identifiant unique pour chaque bloc de base dans le système.

6.3.2.3 Code injecté

Les programmes Java peuvent avoir plusieurs fils d'exécution distincts. VITRAIL JBInsTrace trace donc l'exécution en enregistrant pour chaque évènement le numéro (ou identifiant) du fil d'exécution courant. La classe `Thread` du JRE Java implémente la méthode statique `currentThread()` qui permet à tout instant de connaitre l'identifiant du fil d'exécution courant. Cette méthode étant très peu complexe et implémentée nativement (dans la JVM), son exécution est extrêmement rapide. VITRAIL JBInsTrace fait de nombreux appels à cette méthode car pour chaque appel à la méthode `notifyEvent(...)` de notre classe `Tracer`, nous passons via les arguments *le nombre codant l'évènement* et *l'identifiant du fil d'exécution courant.*

Le code Java qui correspond à notre code d'instrumentation (c'est à dire le code que nous insérons dans le programme, pour faire le traçage) est:

```
Tracer.eventNotifier(eventNumber, (int)Thread.currentThread().getId());
```

La figure 6.4 montre un exemple de morceau de code source instrumenté (en pseudo-Java code). La partie gauche, figure 6.4(a), représente le code source avant instrumentation, et la partie droite, 6.4(b), représente le code source après instrumentation.

Comme illustré dans la figure 6.4, l'instrumentation est faite :

- au début de chaque méthode. Le `nombre codant l'évènement` est composé du type d'évènement « une méthode commence » (en binaire « 01 »), suivi par l'identifiant de méthode.
- à la fin de chaque méthode. Le `nombre codant l'évènement` est composé du type d'évènement « une méthode termine » (en binaire « 10 »), suivi par l'identifiant de méthode.
- au début de chaque bloc de base. Le `nombre codant l'évènement` est composé du type d'évènement «un bloc de base commence » (en binaire « 11 »), suivi par l'identifiant de

méthode et l'identifiant de bloc de base dans la méthode.

Dans le cas particulier où une classe est chargée, la notification de cet évènement se fait de manière interne au traceur. Notre classe `Instrumentor` notifie notre classe `Tracer` du chargement d'une classe et celui-ci sauvegarde l'évènement correspondant dans la trace. Le `nombre codant l'évènement` est alors composé du type d'évènement « une classe est chargée » (en binaire « 00 »), suivi par l'identifiant de la classe.

Avec cette technique, la classe `Tracer` est appelée un très grand nombre de fois, mais nous obtenons des performances tout à fait acceptables (voir section 6.5). Notre technique n'ajoute donc aucune méthode, ni aucun argument aux méthodes. Nous ne modifions pas la sémantique du programme original car seuls des appels de méthodes vers notre traceur sont ajoutés. Le traceur prend ensuite temporairement le relais sur le programme observé, pour exécuter notre code d'enregistrement de la trace, avant de rendre la main.

6.3.2.4 Type dynamique du receveur

Les identifiants de classe ont deux objectifs : décrire le type des instances sur les sites d'appels virtuels et identifier les classes chargées par la JVM. Pour retrouver les identifiants de classes, nous utilisons la réflexion au moyen de la méthode `getClass():java.lang.Class` sur la référence de l'objet `this` ([SHR+00]). En effet, notre traceur est averti qu'une méthode commence au niveau de l'appelé et non au niveau de l'appelant. C'est-à-dire que le `Tracer` est appelé en début de méthode. Le mot clé `this` peut donc être utilisé pour récupérer facilement le type de l'instance courante (receveur) de la méthode. Notre traceur ne se contente pas de suivre les appels de méthodes mais suit également le typage des instances sur les sites d'appels. Pour cela, le code d'instrumentation destiné à tracer les évènements « une méthode commence » est légèrement différent de celui servant à tracer les autres évènements, car nous souhaitons passer le type de l'instance courante en paramètre à la classe `Tracer`. Nous définissons donc une seconde méthode statique `eventNotifierWithInstanceType(...)` de la classe `Tracer` avec 3 paramètres: le nombre codant l'évènement, l'identifiant de fil d'exécution, et le type de l'instance qui exécute la méthode observée.

Le pseudo-code Java inséré au début de chaque méthode est le suivant :

```
Tracer.eventNotifierWithInstanceType(eventNumber,
                (int)Thread.currentThread().getId(), this.getClass());
```

6.3.3 Éviter les perturbation de la trace

Le programme et l'agent (VITRAIL JBInsTrace) sont exécutés en même temps sur la même JVM et VITRAIL JBInsTrace utilise les classes du JRE de la même manière que le programme qui est tracé. Ceci signifie que le programme et VITRAIL JBInsTrace partagent certaines classes du JRE (par exemple `String`), qui sont elles aussi instrumentées. Donc certaines précautions doivent être prises pour ne pas tracer le traceur lui-même, ainsi que pour ne pas avoir de récursion infinie entre les classes du JRE et le `Tracer`.

Pour résoudre ce problème nous avons implanté un système de drapeaux (gardes) qui autorise ou non le traçage des évènements. Le principe est trivial mais des précautions supplémentaires doivent être prises parce que les programmes Java utilisent plusieurs fils d'exécutions (*multi-threaded*). Notre technique gère donc un drapeau par fil d'exécution du programme.

La classe `InProgramCodeFlagsManager` (voir figure 6.6) centralise et gère un drapeau par fils d'exécution (un objet `Boolean` est associé à chaque fil d'exécution du programme). Elle fournit

les méthode `getInProgramCode()` et `setInProgramCode(...)`, qui respectivement retourne et modifie le drapeau correspondant au fil d'exécution courant.

Lorsque les méthodes de VITRAIL JBInsTrace s'exécutent la première opération qui est faite est de vérifier la valeur du drapeau `inProgramCodeFlags[n]` du fil d'exécution courant `n` pour savoir si le traçage est autorisé ou non. En effet, si `inProgramCodeFlags[n]` est vrai alors le traçage est autorisé, sinon cela signifie que VITRAIL JBInsTrace s'exécute et donc que le traçage ne doit pas être effectué. Initialement, tous les drapeaux sont à vrai, le traçage peut donc commencer pour tous les fils d'exécution. Lorsque VITRAIL JBInsTrace intervient (avec l'intervention de la classe `Tracer` ou `Instrumentor`), la méthode `setInProgramCode(n, false)` est appelé pour donner la valeur `false` au drapeau du fil d'exécution courant `n`. Le traçage est alors interrompu sur ce fils d'exécution pendant l'exécution du code de VITRAIL JBInsTrace. Bien entendu, à la fin du traçage d'une évènement ou à la fin d'une instrumentation, VITRAIL JBInsTrace ré-autorise à nouveau le traçage en donnant la valeur `true` au drapeau.

Le diagramme de séquence de la figure 6.5 montre le mécanisme de rejet du traçage d'une méthode du JRE appelé par l'`Instrumentor`.

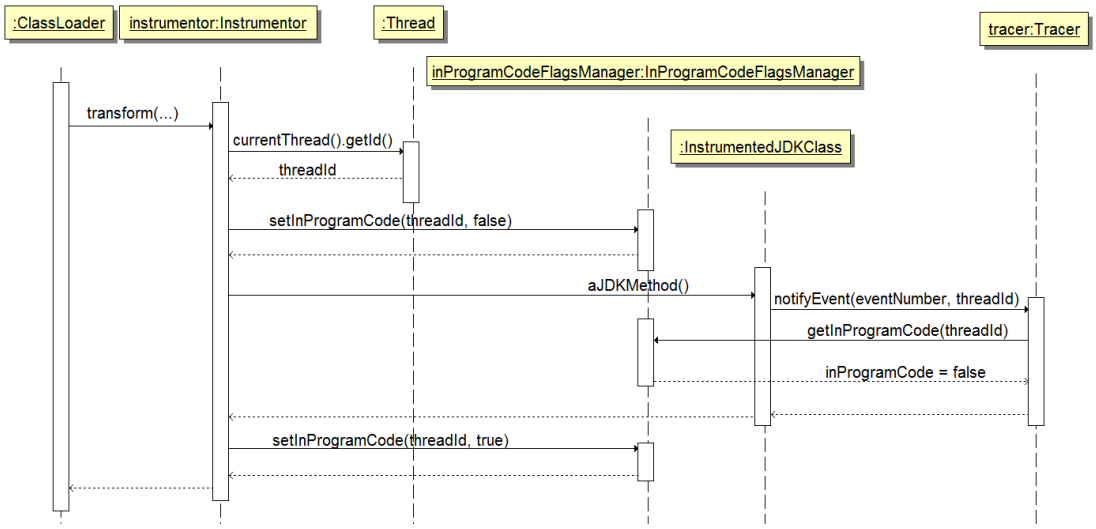


FIGURE 6.5 – Diagramme de séquence montrant les opérations qui permettent d'éviter la pollution de la trace

La figure 6.6 présente le diagramme de classe UML de VITRAIL JBInsTrace.

6.3.4 Le traceur

La classe `Tracer` est un *singleton* qui peut être accédé par toutes les classes du programme. Ce singleton contient un tampon qui sauvegarde la liste des évènements observés avant de l'écrire sur le disque. Le `Tracer` peut fonctionner de deux manières différentes pour produire deux types de traces :

Mode 1 – une seule trace, commune à tous les fils d'exécution, où chaque évènement est lié à son identifiant de fil,

Mode 2 – plusieurs traces, une pour chaque fil.

Avec le mode 1 (une seule trace), l'utilisation de méthodes `synchronized` est obligatoire pour préserver l'ordre des évènements et éviter les *dead locks*. L'idée est de se dire que lorsqu'un évènement est envoyé sur un fil d'exécution. Le traceur accède à la ressource partagé (la trace) pour sauvegarder cet évènement. Si le traceur est interrompu durant ce processus, alors les

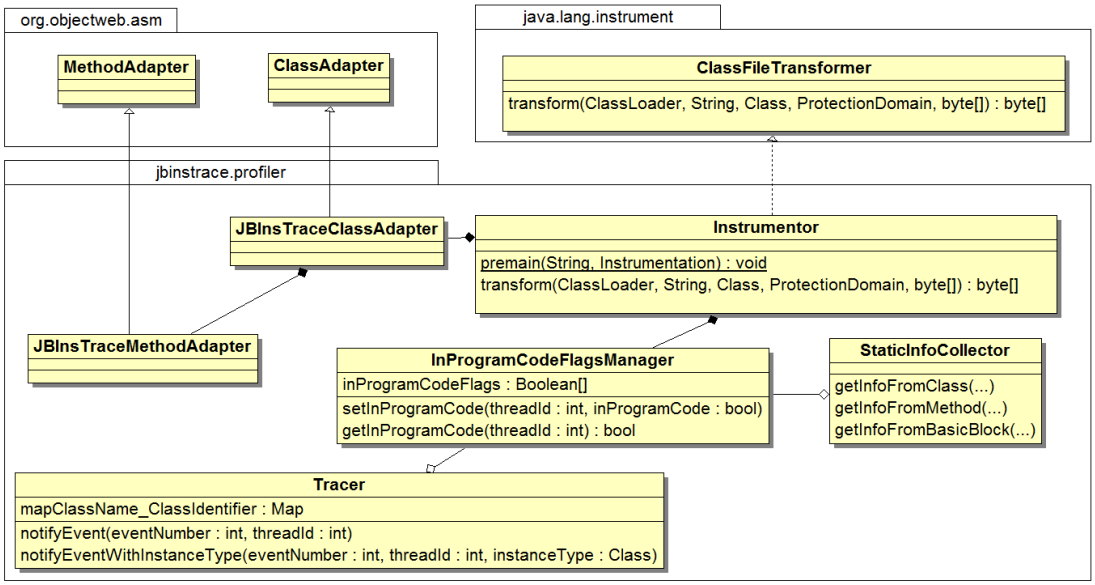


FIGURE 6.6 – Diagramme de classe UML de VITRAIL JBInsTrace.

autres fils d'exécution n'ont pas le droit d'accès à cette ressource. Nous utilisons les `Boolean` de l'attribut `inProgramCodeFlags[]` pour synchroniser l'exécution de JBInsTrace par rapport aux fils d'exécution (l'attribut `inProgramCodeFlags[n]` s'occupe de la synchronisation sur le fil `n`). De cette manière, c'est le même objet qui joue le rôle de drapeau et d'objet synchronisant les fils d'exécution. En revanche ce mode ralentit plus l'exécution du programme, puisque les différents fils d'exécution sont en compétition pour l'accès en exclusivité à des ressources critiques, ce qui provoque des attentes.

Le `Tracer` est appelé de manière intensive par le code instrumenté (à chaque évènement). L'exécution de `notifyEvent(...)` et de `notifyEventWithInstanceType(...)` doit donc être la plus rapide possible. C'est pourquoi seules les opérations suivantes sont effectuées dans ces méthodes :

- vérifier la valeur du drapeau `inProgramCodeFlags[n]` avec `n` qui correspond au fil d'exécution courant (voir figure 6.5),
- sauvegarder le couple `(event number, thread ID)` dans le tampon unique (cas du mode 1) ou sauvegarder le `nombre codant l'évènement` dans le tampon correspondant au fil d'exécution courant (cas du mode 2),
- dans `notifyEventWithInstanceType(...)` une opération supplémentaire consiste à sauvegarder l'identifiant de la classe qui correspond au type de l'instance courante.

6.4 Exploitation de la trace et des informations statiques

Le résultat de l'exécution de VITRAIL JBInsTrace est un fichier qui contient le flot de contrôle du programme, avec une granularité au niveau du bloc de base, et cela pour chaque fil d'exécution. De plus, les informations statiques collectées sur le code source du programme sont également disponibles dans le fichier trace. Ces informations statiques sont ensuite utilisées par notre analyseur de trace VITRAIL JBInsTrace Analyzer (voir la partie droite de la figure 6.1).

Dans cette section, nous expliquons comment nous exploitons la trace de VITRAIL JBInsTrace pour faire une analyse détaillée de l'exécution des programmes Java. La sous-section 6.4.1 détaille comment nous avons construit notre analyseur de la trace de VITRAIL JBInsTrace.

Ensuite la sous-section 6.4.2 décrit comment nous traitons les exceptions qui sont levées à l'exécution. Finalement, la sous-section 6.4.3 explique la détection des appels de méthode native.

6.4.1 Notre analyseur de trace: VITRAIL JBInsTrace Analyzer

La première étape de notre analyseur consiste à lire les informations statiques et à construire la structure de données qui va représenter le programme analysé. Notre analyseur implante un meta modèle des programmes Java qui modélise la notion de bloc de base, et qui est très similaire à celui proposé dans [HPM08].

La deuxième étape consiste à créer la table qui lie les identifiants des classes, méthodes et blocs de base aux instances représentant ces éléments dans le meta modèle. Après cela, VITRAIL JBInsTrace Analyzer met en relation chaque évènement de la trace avec les instances du meta modèle correspondantes et peut ainsi obtenir toutes les informations statiques des éléments (tels que les métriques de cet élément, sa classe parent, etc.)

Par exemple, si un évènement de type « le bloc de base d'identifiant b est exécuté » est rencontré dans la trace, l'identifiant b est recherché dans le dictionnaire liant les identifiants numériques aux instances du meta modèle. Ainsi la méthode contenant le bloc de base b est retrouvée, puis la classe contenant la méthode et toutes les autres informations statiques.

Cette technique permet de re-simuler la pile d'exécution et de calculer des métriques dynamiques très précises. Par exemple, pour chaque évènement de type « une méthode commence », nous pouvons comparer le type statique du site d'appel courant (une information statique) avec le type dynamique de l'instance (une information dynamique, de la trace), ce qui permet de calculer la métrique qui consiste à compter combien de fois un site d'appel prend des instances de types différents.

Les auteurs de [DDHV03] ont défini plusieurs métriques dynamiques intéressantes que nous avons implantées dans VITRAIL JBInsTrace Analyzer. Ces métriques traitent de plusieurs aspects dynamiques : le nombre de bytecodes exécutés, les structures de données utilisées, l'utilisation mémoire, le polymorphisme et la concurrence entre les fils d'exécution. Notre analyseur nous permet également de générer un fichier qui servira d'entrée pour notre outil de visualisation VITRAIL Visualizer ([CZB11]). Cet outil sera abordé dans le chapitre 7.

6.4.2 Traçage des exceptions

Un bloc de base est normalement exécuté de son début jusqu'à sa fin, sauf si une exception intervient. Du point de vue de la trace, les exceptions sont des évènements imprévisibles qui redirigent brutalement le flux d'exécution vers un bloc qui gère l'exception levée. La pile d'appel doit donc être dépilée jusqu'à trouver le bloc `catch` (qui gère l'exception). Par conséquent, les structures `try/catch` doivent être connues : pour chaque bloc de base il faut connaitre la liste des exceptions qui peuvent être levées et le premier bloc de base (de la clause catch `catch`) vers lequel le flux d'exécution peut ainsi être redirigé.

Voici un exemple d'information statique d'un bloc de base qui peut lever deux exceptions différentes:

```
<basic_block id="5095441"
           metrics="7 0 0 1 0 1 0 1 0 0 0 0 2 0 0 1 0 1 0">
 <exception_handler_list>
       <exception_handler basic_block_id="3764234"
                       type="java/io/InterruptedIOException" />
```

```
      <exception_handler basic_block_id="3764235"
                         type="java/io/IOException" />
 </exception_handler_list>
</basic_block>
```

Ainsi nous savons qu'à tout moment, si une exception de type `java/io/InterruptedIOException` est levée dans le bloc 5095411 alors le bloc de base 3764234 sera exécuté. Sinon, si l'exception levée est de type `java/io/IOException` alors c'est le bloc de base 3764235 qui sera exécuté.

6.4.3 Traçage des appels à des méthodes natives

Avec notre technique de traçage, seul le code d'instrumentation en début de méthode informe le `Tracer` si une méthode est effectivement appelée. Par conséquent, les appels vers des méthodes natives, qui ne sont pas écrites en Java et ne sont donc pas instrumentées, doivent être traités différemment pour être détectés. Pour mémoire, notre technique n'ajoute pas de nouvelle méthode (à cause de la limitation de Java sur l'instrumentation des classes Core JRE), l'utilisation de méthodes relais (ou *wrappers*) pour détecter l'appel vers des méthodes natives n'est donc pas une solution dans notre cas.

Notre solution consiste à lister les différents sites d'appels de chaque bloc de base pour vérifier que chacun de ces sites aboutit à l'appel d'une méthode dans la trace. Si aucun évènement « une méthode commence » (issu de la trace) ne correspond à un certain site d'appel (issu des informations statiques) alors on peut conclure que cet appel de méthode s'est résolu sur une méthode native. En effet, les méthodes natives n'étant pas instrumentées, aucun évènement n'est envoyé au traceur et donc aucun évènement n'est enregistré dans la trace pour une méthode native.

En Java, les exceptions qui apparaissent dans certaines méthodes natives sont levées exactement de la même façon que les autres, elles sont donc gérées exactement de la même manière par notre analyseur (voir section 6.4.2).

6.5 Performances: résultats expérimentaux et analyse

Dans cette section, nous montrons les performances de VITRAIL JBInsTrace version 0.6 sur un ensemble de *benchmarks* Java. Nous présentons d'abord les résultats obtenus sur de célèbres logiciels Open Source Java, puis sur la suite de tests de performances *SPECjvm2008*. Nos tests ont été réalisés sur un ordinateur 8-cœurs, 64-bit Intel(R) Xeon(R) CPU E5440 2.83GHz, disposant 16 GO de mémoire vive, utilisant le système d'exploitation Windows Vista SP1. Tous les tests ont été réalisés avec notre traceur en mode 2 (une trace par fil d'exécution).

Les *benchmarks* utilisés sont:

- *ArgoUML v0.28*, le *leader* des logiciels Open Source de modélisation UML. Lors de son démarrage, 5173 classes sont chargées.
- *JEdit v4.3pre17*, un éditeur de texte qui a une grande maturité et beaucoup de fonctionnalités. Des centaines de personnes ont participé à son développement. Lors de son démarrage, 2822 classes sont chargées.
- *Columba v1.0*, un client de messagerie avec une interface graphique. Lors de son démarrage, 3646 classes sont chargées.
- *Apache Ant v1.8.2*, est un outil permettant l'automatisation des tâches de compilation et déploiement de projets Java. Ant est utilisé pour gérer le processus de construction d'applications Java. Nous avons tracé la compilation de Ant en utilisant Ant. Le nombre

de classes chargées s'élève à 1438 classes. Nous avons choisi de le faire se compiler lui-même parce que d'après [AZ08] la compilation de Ant active la plupart des fonctionnalités qui permettent de compiler un programme Java.

– *SPECjvm2008*, une suite de benchmarks qui se focalise sur les performances du JRE. Celle-ci est composée de 9 benchmarks exécutant des algorithmes très différents: *Compiler* compile avec OpenJDK, *Compress* compresse des données, *Crypto* cripte des données, *Derby* manipule une base de donnée, *MPEGAudio* encode du mp3, *Scimark* manipule des nombres flottants, *Serial* convertit des données parallèle-série, *Sunflow* exécute des rendues graphique avec illumination Globale, *XML* valide des documents xml. SPECjvm2008 [Ben08] est souvent utilisé comme benchmark de dé référence pour calculer les performances de composants matériels tels que le processeur ou la mémoire ([SCWP09]).

Un problème lié à l'instrumentation des programmes est qu'elle engendre un ralentissement incontournable. Avec VITRAIL JBInsTrace, trois éléments contribuent à ce ralentissement : le processus d'instrumentation, l'exécution du code injecté (i.e., l'exécution du traceur proprement dit) et enfin la sauvegarde sur le disque de la trace et des informations statiques.

Il est bien connu que l'écriture de données sur le disque est beaucoup plus lente que le stockage en mémoire vive. C'est pour cette raison que VITRAIL JBInsTrace possède deux modes de fonctionnement au niveau de l'écriture sur le disque.

Dans le premier mode, lorsque la quantité de mémoire vive n'est pas suffisante pour contenir la trace, la trace est écrite au fur et à mesure sur le disque. Lors de l'écriture, le programme analysé est stoppé, les données sont écrites puis le programme reprend son exécution. Cette technique a pour intérêt de permettre d'enregistrer des traces d'exécution très grandes, sans avoir de problèmes de mémoire vive.

Dans le second mode, lorsque la mémoire vive est suffisante pour contenir la trace entière, celle-ci reste en mémoire jusqu'à la fin de l'exécution du programme, puis est sauvegardée sur le disque.

Le tableau 6.1 présente les performances de VITRAIL JBInsTrace lors du traçage d'ArgoUML, JEdit, Columba et Ant. Le but de cette expérimentation est de montrer que le programme instrumenté reste utilisable, même lorsque le traceur est actif. Les trois premiers benchmarks sont des logiciels interactifs, ce qui est problématique pour nos mesures de performance car l'intervention de l'utilisateur est nécessaire. Néanmoins, nous souhaitons tester notre analyseur avec des logiciels qui ont un profil semblable aux logiciels commerciaux. ArgoUML en fait partie et les nombreuses publications de recherche l'utilisant comme logiciel pour leurs analyses prouvent l'intérêt des chercheurs pour ce logiciel [DL06, VRDBDR07, CVF11] et bien d'autres.

Pour ne pas perturber les mesures de performance avec l'intervention de l'utilisateur et pour répéter exactement les mêmes scénarios plusieurs fois, nous traçons uniquement le lancement de chaque logiciel. La phase de lancement des logiciels interactif est bien souvent plus demandeuse en ressource que lors de l'utilisation du logiciel (après démarrage). En effet, ArgoUML par exemple met environ 9 secondes à démarrer sur notre machine alors qu'aucune des ces fonctionnalités ne demande 9 secondes pour se réaliser (ou éventuellement l'ouverture d'un très grand projet UML). Par ailleurs, nous avons fait de nombreux tests manuels de l'exécution de logiciels interactifs et nous observons que l'exécution du logiciel durant sa phase d'utilisation a bien souvent des performances plus élevées qu'au démarrage. Nous décidons donc de mesurer les taux de ralentissement uniquement lors du démarrage du logiciel pour ne pas perturber les mesures.

La seconde colonne du tableau montre les temps d'exécution des programmes seuls (sans VITRAIL JBInsTrace). La troisième colonne montre le temps d'exécution du programme avec

le traceur actif, sans compter le temps d'écriture de la trace sur le disque (à la fin de l'exécution). En effet, nous avions assez de mémoire vive pour garder la trace en mémoire pendant toute la durée des différents scénarios et l'écrire à la fin de l'exécution. La quatrième colonne montre donc le temps d'écriture de la trace. La cinquième colonne fait la somme des colonnes trois et quatre et comptabilise donc le temps total d'exécution du programme instrumenté avec le traceur actif.

Logiciels	sans le traceur	avec le traceur		
		exécution	sauvegarde	total
ArgoUML	9	44	152	196
JEdit	6	14	15	29
Columba	7	17	19	36
Ant	5	40	170	210

TABLE 6.1 – Impact de VITRAIL JBInsTrace sur le temps d'exécution (en secondes).

Si nous regardons les temps d'exécution totale avec et sans notre traceur nous constatons que le facteur de ralentissement est de 21.8 pour ArgoUML, 4.8 pour JEdit, 5.1 pour Columba et 42 pour Ant. Les variations de performance entre les benchmarks s'expliquent par les différentes techniques de codage des logiciels et également par la complexité des opérations réalisées par chaque benchmark. Néanmoins, il est clairement souhaitable d'améliorer les performances de notre outil.

Cependant, notre objectif principal est d'obtenir une trace très détaillée de l'exécution d'un programme, du JRE et des bibliothèques, tout en ayant un outil facile d'utilisation et flexible au niveau des informations qui peuvent être collectées. Un tel niveau de détail et de flexibilité implique nécessairement un coût en terme de performance.

Considérons maintenant les différentes étapes qui contribuent à ce ralentissement. Le tableau 6.1 révèle que le plus gros facteur de ralentissement est le temps passé à sauvegarder les informations sur le disque. Ce temps de sauvegarde dépend principalement des capacités du matériel physique utilisé. En effet, l'écriture sur un disque dur SSD sera significativement plus rapide que sur les disques durs classiques.

Au-delà du ralentissement "brut" observé sur le logiciel instrumenté, qui est une indication de performance importante, il est également judicieux de vérifier si les programmes interactifs restent utilisables ou non avec le traceur actif. Il est donc intéressant de soustraire du temps total celui passé à écrire les données sur le disque à la fin de l'exécution du benchmark, afin d'obtenir le niveau de ralentissement effectivement *perçu* par l'utilisateur. En faisant ceci, nous obtenons des facteurs de ralentissement perçu nettement plus raisonnables, : 4.9 pour ArgoUML, 2.3 pour JEdit, 2.4 pour Columba et 8 pour Ant. Ceci est confirmé par notre propre expérience qui nous a montré les programmes instrumentés et tracés selon notre technique restent tout à fait utilisables.

Dans le cas où la trace ne tient pas en mémoire vive, la trace peut être enregistrée en mode incrémental, ce qui va stopper le programme (lorsque la mémoire est pleine) pour enregistrer la trace au fur et à mesure, par morceaux. Le temps d'exécution total du programme instrumenté dépendra alors beaucoup de la quantité de mémoire disponible et du type de disque dur utilisé.

La figure 6.7 présente les temps d'exécution de VITRAIL JBInsTrace sur les 9 benchmarks de SPECjvm2008. Ces benchmarks sont plus petits que les 4 applications Java présentées précédemment mais ils sont non-interactifs et exécutent beaucoup d'opérations. Le facteur de ralentissement est calculé de cette manière: $\frac{oprations\ par\ seconde\ sans\ le\ traceur}{oprations\ par\ seconde\ avec\ le\ traceur}$. Pour cette expérience, le temps d'écriture de la trace sur le disque n'a pas été pris en compte pour les raisons énoncées

plus haut.

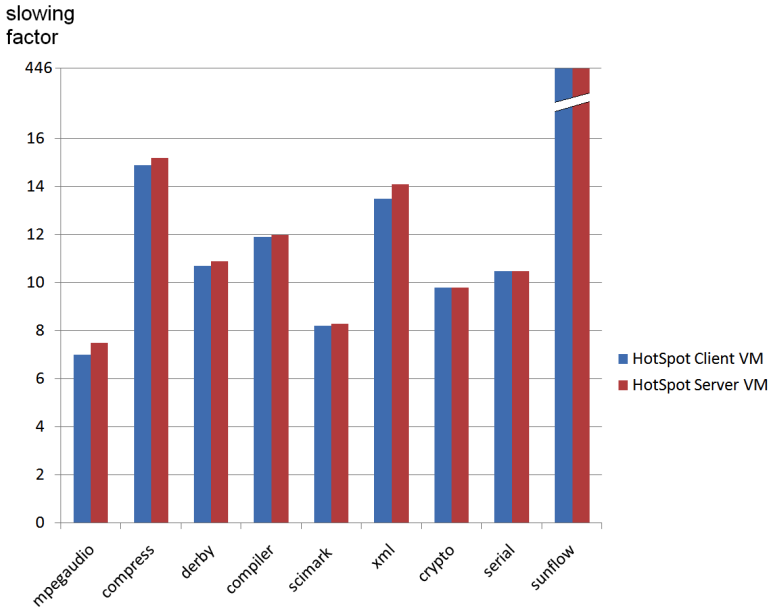


FIGURE 6.7 – Résultats de performance sur les tests de performances SPECjvm2008.

Les différences observées entre les 9 benchmarks de SPECjvm2008 6.7 et les 4 applications Java 6.1 sont dues aux différentes natures des deux ensembles de benchmark. En effet, les 4 applications Java sont des programmes interactifs avec beaucoup de fonctionnalités alors que chacun des 9 benchmarks de SPECjvm2008 réalisent, en boucle, une opération bien spécifique (compression, compilation, cryptage, etc.). De se fait, la complexité à l'exécution (en terme de nombre de saut) est plus importante avec les 9 benchmarks de SPECjvm2008. Intrinsèquement, plus de blocs de base sont exécutés et donc plus d'évènement sont traités par le traceur. Ce phénomène explique donc les différences entre les deux ensembles de benchmarks utilisés mais nous constatons que tous les résultats obtenus sont cohérents.

Le test sunflow est un cas pathologique qui n'est pas représentatif des performances constatées sur VITRAIL JBInsTrace. En effet, sunflow est un benchmark qui exécute des opérations de rendu graphique en 3D sur plusieurs fils d'exécution en exploitant au maximum les capacités de calcul en parallèle de notre processeur. L'hypothèse que nous émettons pour tenter d'expliquer cette dégradation des performances est que l'ajout du code instrumenté vient perturber considérablement l'exécution du rendu graphique car les calculs de matrice de rendu sont constamment interrompus par les envois de message avec le traceur. Ces messages s'exécutant sur plusieurs fils d'exécution et l'objet `Tracer` est le seul à recueillir les évènements provenant de ces fils d'exécution. Les performances d'exécution se voient donc diminuée. Cette hypothèse doit encore être validée. Nous proposons une solution pour ce problème de performance dans la section suivante (section 6.6).

Comme le montre la figure 6.7, les mesures ont été réalisées avec les JVMs HotSpot Client et HotSpot Server, avec très peu de différences entre les deux JVM.

6.6 Conclusion et perspectives

Dans ce chapitre, nous avons présenté notre technique de traçage des programmes Java, implantée par notre logiciel VITRAIL JBInsTrace.

Notre technique instrumente le bytecode des classes à l'exécution et est entièrement dynamique, ce qui permet d'instrumenter également les classes générées dynamiquement. Elle prend en compte aussi bien les classes du programme que les classes du JRE et des bibliothèques (si souhaité), ce qui permet d'observer de façon très complète le comportement des programmes. La granularité de la trace obtenue est au niveau du bloc de base, ce qui permet d'observer de façon très précise le comportement des programmes.

Le résultat d'un tel traçage apporte une très grande quantité d'informations. Les résultats expérimentaux ont montré que les performances de notre *tracer* sont raisonnables et que les logiciels interactifs tracés sont toujours utilisables avec le traceur actif. Même si la vitesse d'exécution n'était pas notre but principal, le ralentissement perçu par l'utilisateur reste acceptable.

Parmi les autres techniques de traçage Java, on peut noter *JP2* [SMB+11] qui instrumente statiquement la classe `Thread` du JRE afin d'inclure des attributs `public` supplémentaires qui enregistrent le *Calling Context Tree*[1] *(CCT)* à l'exécution. Grâce à cette technique, les métriques et l'arbre d'appel sont calculés au fur et à mesure de l'exécution du programme, ce qui entraîne un certain ralentissement et une limitation au niveau de la complexité des métriques calculées. JP2 implante donc des métriques simples telles que le nombre de bytecodes exécutés, le nombre d'appels de méthode, le temps d'exécution, etc. Le problème de la détection des méthodes natives persiste avec JP2 car le retour à une instrumentation purement statique est imposé pour détecter l'appel de ces méthodes.

JP2 est semblable à VITRAIL JBInsTrace sur plusieurs points et les deux ont été développés au même moment. Bien que la technique d'instrumentation de JP2 soit efficace, nous obtenons avec VITRAIL JBInsTrace de meilleurs résultats de performance (tant au niveau mémoire qu'au niveau du facteur de ralentissement), tout en étant plus souple car entièrement dynamique. La grosse différence entre nos deux techniques réside en la capacité de VITRAIL JBInsTrace à récupérer le typage des instances sur les sites d'appels. En effet, cette information est cruciale pour nos études du polymorphisme à l'exécution.

Nous pourrions faire plus de tests de performances pour avoir des résultats encore plus précis sur les performances de VITRAIL JBInsTrace. Les tests potentiels seraient:

- l'utilisation de la machine virtuelle d'IBM pour voir s'il existe de grandes différences entre les deux agents Java.
- tester les performances globales en utilisant un disque SSD.
- tester les performances sur un grand ensemble de programmes Java. Pour cela nous avons déjà récupéré le Qualitas Corpus [TAD+10] (le plus gros corpus de logiciel Java à notre connaissance) et souhaitons analyser l'exécution de la plupart des logiciels présents et faire plus de tests de performances.
- comprendre en détails le problème de performance avec le benchmark sunflow et implémenter la solution plus efficace que nous décrivons ci-dessous.

Une solution pour tracer des programmes exploitant énormément les fils d'exécution et avoir de bonnes performances serait d'avoir un objet singleton `Tracer` par fils d'exécution. Ainsi, chaque fil communiquerait avec son `Tracer` qui serait sur le même fil, ce qui éviterait la congestion qu'il y a quand tous les fils d'exécution communiquent avec un `Tracer` unique (qui s'exécute sur le fil 1 par exemple). Il s'agit bien du mode de fonctionnement 2 (voir section 6.3.4) pour lequel nous ajouterions un traceur par fil d'exécution. Ceci peut être l'objet de travaux futurs.

Pour les raisons expliquées en section 6.3.1, des limitations existent pour l'instrumentation

1. Le *Calling Context Tree* est une structure de données représentant toutes les informations d'exécution de chaque fil du programme.

des classes Core JRE (classes chargées durant la phase d'initialisation de la JVM). En effet, il n'est pas possible d'ajouter, de supprimer, de renommer des champs ou méthodes, de changer les signatures des méthodes, ni de changer l'héritage ([SM08]) des classes. C'est limitations sont contraignantes et quelques possibilités d'utilisation supplémentaire s'ouvriront à nous lorsque ces limitations seront levées. En effet, dès lors, nous pourrions ajouter la méthode `finalize()` à chaque classe du programme, du JRE et des bibliothèques pour étudier le moment où les objets sont détruits par le ramasse-miettes (*garbage collector*).

Fondamentalement, la technique ne change pas mais lever cette limitation augmenterait les possibilités d'utilisation du traceur.

La levée des limitations sur les classes Core JRE permettrait également l'implantation de méthodes relais, (ou wrappers) pour la détection des méthodes natives. En effet, ces méthodes relais permettraient de détecter et d'enregistrer directement dans la trace l'appel aux méthodes natives. Actuellement nous traquons les appels aux méthodes natives en comparant les méthodes attendues (dans le code) et les méthodes exécutées (dans la trace). Cette effort de détection des appels aux méthodes natives serait donc complètement inutile avec l'utilisation des méthodes relais, ce qui devrait améliorer nos performances.

Une autre direction serait d'étudier les résultats du traçage pour avoir une meilleure connaissance de l'exécution de logiciels. Au moment de la rédaction de cette thèse, nous avons encore de nombreux travaux en cours pour étudier les données résultant de l'analyse.

Notre traçage permettant de recueillir le typage des instances sur les sites d'appels, nous voulons exploiter au maximum cette fonctionnalité. Nous cherchons notamment à mesurer le taux de polymorphisme dans les programmes. En effet, nous aimerions savoir si les sites d'appels sont plutôt monomorphes ou plutôt polymorphes. La réponse nous permettrait de mieux comprendre les habitudes de programmation des développeurs Java d'une part, et de faire des optimisations au niveau de la JVM concernant le choix de la méthode ciblée par chaque site d'appel d'autre part.

Nous exploitons la capacité de VITRAIL JBInsTrace Analyzer à fournir à la fois les métriques statique et les métriques dynamique en comparant les métriques de couplage statiques et dynamiques pour déterminer laquelle est un bon prédicteur des changements dans les logiciels. En effet, des travaux dans le domaine de l'analyse statique montrent que les métriques de couplage statique permettent de prédire les changements du logiciel [ASJ01]. Nous faisons l'hypothèse que l'évolution du code source du logiciel est fortement liée à l'exécution du logiciel. Donc si notre analyse est restreinte aux modules qui jouent un rôle lors de l'exécution du logiciel, les mesures de couplage sur cet ensemble pourraient donner des résultats plus précis que les mesures statiques qui sont faites sur l'ensemble du logiciel.

Un autre objectif de nos recherches est d'apporter des outils qui facilitent la compréhension du logiciel. Pour cela nous pensons que la visualisation des logiciels a les caractéristiques nécessaires pour faciliter le processus cognitif de l'utilisateur face à la grande quantité d'information contenue dans le logiciel. En effet, nos analyses dynamiques sur des scénarios d'exécution d'ArgoUML révèlent des millions d'appels de méthode. Etudier l'ensemble des appels de méthode sous format textuelle est très difficile, voire impossible. Au contraire, utiliser des abstractions et métaphores visuelles facilite la cognition. Dans le chapitre 5 nous expliquons comment nous représentons les relations du programme grâce à une visualisation de la ville logicielle.

7

Nouvelle technique de visualisation du graphe d'appel dynamique

Sommaire

7.1 Introduction

Dans le chapitre précédent, nous avons détaillé notre traceur d'exécution et les différentes informations obtenues grâce à nos analyses. Dans ce chapitre, nous abordons maintenant une façon d'exploiter au mieux ces informations grâce à la visualisation.

En effet, les grands logiciels que nous utilisons ont une structure et un mode de fonctionnement qui est de nature très complexe. Ceci est bien souvent problématique pour les développeurs qui s'aventurent dans le code source du logiciel pour l'améliorer ou corriger des bogues. Le processus de développement et de maintenance peut ainsi être ralenti de façon importante lors de cette phase de compréhension du logiciel ce qui peut induire un non-respect des délais, parfois très courts, et peut conduire à un surcoût imprévu voir à l'échec du projet. Pouvoir visualiser le logiciel d'une façon globale, rapide et efficace est donc selon nous un aspect crucial pour améliorer les pratiques de développement ainsi que le qualité finale du produit.

Le premier point à visualiser doit être la structure du logiciel, qui est un aspect statique. De nombreuses visualisations s'y emploient, que nous avons recensées en section 3 page 15.

Mais bien entendu, pour comprendre un logiciel, il ne suffit pas de voir sa structure, il faut aussi comprendre comment ses différents composants interagissent. La section 5 étudie l'état de l'art sur la visualisation de l'aspect dynamique du logiciel.

Si les travaux ont été relativement nombreux sur le domaine de la structure du logiciel, avec des résultats très intéressants même s'ils ne sont pas encore répandus dans l'industrie, la situation est nettement moins avancée en ce qui concerne la visualisation des interactions ou des relations du logiciel. Sur ce dernier plan, les travaux sont moins satisfaisants, car malgré les travaux existants, il n'y a toujours pas de visualisation qui fournisse une représentation vraiment satisfaisante des interactions dans le logiciel. En effet, certains travaux tentent de représenter les relations du logiciel avec un graphe dont le placement des noeuds est optimisé pour plus de lisibilité [HMM00] mais les algorithmes de placement peinent encore à représenter lisiblement les très gros logiciels. Nous avons donc décidé d'orienter nos travaux de recherche dans cette direction.

Il existe différents types de relations dans le logiciel. D'un point de vue statique, nous avons par exemple les relations d'héritage, et les relations d'appels (statiques) de méthodes qui permettent entre autres la réutilisabilité (décomposition du système en composants) et l'adaptabilité des objets grâce au polymorphisme. Ce type de relation constitue un aspect très important des langages à objets et qui renseignent sur la structure du logiciel, se comptent par centaines voire par milliers pour les logiciels de grande taille tel que ArgoUML. D'un point de vue dynamique, un autre type de relation encore plus présente dans le logiciel est les relations par appel (dynamique) de méthode. En effet ces relations peuvent se compter par millions pour certains logiciels. Un simple programme tel que *Hello World* en Java charge des centaines de classes lors de son exécution et des milliers de relations d'appel de méthode apparaissent entre les différents objets, classes et méthodes. On comprend aisément qu'avoir un moyen efficace de visualiser ces relations aiderait fortement le développeur à comprendre le fonctionnement du logiciel et à être plus efficace pour la réalisation de certaines tâches.

Visualiser les relations d'un logiciel est un sujet qui a déjà fait l'état de nombreuses publications [CZ11b, Die07]. Les techniques employées pour représenter un logiciel sont variées : elles peuvent être graphiquement en deux dimensions ou en trois dimensions, reposer sur des métaphores du monde réel ou non, etc.

Néanmoins une représentation *efficace* des relations d'un logiciels reste un sujet de recherche d'actualité. Le domaine de la visualisation des logiciels [CZ11b] prend donc en compte la façon dont les êtres humains perçoivent l'information [SFM99]. En effet, meilleure est la perception, plus la charge cognitive nécessaire est réduite [SR96, PBG98, Tud03]. Nous pensons que l'utilisation de métaphores facilite la compréhension de la visualisation (et donc du logiciel). De cette manière, la visualisation apporte une vue « tangible » et intuitive du logiciel [Sta98, WL07b, Die07]. Or l'être humain est plus efficace pour comprendre l'information lorsque celle-ci est représenté de manière concrète (graphiquement) plutôt que virtuelle ou théorique [Mar82, Bie87, Spe90]. Ainsi représenter graphiquement le logiciel est un bon moyen pour permettre une compréhension plus rapide et plus précise de la structuration du logiciel et de ses fonctionnalités. Les mécanismes de perception de l'être humain doivent donc être pris en compte pour améliorer certains aspects cognitifs [SFM99, SR96, PBG98, Tud03]. Des expérimentations empiriques ont montré qu'utiliser des techniques de visualisation lors du développement logiciel augmente les chances de réussite [Gra92, Zha03].

Les *graphes* ont toutes les caractéristiques requises pour modéliser les relations d'un logiciel [Mun97, HMM00, LN03, NL05, SS06]. Les éléments d'un logiciel sont associés aux nœuds du graphe et les relations aux arêtes. Néanmoins, il faut bien se rendre compte que visualiser toutes les relations d'un logiciel revient à visualiser un énorme graphe avec beaucoup d'arêtes et de nœuds. Lorsqu'aucune technique de visualisation n'est utilisée, la représentation de ce très gros graphe a de grandes chances d'être confuse car comportant beaucoup de congestions d'arêtes et de nombreux recouvrements de nœuds, ce qui rend impossible l'examen détaillé d'une relation particulière ou même la compréhension du fonctionnement global d'un logiciel.

Pourtant avoir une vue globale d'un logiciel est très important. En effet, elle permet de déterminer les composants qui requièrent une attention particulière tout en donnant une vue du système dans sa globalité [SZ00].

Quelques principes directeurs permettent d'améliorer la représentation des graphes :

- substituer un groupe d'objets par un seul [BD07].
- placer les nœuds de manière à minimiser le nombre de chevauchements [NL05].
- regrouper les arêtes avant de les séparer à nouveau [Hol06].
- utiliser une vue 3D pour naviguer et trouver une vue compréhensible, sans trop de chevauchements [HMM00].

Dans ce chapitre, nous décrivons notre technique de visualisation nommée *3D Hierarchical Edge Bundles*, abrégée par *3D-HEB*, qui est basée sur ces principes directeurs.

Notre contribution consiste en l'apport d'une solution, basée sur la représentation des graphes en 3D, permettant de pallier les problèmes relatifs à la représentation de très gros graphes. Pour cela nous combinons une métaphore de la ville et une manière efficace de dessiner les relations d'un bâtiment à un autre.

Effectivement, nous pensons que la métaphore de la ville permet déjà une bonne représentation du logiciel sur deux critères [WLR11]: la représentation implicite de la structure du logiciel et la représentation implicite des métriques du logiciel. Notre contribution ajoute donc un troisième critère en visualisant les relations de manière efficace au sein de cette ville logicielle. Nous combinons et adaptons les deux techniques décrites en section 7.2 pour une visualisation des relations plus claire et qui facilite la compréhension du logiciel.

Ce chapitre est organisé en 4 parties. Dans la section 7.2, nous expliquons les techniques de visualisation qui nous ont inspirés pour développer notre propre technique 3D-HEB. Dans la section 7.3, nous expliquons les deux types de placement qui serviront à construire les métaphores de villes. Ensuite dans la section 7.4, nous détaillons la manière dont nous avons placé les points d'attraction hiérarchique qui serviront à guider les arêtes dans l'environnement 3D. La section 7.5 montrent les résultats de notre technique, en donnant des exemples de visualisations de relations avec notre outil VITRAIL Visualizer. Et enfin nous concluons en section 7.6.

7.2 État de l'art

Dans cette section, 7.2, nous expliquons les techniques de visualisation qui nous ont inspirés pour développer notre propre technique 3D-HEB. Il nous semble judicieux de détailler ces techniques ici plutôt que dans le chapitre d'état de l'art 3 page 15 car nos travaux s'en inspirent directement. Néanmoins, nous ne détaillons ici que les caractéristiques pertinentes de ces visualisation dans le cadre de nos recherches.

Notre technique de visualisation s'inspire de deux visualisations existantes et toutes aussi intéressantes l'une que l'autre. La première est la métaphore de la ville [Die93, Die94, DF98], qui permet de créer des visualisations représentant la hiérarchie du logiciel. La seconde technique

est celle des *Hierarchical Edge Bundles (HEB)* [Hol06, HCVW07b] qui représente les relations des logiciels en 2D en limitant les croisements d'arêtes grâce à la formation de groupes d'arêtes.

Nous avons déjà proposé un état de l'état très conséquent dans la première partie de ce manuscrit. Nous ne décrirons donc ici que succinctement les deux techniques sur lesquelles la notre est basée. La métaphore de la ville est décrite dans la sous-section 7.2.1, et les hierarchical edge bundles dans la sous-section 7.2.2.

7.2.1 Métaphore de la ville

La visualisation d'un logiciel grâce à l'utilisation d'une *métaphore du monde réel* consiste à représenter le logiciel dans un contexte familier, en utilisant des représentations graphiques, des formes que l'utilisateur reconnait automatiquement [GR93, DSGA$^+$00, KM00, Plo02, KvdWVW01]. D'après [RDSGA$^+$00], cette technique permet une reconnaissance plus rapide de la structure globale d'un logiciel et une meilleure compréhension des situations plus complexes.

Ces avantages peuvent compenser le désavantage principal des visualisations 3D, à savoir la désorientation dans l'espace 3D. De plus, les visualisations utilisant des métaphores du monde réel se basent sur la compréhension naturelle et intuitive de l'être humain du monde qui l'entoure [GR93, DSGA$^+$00, KvdWVW01, Plo02]. Dans un tel monde métaphorique, la navigation au sein de l'espace 3D est facilitée [KM00]. Par exemple, la ville représentant le logiciel est divisée en arrondissements, eux-mêmes divisés en rues, puis en bâtiments, etc.

La contrainte principale de la métaphore de la ville est que cette dernière est représentée sur un (seul) plan [LSP08], ce qui est problématique pour la représentation des relations entre éléments. Même si certains travaux tentent de donner de la hauteur à la métaphore de la ville, comme par exemple les travaux présentés chapitre 3 section 3.4.3 avec la métaphore des iles et des villes où les iles sont représentées à différents niveaux d'élévation [PEQ$^+$07], cette métaphore de ville reste assez plate.

Même si la visualisation de la ville n'est pas forcement totalement réaliste, nous considérons comme [YG03] que certains écarts à la réalité ne gênent pas la compréhension. Au contraire, nous pensons que simplifier les formes géométriques et retirer des détails inutiles permet de se focaliser sur les informations importantes de la visualisation et de diminuer la charge cognitive.

La métaphore de la ville a été très largement étudiée et a donné lieu à beaucoup de publications scientifiques [LSP05, DSP08, WL07b, WL08b, AD07, PEQ$^+$07]. Les expérimentations dans [WLR11] montrent que l'utilisation de CodeCity (un outil de visualisation utilisant une métaphore de la ville) améliore les performances des tâches réalisées sur le logiciel étudié (+24%) ainsi que le temps utilisé pour réaliser ces tâches (-12%), par rapport à l'utilisation d'Eclipse pour explorer le code et d'Excel pour explorer les métriques. Les expérimentations avec l'outil VERSO dans [LD07] montrent qu'utiliser une métaphore de la ville aide à détecter les anomalies de conception.

7.2.2 Technique du Hierarchical Edge Bundles (HEB)

Les logiciels à objets ont une structure particulière dans laquelle les méthodes sont définies dans des classes, qui sont elles-mêmes incluses dans les paquets. Cette arborescence peut être montrée de manière efficace grâce à des représentations d'arbre tel que le *Treemap* [JS91, Shn92], *Balloon tree*, *Radial Layout* [SZ00, LSP05], etc.

Holten [Hol06] propose la visualisation Hierarchical Edge Bundles pour montrer les relations du logiciel sous la forme d'un graphe 2D placé par-dessus la représentation de l'arborescence. Les composants du logiciel sont les nœuds du graphe et les relations sont les arêtes. L'idée

principale de cette technique est qu'elle introduit des points imaginaires qui attirent les arêtes et les regroupent afin de réduire l'encombrement de la visualisation et en améliorer la lisibilité.

Ces points appelés *Hierarchical Attraction Points (HAPs)* (en français les points d'attraction hiérarchiques) sont définis pour chaque élément du logiciel et placés en fonction de la position de cet élément dans la hiérarchie du logiciel. En réalité les HAPs forment un arbre qui décrit la hiérarchie de l'arborescence.

L'intérêt principal de cette technique réside dans le fait que les arêtes sont affectées par le pouvoir d'attraction des HAPs et que par conséquent, les arêtes sont courbées et regroupées à chaque niveau de la hiérarchie.

7.3 Placements de la ville

Avant d'expliquer dans la section 7.4 comment nous avons couplé, dans un espace 3D, la visualisation de la ville avec la technique du Hierarchical Edge Bundle, nous expliquons brièvement dans cette section les différents types de placement qui serviront au placement des bâtiments de la ville.

En effet, le placement de objects graphiques dans la métaphore de la ville peut être réalisé de plusieurs façons différentes.

Bien souvent, la structure de base de la ville (c'est à dire le plan global, sur lequel les bâtiments sont posés) reflète la structure hiérarchique des paquets du logiciel. La structure des paquets étant une structure arborescente, tel que tous les nœuds sauf la racine ont un unique parent, toutes les représentations d'arbres sont envisageables. Dans nos travaux, nous avons implémenté deux placements différents de manière à montrer que les 3D Hierarchical Edge Bundles (3D-HEB) peuvent être appliqués quel que soit le placement des éléments graphiques (ie. le plan de ville).

Dans la suite de cette section, nous présentons ces deux placements implémentés dans notre outil VITRAIL Visualizer.

La structure hiérarchique utilisée dans nos exemples est définie dans la figure 7.1. La flèche entre la `classe 2` et `classe 4` signifie qu'il existe une relation d'appel entre ces deux classes.

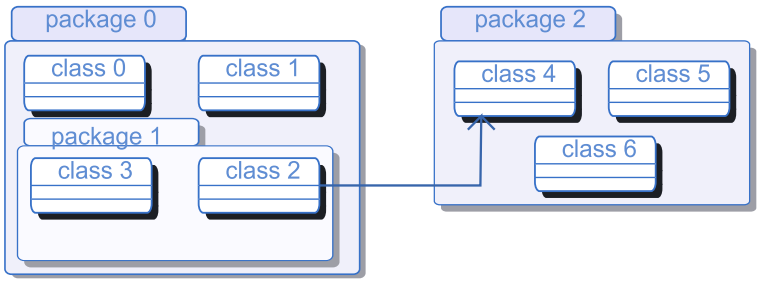


FIGURE 7.1 – Exemple simple de hiérarchie de logiciel.

7.3.1 Placement imbriqué

Habituellement, un *Treemap* imbrique des boîtes les unes dans les autres, avec une taille de boite proportionnelle à la taille de l'élément représenté (sachant que la taille de chaque élément dépend de sa taille et de celle de tout ses descendants). Dans le cadre de la représentation de l'arborescence des logiciels, cette taille peut représenter le nombre de fichiers que contient chaque paquet (descendants inclus) ou encore le nombre de lignes de code contenues dans tous

les fichiers du paquet (descendants inclus). D'un point de vue de la représentation, la taille du Treemap est ajustable car il est possible de multiplier par un coefficient la surface de chaque boite pour agrandir la représentation.

Dans notre technique, nous utilisons une variante de Treemap à laquelle nous avons ajouté des options pour avoir plus de contrôle sur la taille des boîtes composant le Treemap. Notre technique nous permet ainsi d'avoir un contrôle total sur la taille des classes, de l'espace entre les paquets, etc. Pour cela, nous traversons la hiérarchie du logiciel des feuilles jusqu'à la racine et nous calculons la taille de chaque nœud en fonction des tailles de ses descendants. Ensuite, nous représentons graphiquement chaque élément avec la taille calculée. Les sous-paquets sont dessinés au-dessus de leur paquet parent. Les classes sont ensuite posées sur le paquet auquel elles appartiennent.

La figure 7.2 représente la structure hiérarchique de la figure 7.1 grâce à notre adaptation du *placement imbriqué* (ou *Nested Layout*).

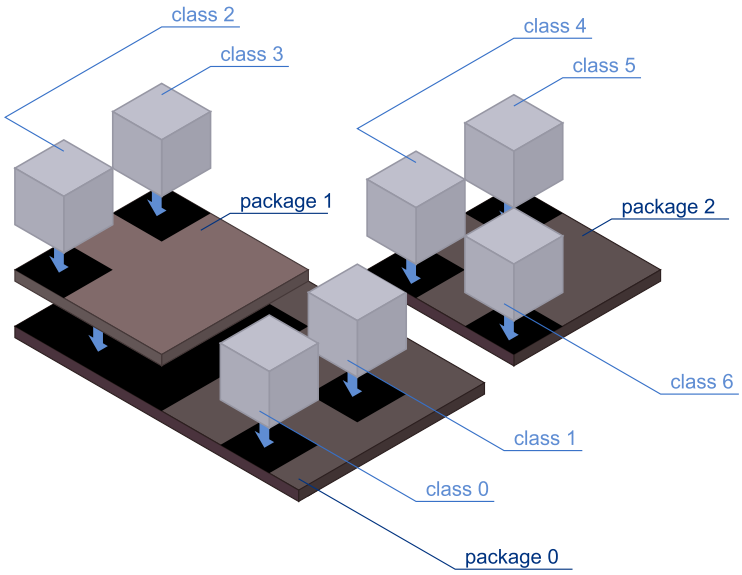


FIGURE 7.2 – Notre placement imbriqué

7.3.2 Placement en rues

Notre second placement est de type *placement en rues* (ou *Street Layout*) car il donne l'impression de représenter des rues avec les bâtiments de chaque côté. Ce placement est inspiré par celui décrit dans [SL10]. Les paquets sont représentés par les rues, tandis que les sous-paquets sont placés perpendiculairement à leur paquet parent et que les classes contenues dans les paquets sont représentées autour des rues auxquelles elles appartiennent.

La figure 7.3 représente la hiérarchie de la figure 7.1 avec le placement en rues.

7.4 Dessiner les relations

Il existe de nombreuses publications sur la métaphore de la ville pour la représentation d'un logiciel, mais très peu proposent un moyen de représentation efficace des relations dans la ville [AD07, AP07, PEQ+07]. L'idée de base qui est proposée par les chercheurs du domaine est de dessiner les arêtes en allant directement d'un bâtiment à un autre. Le problème est que

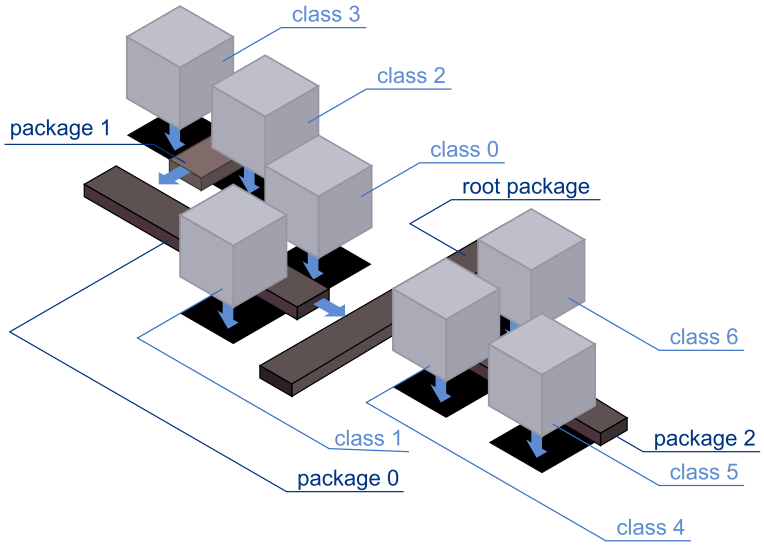


FIGURE 7.3 – Placement en rues

tous les bâtiments de la ville sont représentés au même niveau d'élévation car ils reposent sur un seul plan. Ainsi, représenter les liens grâce à des arêtes allant directement d'un bâtiment à un autre n'est pas très efficace car la représentation des très nombreuses relations engendre de nombreux recouvrements dans la représentation de la ville. De plus, en utilisant une arête droite pour représenter une relation entre deux bâtiments, il est possible que cette arête traverse d'autres bâtiments (non impliqués dans la relation représentée), ce qui rend la représentation de la relation plus confuse.

Notre contribution s'inspire de la représentation (2D) des Hierarchical Edge Bundles que nous adaptons en 3D en plaçant des points d'attraction hiérarchique (3D-HAPs) dans un espace 3D (voir section 7.4.1). Ces points ont pour effet d'attirer les arêtes à travers la visualisation. De cette manière, les arêtes sont regroupées, ce qui résulte en une visualisation plus lisible et plus aisément compréhensible.

Les sections suivantes détaillent notre contribution. Nous expliquons dans la section 7.4.1 comment notre placement des 3D-HAPs guide les arêtes. Dans la section 7.4.2 nous détaillons comment nous choisissons les 3D-HAPs pour qu'ils n'attirent que certaines arêtes. Dans la section 7.4.3 nous parlons de la puissance d'attraction de chaque 3D-HAP et comment cette attraction modifie la représentation. Finalement, dans la section 7.4.4 nous décrivons comment nous avons utilisé la couleur pour représenter la direction et quantifier le nombre de relations.

Il faut remarquer que la courbe qui est donnée à chaque arête n'a pas de sémantique, elle est simplement là pour éviter les croisements disgracieux. Nous montrons plusieurs exemples de représentations dans la section 7.5.2.

7.4.1 Placement de points d'attraction hiérarchique dans l'espace 3D

Comme nous l'avons déjà expliqué, notre technique est inspirée de celle des Hierarchical Edge Bundles 2D [Hol06], que nous avons adaptée pour notre représentation 3D. Le principe de base de notre technique est le placement des points d'attraction dans l'espace 3D en tenant compte de la hiérarchie du logiciel (l'arborescence des paquets). Ces points d'attraction hiérarchiques 3D (3D-HAPs) sont utilisés pour conduire les arêtes d'un 3D-HAP à un autre à travers la visualisation.

La figure 7.4 schématise notre technique de placement des 3D-HAPs. Il y a sept 3D-HAPs au niveau `class level`, chacun étant relié à une classe.

Il y a un 3D-HAP au niveau `level 1` qui correspond au paquet `package 1`. Finalement, deux 3D-HAPs sont au niveau `level 0` et correspondent aux paquets racines `package 0` et `package 2`.

On peut constater qu'avec notre placement des 3D-HAPs (sur l'axe vertical), nous inversons l'ordre de représentation de la hiérarchie par rapport au placement des boites du Treemap. En effet, le Treemap représente la structure dans l'ordre: `level 0`, `level 1` et `class level` (de la boite englobante en bas vers la boite la plus imbriquée la plus en haut) alors que les 3D-HAPs (sur l'axe verticale) sont représentés d'abords au niveau `class level`puis au `level 1` et enfin au `level 0`.

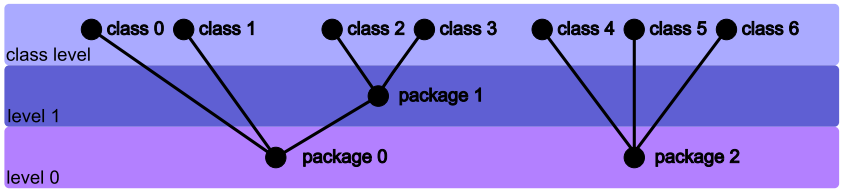


FIGURE 7.4 – Décomposition en niveaux

La figure 7.5 montre comment les 3D-HAPs sont placés dans l'espace 3D. Tout d'abord, nous déterminons pour chaque HAP les deux coordonnées qui vont le positionner sur l'axe du milieu de l'élément du Treemap qui lui est associé. Ensuite la troisième coordonnée représente le niveau d'élévation du 3D-HAP et celle-ci dépend de la profondeur de l'élément qui est représenté par le 3D-HAP dans l'arborescence : plus l'élément est profond dans la hiérarchie d'héritage, plus le 3D-HAP est proche du Treemap. Par conséquent, les 3D-HAPs les plus élevés au-dessus de la ville sont les paquets racines (*Root*), tandis que 3D-HAPs les moins élevés correspondent au classes.

Avec notre technique, il est possible de configurer le niveau d'élévation maximal pour ne pas éloigner trop les 3D-HAPs de la représentation de la ville. Ceci permet de voir la ville et les relations sans devoir faire de trop grands sauts visuels.

7.4.2 Points d'attraction hiérarchiques 3D pertinents pour une relation

Cette section décrit comment nous choisissons les 3D-HAPs qui composent le chemin que parcourt chaque relation à travers la visualisation 3D.

Avec notre technique 3D-HEB, une relation entre deux éléments est représentée par une arête qui passe par plusieurs 3D-HAPs. Notre algorithme cherche le plus petit chemin, en terme de quantité de 3D-HAPs parcourus, entre ces deux éléments.

L'algorithme qui permet de tracer une arête d'un bâtiment à un autre en utilisant les 3D-HAPs est expliqué ci-après.

Pour cela, nous définissons les prédicats suivants :

- $path(X, Y)$: l'ensemble des 3D-HAPs parcourus depuis l'origine X vers la destination Y [1].
- $higher3DHAP(s)$: le 3D-HAP avec la plus grande élévation de l'ensemble s.
- $level(P)$: le niveau d'élévation du 3D-HAP P.

1. Nommer X et Y « origine » et « destination » est une convention purement géométrique et pratique : elle n'implique pas que la relation entre X et Y soit orientée.

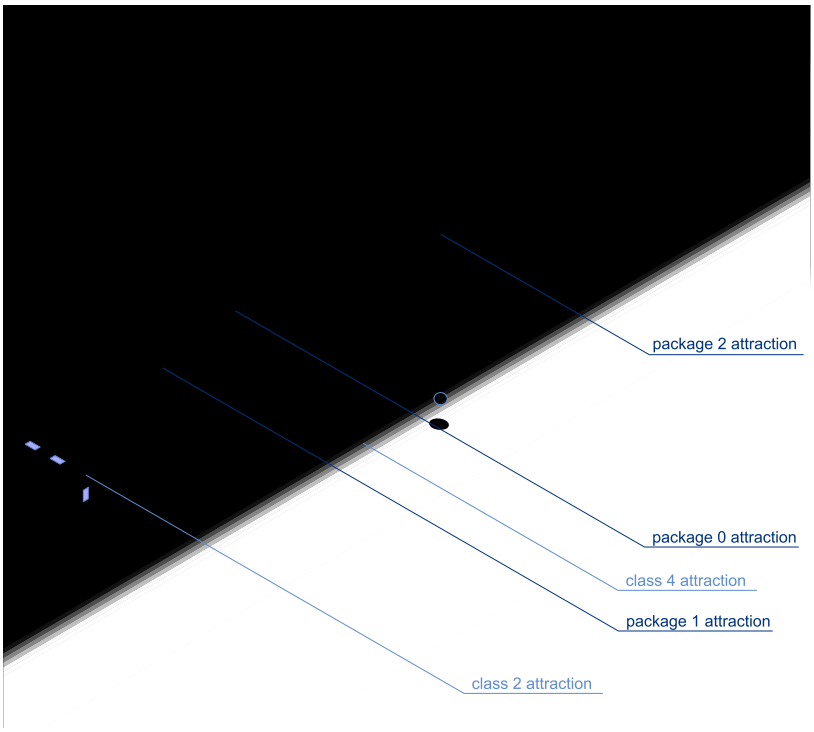


FIGURE 7.5 – Placement des points d'attraction hiérarchiques 3D (3D-HAPs)

– *Root*: le 3D-HAP qui correspond au plus haut niveau dans l'arbre d'arborescence des paquets.

Nous calculons l'ensemble des 3D-HAPs, depuis le 3D-HAP de l'élément d'origine jusqu'au 3D-HAP qui correspond à l'élément destination. Nous effectuons l'opération ensembliste de différence symétrique entre les deux ensembles suivants :

– l'ensemble des 3D-HAPs depuis l'origine O jusqu'au *Root*. Nous appelons cet ensemble "l'ensemble des 3D-HAPs de l'origine".
– l'ensemble des 3D-HAPs depuis la destination D jusqu'au *Root*. Nous appelons cet ensemble "l'ensemble des 3D-HAPs de la destination".

La différence symétrique permet d'éliminer tous les 3D-HAPs qui apparaissent dans les deux ensembles. Ces 3D-HAPs sont inutiles pour la représentation de la relation car, au niveau de 3D-HAP le plus élevé, nous souhaitons visualiser les relations entre paquet

$$path(O, D) = path(O, Root) \Delta\ path(D, Root)$$

Quand on veut représenter une relation orientée avec notre technique 3D-HEB, il est essentiel de différencier les chemins d'origine et de destination. Par conséquent, le chemin $path(O, D)$ peut être divisé en deux sous-chemins de 3D-HAPs : un chemin lié à l'origine (que nous appelons $pathOrigin$) et un chemin lié à la destination (que nous appelons $pathDestination$).

Ces chemins sont calculés ainsi:

$$pathOrigin = path(O, D) \backslash path(D, Root)$$

$$pathDestination = path(O, D) \backslash path(O, Root)$$

Nous verrons plus loin, dans la section 7.4.4 que nous utilisons la couleur pour différencier ces deux ensembles de 3D-HAPs qui composent le chemin d'une relation.

7.4.3 Pouvoir d'attraction

De la même manière que dans la version originale de la technique (2D) HEB [Hol06], nos 3D-HAPs attirent les arêtes en tenant compte d'un paramètre β, qui permet de définir le pouvoir d'attraction de chaque 3D-HAP. Cette attraction a pour effet de courber les arêtes dans l'espace 3D. Dans cette section nous expliquons les calculs qui ont été implémentés pour créer cette courbure par rapport au paramètre β.

La figure 7.6 montre une vue schématique d'un groupe d'arêtes. Cette figure visualise deux relations : une de la classe `class 2` vers la classe `class 4` et une autre de la classe `class 0` vers la classe `class 6`. Les arêtes sont donc attirées par les 3D-HAPs pour parcourir le chemin depuis le premier 3D-HAP correspondant à la classe origine, jusqu'au 3D-HAP de la classe destination.

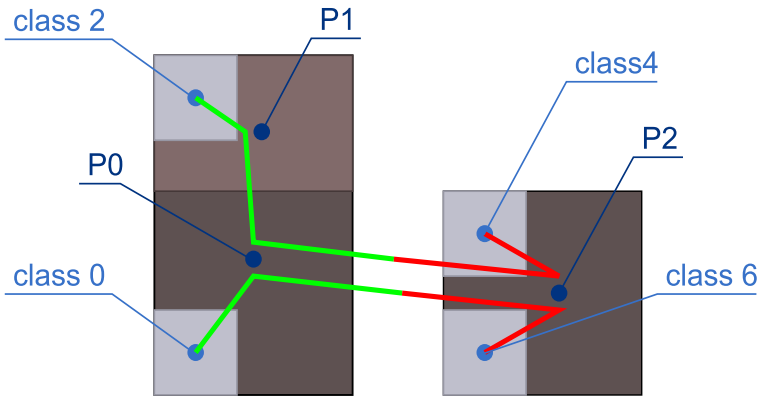


FIGURE 7.6 – Exemple de vue schématique d'un groupe d'arêtes.

Les figures 7.7 et 7.8 montrent une représentation 2D des 3D-HAPs de la relation entre `class 2` et `class 4`. Les points P_0, P_1, P_2 sont les 3D-HAPs qui représentent respectivement `package 0, package 1, package 2`.

Dans la figure 7.7, nous appliquons une projection des 3D-HAPs sur la ligne OD. Cette projection est utilisée pour calculer la puissance d'attraction des 3D-HAPs sur les arêtes.

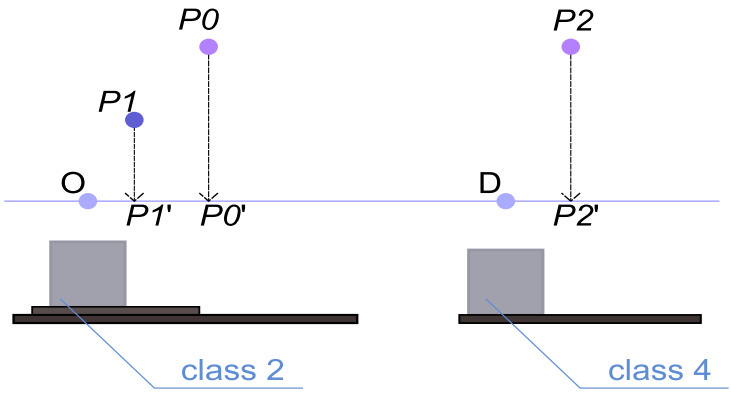


FIGURE 7.7 – Projection des 3D-HAPs sur la ligne OD

La figure 7.8 montre trois représentations avec trois différentes valeurs de β. Lorsque $\beta = 0$ les arêtes ne sont pas attirées par les 3D-HAPs, donc l'arête prend une direction droite qui va directement du 3D-HAP O (correspondant à la classe origine) vers le 3D-HAP D (correspondant à la classe destination). Au contraire, lorsque $\beta = 1$ l'arête est complètement attirée par tous

les 3D-HAPs qui composent le chemin de la relation et passe donc par ces points. Toutes les valeurs intermédiaires de β permettent d'exercer une force plus ou moins importante qui permet de courber plus ou moins les arêtes en les rapprochants plus ou moins des 3D-HAPs.

Nous définissons ci-dessous les formules mathématiques que nous avons implémentées pour arriver à ce résultat (nous prenons P_0 de la figure 7.8 comme exemple) :

$P'_0 = O + \frac{\vec{OD} * (\vec{OP_0} \cdot \vec{OD})}{||\vec{OD}||^2}$

$P''_0 = P_0 * \beta + (1 - \beta)(P'_0 * \beta)$

avec :

O : le 3D-HAP de la classe origine (au niveau `class level`)

D : le 3D-HAP de la classe destination (au niveau `class level`)

β : le pouvoir d'attraction $\beta \in [0, 1]$ des 3D-HAPs

P_n : le 3D-HAP n

P'_n : la projection de chaque P_n sur la ligne OD

P''_n : les interpolations des 3D-HAPs en fonction du facteur β.

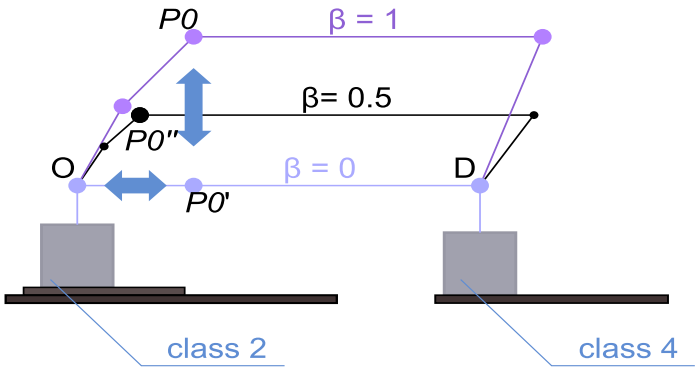


FIGURE 7.8 – Interpolation des P_n en fonction du facteur β.

Dans la section 7.4, nous avons montré que les 3D-HAPs sont placés à différents niveaux d'élévation et que le plus bas correspond au niveau `class level`. Ce niveau est différent des autres car il assure la connexion entre les 3D-HAPs et la représentation de la ville. Ainsi tous les 3D-HAPs au niveau `class level` sont placés sur un même plan indépendamment de la taille de chaque bâtiment. Nous n'appliquons pas le facteur β sur ces 3D-HAPs au niveau `class level` car celui-ci est juste le point de départ et d'arrivée des relations.

Si $\beta = 0$, alors les 3D-HAPs n'ont aucune influence sur les arêtes car leurs pouvoir d'attraction est nul. Par conséquent, toutes les arêtes représentées restent sur le niveau d'élévation `class level` , formant un amas d'arêtes aplati sur un même plan (voir section 7.12).

Chaque 3D-HAP implémente le même algorithme d'attraction, donc si plusieurs relations quittent le même 3D-HAP d'origine, les arêtes les représentant vont prendre exactement le même chemin et vont être représentées par une seule et unique arête jusqu'à ce que leurs chemins divergent. Pour résoudre ce problème nous avons décidé de disperser les arêtes autour du premier 3D-HAP d'origine afin d'avoir un point de départ légèrement différent pour chaque arête. Le point de départ étant légèrement différent pour chaque départ d'arête, le trajet calculé sera lui aussi légèrement différent mais suffisamment pour distinguer les différentes relations.

7.4.4 Quantification et direction

La visualisation du nombre de relations constitue une source d'information importante pour les développeurs.

En effet, une arête représente une relation entre deux éléments mais cette relation peut avoir plusieurs occurrences. Par exemple, imaginons que la relation entre `class 2` et `class 4` de la figure 7.1 soit une relation d'appel avec 50 occurrences. Dans ce cas, nous devons représenter ces 50 occurrences afin de mieux détecter l'importance de cette relation par rapport aux autres.

Face à ce problème, nous avons expérimenté plusieurs techniques. Dans un premier temps, nous avons augmenté le diamètre des arêtes en fonction du nombre d'occurrences, mais cette solution n'a pas été efficace car la taille des arêtes était difficilement visible lors de leur regroupement. Puis nous avons décidé de représenter la quantification des relations grâce à la couleur. Quatre nuances de vert différentes sont utilisées pour la coloration des arêtes.

Nous utilisons la mesure statique des quartiles sur l'ensemble des relations afin de diviser le nombre de relations en quatre ensembles de même taille. Nous associons les quatre teintes de vert au nombre d'occurrences de chaque arête. Les directions des relations sont toujours représentées du vert vers le rouge. Pour les relations unidirectionnelles, une extrémité de l'arête est colorée en vert (l'origine) l'autre extrémité en rouge (la destination). Le rouge signifiant donc: aucune occurrence de la relation en partant de cette classe. La figure 7.9 montre le résultat obtenu. De cette manière, chaque extrémité des relations est coloré par le poids (c'est-à-dire le nombre d'occurrences) de la relation.

Dans le cas où la relation est bidirectionnelle, les deux extrémités sont colorées en vert par rapport à leurs poids respectifs. De cette manière, nous visualisons la direction dominante de la relation.

Dans le cas où la relation est unidirectionnelle, la couleur vert indique la direction et le poids de la relation alors que la destination reste rouge. Cette utilisation des couleurs est une analogie avec les feux de signalisation du trafic routier, le vert signifiant que la circulation est possible et le rouge que la circulation est stoppée. Cette approche avec l'utilisation de la couleur a été implémentée pour réduire le nombre d'arêtes, plusieurs relations identiques sont maintenant représentées par une seule et même relation avec un vert sombre. Néanmoins, il aurait été possible de représenter absolument toutes les relations, chacune par une arête différente mais nous aurions pu rencontrer des problèmes de performance du passage à l'échelle avec de gros logiciels. Avec notre technique de coloration des arêtes, la couleur rouge semble dominer la visualisation mais il est très important de visualiser ces relations unidirectionnelles.

Nous avons expliqué en section 7.4.2 que le chemin d'une relation est composé de deux sous-ensemble de 3D-HAPs : un ensemble de 3D-HAPs qui correspond à l'origine et un ensemble qui correspond à la destination. Le changement de couleur intervient lors de la jonction de ces deux ensembles, lorsque l'arête passe entre $higher3DHAP(pathOrigin)$ et $higher3DHAP(pathDestination)$. Ceci permet de garder une vision claire de la direction et des quantifications des relations.

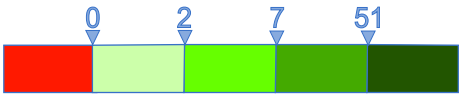
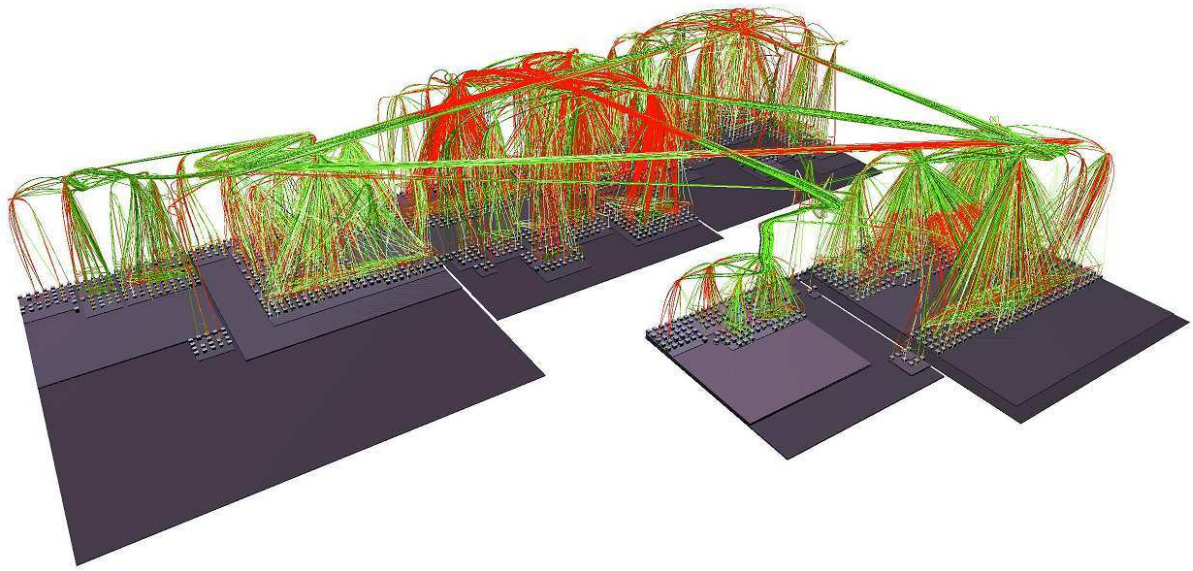
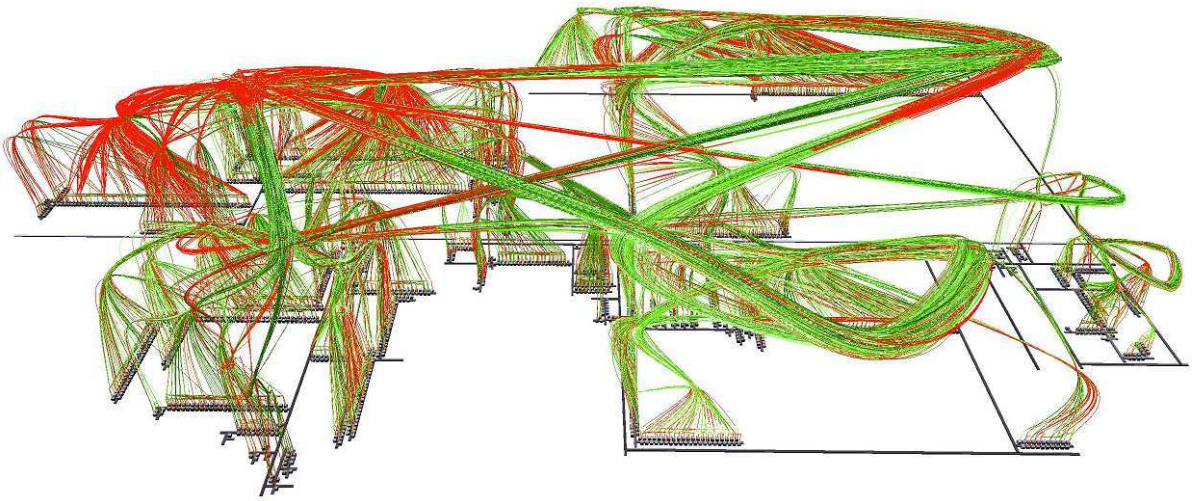


FIGURE 7.9 – Division des relations en quatre sous-ensembles



(a) 3D-HEB avec une métaphore de ville utilisant un placement imbriqué



(b) 3D-HEB avec une métaphore de ville utilisant un placement en rues

FIGURE 7.10 – Visualisation des relations d'appels dynamiques sur une exécution de JEdit, les classes du Java JRE incluses. 2710 classes, 10870 arêtes représentant 4 632 680 relations d'appels. L'important pouvoir d'attraction des 3D-HAPs ($\beta = 0.9$) conduit à une forte courbure des arêtes.

7.5 Résultats

Nous avons vu que le très grand nombre d'interactions entre les éléments du logiciel rend sa représentation sous forme de graphe très difficile et confuse. Dans cette section, nous présentons l'utilisation de notre technique de visualisation sur des logiciels Java réellement utilisés quotidiennement par des utilisateurs. Notre technique 3D-HEB est utilisée avec la visualisation d'une métaphore de la ville et les images présentées dans ce chapitre sont issues de notre outil VITRAIL Visualizer. Il faut noter que notre outil implémente également un curseur qui permet de voir évoluer les relations dynamiquement et que l'utilisateur peut parfaitement naviguer dans l'espace 3D pour explorer plus en détails certaines parties du logiciel.

Dans cette section 7.5.1, nous montrons que notre technique des 3D-HEB est indépendante du placement en proposant deux placements différents pour la ville. Dans la section 7.5.2, nous discutons des différents paramètres qui peuvent être ajustés pour modifier la représentation des relations. Ces paramètres permettent d'améliorer (ou de réduire) la compréhension de la visualisation.

7.5.1 Indépendance par rapport au placement de la ville

Une particularité intéressante de la technique 2D HEB d'Holten [Hol06] est que la représentation des relations est indépendante par rapport à la représentation de la structure du logiciel. Néanmoins, les éléments doivent être hiérarchisés en utilisant par exemple un Treemap, un *Circular Treemap*, *Icicle*, etc. Dans notre cas, cette organisation hiérarchique est indispensable car le niveau d'élévation de chaque 3D-HAP dépend de cette hiérarchie (voir section 7.4.1).

Dans la section 7.5.1.1, nous présentons une vue de notre technique 3D-HEB avec un placement imbriqué de la ville. Ensuite en section 7.5.1.2, nous visualisons le même logiciel avec un placement en rues.

7.5.1.1 3D-HEB avec une ville ayant un placement imbriqué

La figure 7.10(a) montre les relations d'appels dynamiques avec une métaphore de ville utilisant un placement imbriqué. Le facteur $\beta = 0.9$ influence grandement la manière dont les arêtes sont représentées et la figure montre distinctement le fait que les 3D-HAPs attirent les arêtes en leur donnant une courbure particulière. De cette manière les arêtes s'élèvent au dessus de la métaphore de la ville pour finalement se rejoindre et former des groupes de liens.

Notre technique donne du volume à la représentation grâce à la manière dont les arêtes sont représentées. Cela permet de montrer les bâtiments de la ville (classes) tout en affichant un très grand nombre de relations, avec leur direction, et avec une indication de leur nombre d'occurrences. Par exemple, le paquet représenté au milieu au fond de la figure 7.10(a) représente le paquet `java.lang` qui contient toutes les classes de base de Java et le paquet à gauche représente les classes principales de JEdit. On peut remarquer que les relations entre ces deux paquets sont unidirectionnelles et que ce sont effectivement les classes de JEdit qui utilisent les classes du paquet `java.lang`.

Notre technique 3D-HEB appliquée à la métaphore de la ville permet d'avoir une vue d'ensemble du système et de ses relations. De plus, avec la représentation du logiciel par une métaphore de ville, des métriques peuvent être associées à la taille des bâtiments. De cette manière, toute la puissance de la métaphore de la ville pour la visualisation du logiciel est préservée. Afin de permettre une analyse plus détaillée de certaines parties du logiciel, l'utilisateur peut

naviguer librement dans la visualisation et sélectionner les liens sur lesquels il veut poursuivre une inspection détaillée.

7.5.1.2 3D-HEB avec une ville ayant un placement en rues

La figure 7.10(b) visualise le même logiciel et les mêmes relations que dans la figure 7.10(b) mais avec une métaphore de ville utilisant un placement en rues comme expliqué dans la section 7.3.2. Avec ce placement, les 3D-HAPs sont beaucoup plus dispersés dans l'espace 3D et les arêtes doivent parcourir plus de distance pour aller d'un point de la visualisation à un autre. Cette représentation a l'avantage de produire plus d'espace entre les groupes d'arêtes et permet donc de bien distinguer les communications entres les différents paquets.

7.5.2 Influence des paramètres de réglage de la visualisation

Cette section décrit comment le réglage de quelques paramètres peut avoir un impact important sur la visualisation.

La figure 7.11 montre les mêmes données que pour la section 7.5.1, avec un filtre pour visualiser uniquement les relations d'appels dynamiques des classes de JEdit, avec $\beta = 0.8$. Cette figure est utilisée comme référence pour permettre la comparaison avec d'autres figures de cette section.

La section 7.5.2.1 montre l'influence majeure du facteur β sur la représentation des arêtes. Puis la section 7.5.2.2 explique comment certains 3D-HAPs peuvent être retirés du chemin pour simplifier le parcours des arêtes. Enfin en section 7.5.2.3, nous discutons de l'angle de courbure qui est donné aux arêtes lors de l'attraction des 3D-HAPs.

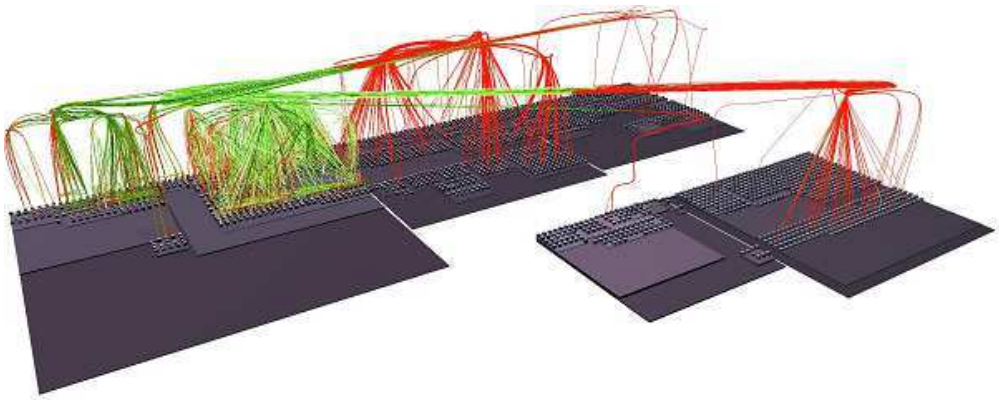


FIGURE 7.11 – Appels dynamique des classes de JEdit. 2710 classes, 2350 arêtes représentant 1 430 347 appels. $\beta = 0.8$. Avec courbure des arêtes.

7.5.2.1 Influence du pouvoir d'attraction sur les arêtes

La figure 7.12 visualise les même relations que la figure 7.11 mais avec $\beta = 0.4$. Ceci produit des arêtes plus droites et plus de croisements et de superpositions d'arêtes. Par rapport à la figure 7.11, il est beaucoup plus difficile de voir les directions des relations. La ville est également considérablement obstruée par les arêtes.

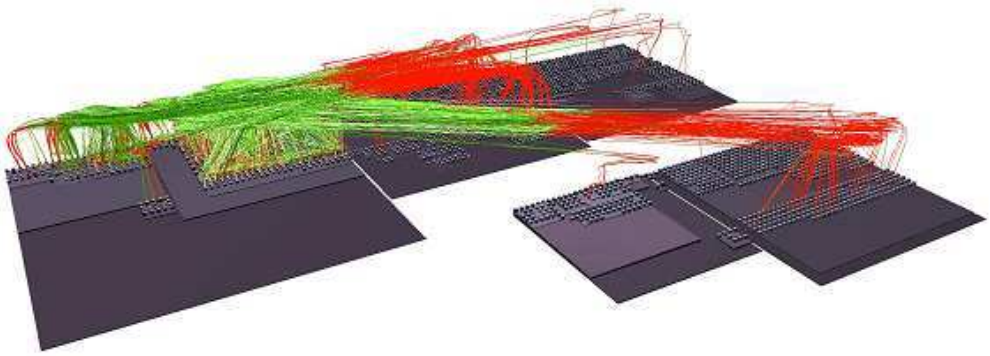


FIGURE 7.12 – $\beta = 0.4$

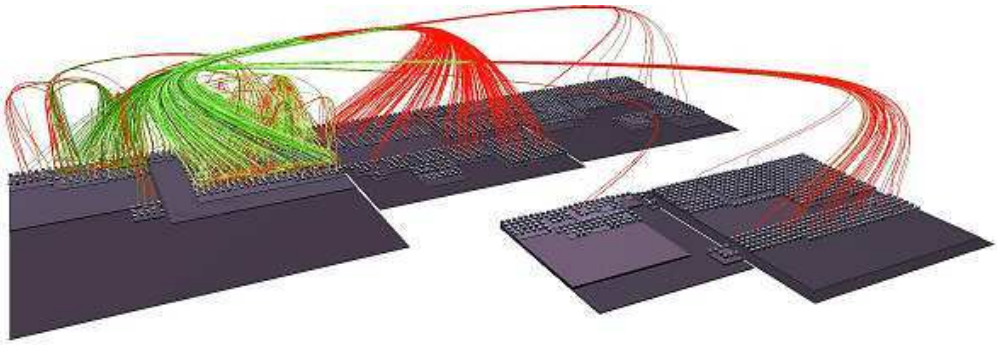


FIGURE 7.13 – Attraction des 3D-HAPs au niveau de la classe et au niveau le plus élevé de chaque chemin.

7.5.2.2 Annulation de l'attraction de certains 3D-HAPs

Avec notre technique de visualisation 3D-HEB, l'attraction de certains 3D-HAPs peut être annulée. Ceci permet d'avoir des représentations d'arêtes plus directes (c'est-à-dire avec moins de courbure). Par exemple, nous pouvons garder le pouvoir d'attraction des 3D-HAPs du niveau classe et les 3D-HAPs du niveau le plus élevé de chaque chemin et annuler l'attraction des autres 3D-HAPs. Le résultat peut être visualisé dans la figure 7.13. Dans la plupart des cas, annuler l'attraction de certains 3D-HAPs permet aux arêtes de faire moins de détours et rend donc les arêtes plus facile à suivre. *In fine* chaque relation comprend seulement deux courbures : une au moment de quitter le chemin $originPath$ et une autre au moment d'entrer dans le chemin $destinationPath$.

7.5.2.3 Arêtes courbées ou à angles vifs

La figure 7.14 montre notre visualisation 3D-HEB avec des arêtes à angles vifs. Aucune courbure, aucun lissage, n'est appliqué aux arêtes. Les arêtes sont toujours attirées par les 3D-HAPs mais elles forment des angles brutaux. Il n'y a pas de différence sémantique entre la version avec courbes et la version avec angles. Dans certaines situations, notamment lorsque β est proche de 0 ou 1, une version peut simplement être plus lisible que l'autre. D'après nos expérimentations, nous avons trouvé que pour un β petit (moins de 0.3) l'utilisation de la version avec angles réduit le nombre de croisements. Au contraire, pour un β élevé (plus de 0,97) les arêtes courbées permettent un regroupement plus efficace des arêtes, ce qui les rend plus faciles

à suivre.

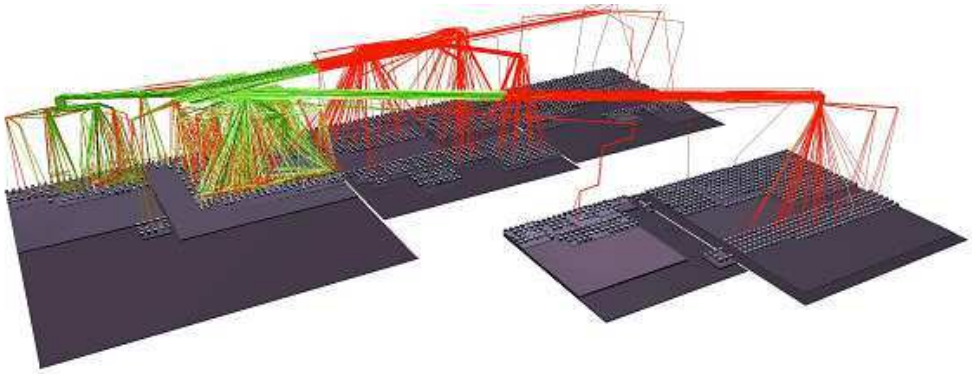


FIGURE 7.14 – Avec des arêtes droites et des angles vifs.

7.6 Conclusion et Perceptives

La visualisation d'un grand système a toujours été un défi, tout particulièrement en ce qui concerne les relations entre ses éléments.

Certains travaux, notamment ceux concernant l'utilisation de métaphores de la ville [WL07b], ont montré des propriétés intéressantes. Les atouts principaux de cette technique de visualisation sont la représentation compacte de la structure du logiciel et la représentation intuitive des métriques du logiciel et ce même pour de très gros systèmes. En revanche, la représentation efficace des relations est un point faible de la métaphore de la ville car peu de travaux de recherche ont apporté un moyen de représenter toutes ces relations de manière compréhensible.

D'un autre côté, la technique des Hierarchical Edge Bundles (HEB) [Hol06] est une technique innovatrice et intéressante qui permet de visualiser les relations du logiciel d'une façon attrayante et compréhensible, mais pas du tout métaphorique (graphe).

La contribution de ce chapitre est donc l'apport d'une nouvelle technique de visualisation qui combine les avantages de la métaphore de la ville et des Hierarchical Edge Bundles.

Nommée 3D Hierarchical Edge Bundles (3D-HEB), notre technique, permet de représenter la structure du logiciel et les métriques grâce à une métaphore de la ville. Les points d'attraction hiérarchiques (décrits en section 7.4) sont placés dans l'espace 3D (3D-HAPs) et positionnés de manière symétrique par rapport à la structure de la ville. Ces 3D-HAPs sont ensuite utilisés pour créer des chemins qui vont servir à diriger les arêtes représentant les relations entre deux éléments du logiciel. Des regroupements d'arêtes sont ainsi formés à différents niveaux d'élévation dans l'espace 3D ce qui produit moins de croisements d'arêtes et moins de recouvrements de la ville, améliorant la lisibilité globale.

Nous proposons deux types de placement différents pour la métaphore de la ville afin d'illustrer l'indépendance de notre technique par rapport au choix de placement de la métaphore de la ville. La manière dont les arêtes sont attirées par les 3D-HAPs est réglable avec le facteur β qui permet donc de configurer la visualisation pour correspondre aux besoins de l'utilisateur. Nos expérimentations ont montrés qu'un facteur $\beta = 0.9$ apporte une meilleure visualisation des relations dans la plupart des cas.

Un autre aspect important de notre visualisation est la représentation de la quantification et la direction de chaque relation. En effet, certains composants sont très liés et il est important de

faire ressortir cet aspect avec la visualisation. De plus, le coté unidirectionnel et bidirectionnel de certains liens apportent des renseignements essentiels à la compréhension des relations à l'exécution.

Enfin nous donnons plusieurs moyens de configurer la visualisation en contrôlant la force d'attraction des 3D-HAPs, la densité de 3D-HAPs ou la forme des arêtes.

Le premier résultat que nous avons présenté dans ce chapitre est la possibilité de visualiser de manière efficace les relations d'un gros logiciel (2700 classes, 4.6 millions d'appels) tout en gardant une visualisation lisible et compréhensible. Notre visualisation du logiciel et de ses relations est donc très complète et capable de représenter utilement beaucoup d'informations ; il en résulte une meilleure compréhension du fonctionnement du logiciel observé. De plus, avec notre technique nous pouvons distinguer les relations à différents niveaux de la hiérarchie. En effet, nous pouvons très distinctement visualiser les communications à tous les niveaux de paquet.

Nos travaux ouvrent de nombreuses perspectives.

Un de nos objectifs principaux est de tester empiriquement la compréhension des relations du logiciel grâce à notre visualisation. Cette expérimentation devrait être faite avec des utilisateurs provenant de différents domaines du logiciel et avec différents niveaux d'expérience professionnelle dans le développement de logiciels. Le but de cette expérimentation serait de mesurer à quel point l'utilisation de notre visualisation permet d'améliorer la compréhension du logiciel.

Un autre objectif serait d'avoir un retour qui permette d'améliorer l'utilisabilité, le rendement et l'efficacité de notre technique 3D-HEB.

Nous souhaitons améliorer le système de navigation 3D en utilisant notamment de nouvelles techniques d'interaction et de visualisation avancées. L'idée est d'amplifier la cognition en utilisant des représentations visuelles interactives bien adaptées aux caractéristiques de la perception humaine. Nous souhaitons par exemple, en première approche, intégrer et utiliser des tableaux blancs interactifs, où l'utilisateur peut contrôler interactivement la visualisation projetée sur les tableaux pour faire toutes sortes d'actions telles que se déplacer, zoomer, dézoomer, etc. L'intérêt principal d'un tel dispositif serait de nous permettre de mieux approcher la visualisation immersive, puisque l'utilisateur aurait un affichage 2m x 1m devant lui, à sa gauche, et à sa droite. Nous souhaitons étudier l'impact de l'utilisation de ce dispositif à base de tableaux interactifs sur la compréhension du logiciel et sur la rapidité d'exécution de tâches de développement.

À plus long terme, nous envisageons de permettre à l'utilisateur de modifier le code source directement grâce à la manipulation de la visualisation. Nous pensons que les outils de visualisation sont tout à fait adaptés pour permettre des actions de re-conception et que ces actions graphiques peuvent être reportées directement dans le code source du logiciel. En effet, les études menées sur Rigi dans les travaux présentés en chapitre 3 section 3.4.1 ont montré que l'utilisation d'une visualisation permettant une modification directe du code source a grandement facilité les tâches de développement et de maintenance.

Troisième partie

Conclusion

8

Conclusion

8.1 Bilan et apports de la thèse

Nous résumons ici nos travaux et leurs résultats. De façon synthétique, nous considérons les contributions suivantes comme importantes dans cette thèse:

- une étude bibliographique de qualité, très complète, du domaine de la visualisation de l'aspect statique du logiciel, qui a donné lieu à un article de survey se voulant la référence dans le domaine [CZ11b].
- la création d'une technique de traçage de l'exécution des programmes, sa validation et son implémentation sous la forme de l'outil VITRAIL JBInsTrace [CZ11a, CZ12].
- la création d'une technique de visualisation de l'exécution du logiciel améliorée basée sur une métaphore de ville avec relations représentées sous forme d'arrêtes courbées et son implémentation sous la forme de l'outil VITRAIL Visualizer [CZ12].

8.1.1 Bibliographique sur la visualisation de l'aspect statique du logiciel

Le travail bibliographique effectué dans le cadre de cette thèse a été considérable, et a porté sur plusieurs domaines. Le plus complètement exploré a été celui de la visualisation de l'aspect statique du logiciel (voir chapitre 3). Cette bibliographie a donné lieu à la rédaction d'un article bibliographique publié dans une revue majeure (et très exigeante) du domaine: *IEEE Transactions on Visualization and Computer Graphics (TVCG)*. Il réunit 190 citations pour vingt pages d'article en double colonne, police neuf [CZ11b]. Il s'agit d'un succès, car notre article est déjà cité par 9 articles du domaine comme point d'entrée dans la littérature. Il remplace en cela un ancien article qui faisait référence [TC09].

8.1.2 Technique et outil d'analyse de l'exécution

Afin d'obtenir les informations dynamiques de l'exécution du logiciel nécessaires pour nos recherches, nous nous sommes tout d'abord intéressés à la conception d'une analyse fine de l'exécution, implantée dans l'outil VITRAIL JBInsTrace qui nous a permis d'extraire des informations difficiles, voir impossible, à obtenir avec les techniques et outils conventionnels (voir chapitre 6).

Une première étape fut la conception d'une nouvelle technique de traçage de logiciel qui apporte un niveau de détail supérieur à l'état de l'art existant (voir 90, chapitre 6, section 6.6) tout en offrant des performances d'exécutions correctes, ie. permettant l'utilisation des logiciels analysés. Pour cela notre technique instrumente dynamiquement toutes les classes du logiciel et

les classes du JRE, au niveau du bloc de base. Ainsi, le traçage se fait non seulement au niveau des appels de méthodes, mais également au niveau du flux d'exécution intra-méthode. Ce niveau de granularité très fin nous permet d'étudier le polymorphisme sur les sites d'appels, les accès aux attributs, les sauts dus aux boucles ou conditionnelles, les accès aux ressources critiques, etc. De plus, ce type d'information n'est pas forcément disponible avec d'autres analyseurs dynamiques.

VITRAIL JBInsTrace a reçu des commentaires très positifs de la part des relecteurs de l'édition spéciale *Experimental Software and Toolkits (EST)* de la revue *Elsevier's Science of Computer Programming* dans laquelle nous avons publié notre technique de traçage [CZ12]. Nous citons un relecteur en traduisant en français: "Je trouve vos recherches très intéressantes et je pense que vous avez une belle plateforme dans vos mains pour continuer à faire de belles recherches dans de domaine de l'analyse dynamique".

Pour résumer, notre outil VITRAIL JBInsTrace nous procure donc une trace très fine de l'exécution des logiciels et nous permet de faire des analyses *post-mortem* détaillées de l'exécution incluant des analyses de typages des sites d'appels, la reconstruction du graphe d'appel à l'exécution, les calculs de nombreuses métriques dynamiques (voir section 8.2).

Nous apportons une nouvelle technique de traçage des programmes qui est très flexible, assez complète et simple d'utilisation. Celle-ci nous permet de recueillir des informations d'une grande importance pour nos travaux de recherches

8.1.3 Visualisation de l'exécution du logiciel

Pour comprendre l'énorme quantité de données qui est récoltée lors de l'analyse dynamique, nous avons développé notre propre visualisation de l'exécution des logiciels nommée: VITRAIL Visualizer (voir chapitre 7). En effet, les visualisations existantes, bien qu'intéressantes, ne permettaient pas une visualisation aussi poussée que nous le souhaitions, notamment au niveau des relations. La trace d'exécution pouvant s'apparenter à un énorme graphe d'appel entre les méthodes, notre problème réside dans la représentation efficace, "ergonomique", de ce graphe.

Certaines techniques telles que la visualisation grâce à l'utilisation d'une *métaphore du monde réel* consistent à représenter le logiciel dans un contexte familier que l'utilisateur reconnait rapidement. La métaphore de la ville [WL07b] consiste à représenter un logiciel par une ville, avec des bâtiments représentant les éléments du logiciel, les quartiers représentent des paquets d'éléments, etc.

Nous avons donc créé une nouvelle technique de visualisation de logiciel qui permet de visualiser (entre autres) les relations entre les composants du logiciel en utilisant une métaphore de ville. L'avantage principal de la métaphore de ville classique est que celle-ci permet une représentation claire de la structure du logiciel et des métriques. Ces métriques sont des valeurs numériques mesurants une ou plusieurs propriétés du logiciel et apportent des informations sur la qualité de la conception du logiciel [WLR11]. Transformer les valeurs numériques des métriques en caractéristiques visuelles permet à l'utilisateur d'avoir une meilleure perception et donc une compréhension plus rapide de l'information.

L'inconvénient de la ville logicielle est que les éléments graphiques reposent tous sur un même plan 3D, ce qui rend difficile la représentation graphique des relations. Pour pallier l'inconvénient d'avoir les éléments sur un même niveau d'élévation, nous avons développé une technique astucieuse qui permet aux arrêtes, qui représentent les appels, de s'élever à différents niveaux d'élévations, en fonction de la structure du logiciel.

Nous nous sommes inspirés de la technique de visualisation 2D appelée "Hierarchical Edge Bundle" [Hol06] que nous avons adaptée à la 3D en l'appelant "3D Hierarchical Edge Bundle".

Notre technique permet aux arêtes d'être regroupées, mais sur plusieurs niveaux d'élévations. De notre point de vue, la visualisation est ainsi plus aérée, aidant ainsi à une meilleure compréhension du point de départ des arrêtes et du point d'arrivée.

Le résultat est surprenant, car toutes les relations d'appels (environ 4,5 millions d'appels avec le logiciel ArgoUML) présentes dans le logiciel à l'exécution sont représentables et la visualisation reste parfaitement lisible. De plus, notre outil permet un filtrage sur les points d'attractions hiérarchiques 3D pour obtenir une représentation simplifiée de toutes les relations, qui donne de très bons résultats.

Enfin, l'implantation d'une animation dans l'outil VITRAIL Visualizer permet la visualisation de l'évolution les relations du logiciel durant son exécution. En effet, cette animation fluide permet de voir évoluer non seulement les relations, mais aussi les métriques associées à la taille et la couleur des bâtiments.

Ces travaux ont été publiés à *VISSOFT*, workshop de référence dans le domaine de la visualisation des logiciels [CZB11].

8.2 Travaux en cours et perspectives

Dans les sous-sections suivantes, nous exposons les perspectives de nos travaux, en commençant par celles déjà entamées, qui devraient déboucher à court terme, puis en expliquant notre vision à moyen terme et long terme.

8.2.1 Comparaison entre analyses statiques et analyses dynamiques

A relativement court terme, nous poursuivons deux grands axes de travail, déjà commencés mais qui n'ont pas fait encore l'objet de publications, dont nous avons les données brutes qui serviront de base à nos analyses et futures publications. Ces deux grands travaux traitent de sujets différents, mais reposent sur la même analyse de l'exécution et des données résultant de VITRAIL JBInsTrace.

Le premier axe de nos travaux de recherches consiste à comparer les métriques résultant d'une analyse statique avec les métriques résultant de plusieurs analyses dynamiques. Les questions que nous nous posons sont les suivantes. Etant donné que l'analyse statique des sites d'appels nous renseigne sur tous les cas d'exécutions possibles, est-ce que les mesures dynamiques de plusieurs scénarios d'exécution sur ces même sites d'appels permettent de s'approcher des mesures statiques? Sinon à quel point se situe la différence entre ces deux aspects?

De plus, nous cherchons à mesurer la distance qu'il existe entre le nombre de types d'instances différentes que prennent les sites d'appels à l'exécution et le nombre de types différents que prennent les sites d'appels avec les analyses C.H.A., R.T.A. et V.T.A.. En effet, l'étude de la présence réelle de polymorphisme dans les programmes Java n'a toujours pas été réalisée, alors que beaucoup de questions traitées en dépendent. Nous aimerions savoir si les sites d'appels sont plutôt monomorphes ou plutôt polymorphes. La réponse à cette question permettrait de mieux comprendre les habitudes de programmation des développeurs Java d'une part, et de faire des optimisations au niveau de la JVM concernant le choix de la méthode ciblée par chaque site d'appel d'autre part.

Le second axe de nos travaux de recherche en cours consiste à comparer les métriques de couplage statique et dynamique pour déterminer laquelle est un bon prédicteur des *changements* dans les logiciels. Nous cherchons à répondre à la question suivante: les métriques de couplage

dynamique calculées sur plusieurs scénarios d'exécutions d'un même programme sont-elles de meilleurs prédicteurs des changements dans les logiciels que les mesures de couplage statique?

En effet, nous faisons l'hypothèse que l'évolution du code source du logiciel est fortement liée à l'exécution du logiciel. Donc si notre analyse est restreinte aux modules qui jouent un rôle lors de l'exécution du logiciel, les mesures de couplage sur cet ensemble pourraient donner des résultats plus précis que les mesures statiques qui sont faites sur l'ensemble du logiciel.

Pour répondre à ces deux problèmes, nous avons normalisé le graphe d'appel dynamique généré à partir de la trace d'exécution avec le graphe d'appel statique généré par l'analyseur Soot [LBLH11]. Ainsi nous calculons des métriques à partir d'un graphe représentant toutes les exécutions possibles (graphe statique) et le graphe résultant de nos analyses dynamiques. Nous nous appuyons sur les travaux de [AVDS10] où les auteurs proposent une méthode pour calculer des métriques de couplage à partir du graphe d'appel. Les graphes statiques et dynamiques ayant la même structure, nous utilisons la même méthode de calcul pour nos métriques statiques et dynamiques. Ceci nous permet de rester cohérents et de calculer exactement la même caractéristique de couplage de manière statique et dynamique. Les analyses (et graphes générés par Soot) correspondent aux analyses classiques dans le domaine de l'analyse statique du code source des logiciels : C.H.A. (*Class Hierarchy Analysis*), R.T.A. (*Rapid Type Analysis*) et V.T.A. (*Variable Type Analysis*) [Sõn09]. Bien évidement, nous devons considérer plusieurs exécutions différentes et fusionner ces graphes d'appels dynamiques pour permettre une comparaison avec le graphe statique.

Nous analysons les jeux de tests des développeurs du logiciel en supposant que ces jeux d'entrée permettent de tester le programme dans son intégralité. Nous avons choisi ArgoUML, car celui-ci a été largement adopté par la communauté des chercheurs. De plus, c'est un programme Java open-source qui est très mature, car développé depuis 2003 et qui compte plus de 150 développeurs.

Nous avons fait tourner nos analyses sur environ dix versions différentes d'ArgoUML sorties entre 2007 et 2012, dont nous avons également analysé le code source (via le dépôt SVN) au complet. Nous sommes donc en possession de toutes les données nécessaires pour nos analyses statistiques que nous avons implantées en Matlab. Ainsi, nous exécutons une analyse statistique descriptive, une analyse de covariance, une analyse en composante principale et une régression linéaire sur les données.

Nous sommes arrivés au point où nous avons toutes les données permettant de répondre aux deux problèmes. Il nous reste à étudier les résultats des mesures statistiques obtenues.

8.2.2 Technique et outil d'analyse de l'exécution

A moyen terme, notre technique d'analyse de l'exécution dynamique de l'exécution pourrait être améliorée, tout comme l'outil VITRAIL JBInsTrace qui l'implante.

Comme dans tout développement, des parties d'implémentation peuvent être modifiées pour obtenir de meilleures performances à l'exécution. Actuellement, nous récupérons l'identifiant du fil d'exécution courant à chaque nouvelle exécution d'un bloc de base ce qui est extrêmement couteux. Une optimisation serait de modifier la classe `Thread` statiquement pour qu'elle appelle le traceur automatiquement à chaque changement de fil d'exécution. Ainsi nous passerions d'un appel par exécution de bloc de base à un appel par changement de fil d'exécution. Le gain peut être significatif, mais nous perdrions l'aspect entièrement dynamique de VITRAIL JBInsTrace.

Comme nous l'avons dit précédemment, l'optimisation de la technique de traçage n'est pas une priorité pour le moment. Nous pensons que l'analyseur d'exécution est suffisamment complet

pour nous permettre de faire des recherches intéressantes sur l'exécution des programmes.

Sur le long terme, nous souhaitons étudier le taux de polymorphisme sur un très grand nombre de logiciels Java. En effet, pour le moment nos recherches se focalisent sur quelques logiciels, mais les résultats obtenus sont-ils généralisables? Pour cela nous avons déjà récupéré le Qualitas Corpus [TAD+10] (le plus gros corpus de logiciel Java à notre connaissance) et souhaitons analyser l'exécution de la plupart des logiciels présents. Un des buts est de déterminer si les sites d'appels des programmes Java sont plutôt monomorphes ou plutôt polymorphes. Ces analyses nous permettraient d'étudier les habitudes de codage des programmeurs Java et également d'apporter des propositions d'optimisation de l'exécution des logiciels.

8.2.3 Visualisation de l'exécution du logiciel

De la même manière, nous avons identifié plusieurs perspectives d'amélioration de la technique de visualisation du logiciel, à moyen terme.

Une perspective de nos travaux de visualisation serait de valider empiriquement notre visualisation des relations du logiciel en utilisant une métaphore de ville. La preuve empirique de [WLR11] sur la métaphore de la ville a montré de très bons résultats avec une amélioration de l'exactitude des tâches réalisées sur le logiciel (+24%) et le temps pour réaliser ces tâches (-12%), par rapport à l'utilisation de Eclipse pour explorer le code et Excel pour explorer les métriques. Néanmoins, notre expérimentation empirique s'intéresserait particulièrement à tester l'efficacité de notre représentation des relations grâce au "3D Hierarchical Edge Bundle". Celle-ci reste donc une perspective intéressante.

Sur le long terme, nous souhaitons également améliorer le système de navigation en utilisant notamment des nouvelles techniques de visualisation avancée. Celles-ci visent à représenter efficacement des données abstraites. Son principe de base est d'amplifier la cognition en utilisant des représentations visuelles interactives bien adaptées aux caractéristiques de la perception humaine. Par exemple, l'utilisation d'un système où la visualisation du logiciel engloberait le développeur (grands écrans disposés en forme de boite, casque, etc) , reconstituant une réalité virtuelle en 3D, permettrait une plus ou moins grande immersion dans l'image interactive et offrirait des possibilités de visualisation et d'interaction sans équivalent.

L'utilisation de dispositifs d'interaction avancés serait également souhaitable, comme par exemple des tableaux blancs interactifs, ou des systèmes de suivi de mouvement (Kinect [OKA11] ou apparentés), nous aiderait également à explorer une nouvelle dimension que nous n'avons pas pu aborder pour l'instant par faute de temps. Il est en effet crucial que l'utilisateur (développeur informatique) puisse aisément et efficacement faire toutes sortes d'actions telles que se déplacer, zoomer, dézoomer, annoter etc. directement à partir de la visualisation. Nous souhaitons donc appliquer ces nouvelles technologies à la visualisation du logiciel pour étudier leurs impacts sur la compréhension du logiciel et sur la rapidité d'exécution de certaines tâches de développement.

Bibliographie

[AAD+08] H. Abdeen, I. Alloui, S. Ducasse, D. Pollet, M. Suen, and I.L.N. Europe. Package Reference Fingerprint: a Rich and Compact Visualization to Understand Package Relationships. In *of 12th Euro. Conf. on Soft. Maintenance and Reeng.*, 2008.

[ABF04] E. Arisholm, L.C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *Software Engineering, IEEE Transactions on*, 30(8):491–506, 2004.

[AC94a] F.B. Abreu and R. Carapuça. Candidate Metrics for Object-Oriented Software within a Taxonomy Framework. *Journal of Systems and Software*, 26:87–96, 1994.

[AC94b] Fernando Brito Abreu and Rogério Carapuça. Object-Oriented Software Engineering: Measuring and Controlling the Development Process. Proc. 4th International Conference on Software Quality, 1994.

[AD07] S. Alam and P. Dugerdil. Evospaces Visualization Tool: Exploring Software Architecture in 3D. *14th Conf. on Rev. Eng.*, 2007.

[AE05] A. Ahmed and P. Eades. Automatic Camera Path Generation for Graph Navigation in 3D. In *ACM Int'l Conference Proc. Series*, pages 27–32. Australian Computer Society, Inc., 2005.

[AG00] V. Ambriola and V. Gervasi. Process Metrics for Requirements Analysis. *Lecture notes in computer science*, pages 90–95, 2000.

[AGE95] F.B. Abreu, M. Goulão, and R. Esteves. Toward the Design Quality Evaluation of Object-Oriented Software Systems. In *5th Int'l Conf. on Soft. Quality*, pages 44–57, 1995.

[AGJ83] A.J. Albrecht and J.E. Gaffney Jr. Software function, source lines of code, and development effort prediction: a software science validation. *Software Engineering, IEEE Transactions on*, 9(6):639–648, 1983.

[AH98] K. Andrews and H. Heidegger. Information Slices: Visualising and Exploring Large Hierarchies using Cascading, Semi-Circular Discs. In *IEEE Infovis Late Breaking Hot Topics*, pages 9–11, 1998.

[AK99] EB Allen and TM Khoshgoftaar. Measuring Coupling and Cohesion: An Information-Theory Approach. In *Proc. 6th International Software Metrics Symp. (METRICS'99)*, pages 119–127, 1999.

[AKSS03] A. Abran, A. Khelifi, W. Suryn, and A. Seffah. Usability Meanings and Interpretations in ISO Standards. *Software Quality Journal*, 11:325–338, 2003.

[AM96] Fernando Brito Abreu and Walclio Melo. Evaluating the Impact of Object-Oriented Design on Software Quality. In *Proc. 3rd International Software Metrics Symp.*, pages 90–99, 1996.

[AP07] S. Alam and D. Ph. EvoSpaces: 3D Visualization of Software Architecture. In *Int'l Conf. on Soft. Eng. and Knowledge Eng.*, 2007.

[Ari02] Erik Arisholm. Dynamic coupling measures for object-oriented software. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS '02, pages 33–, Washington, DC, USA, 2002. IEEE Computer Society.

[ASJ01] E. Arisholm, D.I.K. Sjøberg, and M. Jørgensen. Assessing the changeability of two object-oriented design alternatives–a controlled experiment. *Empirical Software Engineering*, 6(3):231–277, 2001.

[AVDS10] Simon Allier, Stéphane Vaucher, Bruno Dufour, and Houari A. Sahraoui. Deriving coupling metrics from call graphs. In *SCAM*, pages 43–52, 2010.

[AZ08] Serge Demeyer Andy Zaidman. Automatic identification of key classes in a software system using webmining techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6):387–417, 2008.

[Bal99] T. Ball. The concept of dynamic analysis. In *Software Engineering-ESEC/FSE'99*. Springer, 1999.

[BBM96] VR Basili, LC Briand, and WL Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Trans. Soft. Eng.*, 22(10):751–761, 1996.

[BCK05] T. Bladh, D.A. Carr, and M. Kljun. The Effect of Animated Transitions on User Navigation in 3D Treemaps. *Info. Vis.*, 2005.

[BCS04] T. Bladh, D.A. Carr, and J. Scholl. Extending Tree-Maps to Three Dimensions: A Comparative Study. *Lecture notes in Computer Science*, pages 50–59, 2004.

[BCW01] TR Browning, L.M.A. Co, and F. Worth. Applying the Design Structure Matrix to System Decomposition and Integration Problems: a Review and New Directions. *IEEE Trans. on Eng. Management*, 48(3):292–306, 2001.

[BD97] J. Bansiya and C. Davis. Automated Metrics and Object-Oriented Development. *Dr. Dobbs Journal*, 1997.

[BD02] J. Bansiya and CG Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Trans. Software Engineering*, 28(1):4–17, 2002.

[BD04] M. Balzer and O. Deussen. Hierarchy Based 3D Visualization of Large Software Structures. In *IEEE Conf. on Vis.*, 2004.

[BD07] M. Balzer and O. Deussen. Level-of-Detail Visualization of Clustered Graph Layouts. In *Asia-Pacific Symp. on Vis.*, 2007.

[BDJ99] L.C. Briand, J.W. Daly, and Wüst J.K. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Trans. Software Engineering*, 25(1):91–121, 1999.

[BDL05] M. Balzer, O. Deussen, and C. Lewerentz. Voronoi Treemaps for the Visualization of Software Metrics. In *ACM Symp. on Soft. Vis.*, pages 165–172, 2005.

[BDLP08] A. Bergel, S. Ducasse, J. Laval, and R. Peirs. Enhanced dependency structure matrix for moose. In *2nd Workshop on FAMIX and Moose in Reeng.*, 2008.

[BDR96] L.C. Briand, C.M. Differding, and H.D. Rombach. Practical Guidelines for Measurement-Based Process Improvement. *Software Process Improvement and Practice*, 2(4):253–280, 1996.

[BDW98] L.C. Briand, J.W. Daly, and J. Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[BE96] T. Ball and SG Eick. Software Visualization in the Large. *Computer*, 29(4):33–43, 1996.

[Ben08] SPECjvm2008 Benchmarks. Standard performance evaluation corporation. http://www.spec.org/jvm2008/, 2008.

[BG07] S. Boccuzzo and H. Gall. Cocoviz: Towards Cognitive Software Visualizations. In *4th IEEE Int'l Workshop on Vis. Soft. for Understanding & Analysis*, pages 72–79, 2007.

[BG08] S. Boccuzzo and H.C. Gall. Software visualization with audio supported cognitive glyphs. In *Int'l Conf. on Soft. Maintenance*, 2008.

[BH83] VR Basili and DH Hutchens. An Empirical Study of a Syntactic Complexity Family. *IEEE Trans. Software Engineering*, pages 664–672, 1983.

[BHM07] W. Binder, J. Hulaas, and P. Moret. Advanced Java bytecode instrumentation. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 135–144. ACM, 2007.

[Bie87] I. Biederman. Recognition-by-Components: A Theory of Human Image Understanding. *Psych. review*, 94, 1987.

[BK01] S. Bassil and R.K. Keller. Software Visualization Tools: Survey and Analysis. In *IEEE Int'l Workshop on Prog. Comprehension*, 2001.

[BLC02] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 2002.

[BM99] S. Benlarbi and W.L. Melo. Polymorphism Measures for Early Risk Prediction. In *Proc. 21st IEEE International Conference on Software Engineering*, pages 334–344, 1999.

[BMB94] L.C. Briand, S. Morasca, and V.R. Basili. *Defining and Validating High-Level Design Metrics*. Univ. of Maryland, 1994.

[BME+07] Grady Booch, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. *Object-Oriented Analysis and Design with Applications (3rd Ed.)*. Addison-Wesley Prof., 2007.

[BMMIM98] W.J. Brown, R.C. Malveau, H.W. McCormick III, and T.J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., 1998.

[BMW02] LC Briand, WL Melo, and J. Wust. Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects. *IEEE Trans. Software Engineering*, 28(7):706–720, 2002.

[BN01] T. Barlow and P. Neville. A Comparison of 2D Visualizations of Hierarchies. In *IEEE Symp. on Info. Vis.*, pages 131–138, 2001.

[BNDL04] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz. Software Landscapes: Visualizing the Structure of Large Software Systems. In *Joint Eurographics and IEEE TCVG Symp. on Vis.*, 2004.

[BPK03] R.J. Bril, A. Postma, and R.L. Krikhaar. Embedding Architectural Support in Industry. In *Int'l Conf. on Soft. Maintenance*, 2003.

[Bre07] Kadhim M. Breesam. Metrics for Object-Oriented Design Focusing on Class Inheritance Metrics. In *Proc. 2nd IEEE International Conference on Dependability of Computer Systems*, pages 231–237, 2007.

[Bro93] I. Brooks. Object-Oriented Metrics Collection and Evaluation with a Software Process. In *Workshop on Processes and Metrics for OO Softw. Dev.*, 1993.

[Bru04] D.L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation.* PhD thesis, Massachusetts Institute of Technology, 2004.

[Bru07] E. Bruneton. Asm 3.0 a java bytecode engineering library. *URL: http://download. forge. objectweb. org/asm/asmguide. pdf*, 2007.

[BS90] R. Baecker and I. Small. Animation at the Interface. *The Art of Human-Computer Interface Design*, pages 251–267, 1990.

[BT06] Heorhiy Byelas and Alexandru Telea. Visualization of Areas Of Interest in Software Architecture Diagrams. In *Symp. on Soft. Vis.*, 2006.

[BT09] H. Byelas and A. Telea. Visualizing Metrics on Areas of Interest in Software Architecture Diagrams. In *Pacific Vis. Symp.*, 2009.

[BW02] L.C. Briand and J. Wust. Empirical Studies of Quality Models in Object-Oriented Systems. *Advances in Computers*, 56:98–167, 2002.

[BWDVP00] L.C. Briand, J. Wüst, J.W. Daly, and D. Victor Porter. Exploring the Relationships Between Design Measures and Software Quality in Object-Oriented Systems. *The Journal of Systems & Software*, 51(3):245–273, 2000.

[Car08] Yaser Carpendale, Sheelagh Ghanam. A Survey Paper on Software Architecture Visualization. Technical report, Univ. of Calgary, 2008.

[CAT07] F. Chevalier, D. Auber, and A. Telea. Structural Analysis and Visualization of C++ Code Evolution Using Syntax Trees. In *Ninth Int'l Workshop on Principles of Soft. Evolution.* ACM, 2007.

[CE03] K. Casey and C. Exton. A Java 3D Implementation of a Geon Based Visualisation Tool for UML. In *2nd Int'l Conf. on Principles and Practice of Prog. in Java*, pages 63–65, 2003.

[CG90] D.N. Card and R.L. Glass. *Measuring Software Design Quality.* Prentice-Hall, Inc., 1990.

[Che04] C. Chen. *Information Visualization: Beyond the Horizon.* Springer, Inc., 2004.

[Chi04] S. Chiba. Javassist: Java bytecode engineering made simple. *Java Developer's Journal*, 9(1), 2004.

[Chu98] MC Chuah. Dynamic Aggregation with Circular Visual Designs. In *IEEE Symp. on Info. Vis.*, pages 35–43, 1998.

[CHZ+07] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. van Wijk, and Arie van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 49–58, Washington, DC, USA, 2007. IEEE Computer Society.

[CIK03] N. Churcher, W. Irwin, and R. Kriz. Visualising Class Cohesion with Virtual Worlds. In *Asia-Pacific Symp. on Info. Vis.*, 2003.

[CK91] Shyam R. Chidamber and Chris F. Kemerer. Towards a Metrics Suite for Object Oriented Design. In *OOPSLA*, pages 197–211, 1991.

[CK94] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Software Engineering*, 20(6):476–493, 1994.

[CKI99] N. Churcher, L. Keown, and W. Irwin. Virtual Worlds for Software Visualisation. In *Workshop on Soft. Vis.*, 1999.

[CKK01] E.S. Cho, M.S. Kim, and S.D. Kim. Component Metrics to Measure Component Quality. In *8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, pages 419–426, 2001.

[CKN+03] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A System for Graph-Based Visualization of the Evolution of Software. In *Symp. on Soft. Vis.* ACM, 2003.

[CKTM02] S.M. Charters, C. Knight, N. Thomas, and M. Munro. Visualisation for Informed Decision Making; from Code to Components. In *14th Int'l Conf. on Soft. Eng. and Knowledge Eng.*, 2002.

[CMS99] S.K. Card, J.D. Mackinlay, and B. Shneiderman. *Readings in Information Visualization: using Vision to Think*. Morgan Kaufmann, 1999.

[CMS03] A. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. *Static Analysis*, pages 1076–1076, 2003.

[CS00] M. Cartwright and M. Shepperd. An Empirical Investigation of an Object-Oriented Software System. *IEEE Trans. Software Engineering*, 26(8):786–796, 2000.

[CVF11] M.V. Couto, M.T. Valente, and E. Figueiredo. Extracting software product lines: A case study using conditional compilation. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 191–200. IEEE, 2011.

[CZ11a] P. Caserta and O. Zendra. A Convenient, Transparent, Comprehensive and Efficient Technique to Analyze Java Runtime using Dynamic Bytecode Instrumentation. In *6th workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2011.

[CZ11b] P. Caserta and O. Zendra. Visualization of the static aspects of software: a survey. *IEEE Trans. on Vis. & Computer Graphics*, 17(7):913–933, 2011.

[CZ12] Pierre Caserta and Olivier Zendra. "JBInsTrace: A tracer of Java and JRE classes at basic-block granularity by dynamically instrumenting bytecode". *Science of Computer Programming*, (0):–, 2012.

[CZB11] Pierre Caserta, Olivier Zendra, and Damien Bodénès. 3D Hierarchical Edge Bundles to Visualize Relations in a Software City Metaphor. In *6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2011)*, pages 1–8, Williamsburg, États-Unis, September 2011.

[CZH+08] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *J. Syst. Softw.*, 81:2252–2268, December 2008.

[CZvD+09] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *Transactions on Software Engineering*, 35(5):684–702, 2009.

[D+01] M. Dahm et al. Byte code engineering with the bcel api. *Java Informationstage*, 99:267–277, 2001.

[DBETT98] G. Di Battista, P. Eades, R. Tamassia, and I.G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1998.

[DDHV03] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, 2003.

[DDL99] S. Demeyer, S. Ducasse, and M. Lanza. A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualisation. In *6th Working Conf. on Rev. Eng.*, pages 175–186, 1999.

[DF98] A. Dieberger and A.U. Frank. A City Metaphor to Support Navigation in Complex Information Spaces. *J. of Visual Languages & Computing*, 9(6):597–622, 1998.

[DGN05] S. Ducasse, T. Gîrba, and O. Nierstrasz. Moose: an Agile Reengineering Environment. In *10th Euro. Soft. Eng. Conf. Held Jointly with 13th ACM Int'l Symp. on Foundations of Soft. Eng.*, 2005.

[Die93] A. Dieberger. The Information City - a Step Towards Merging of Hypertext and Virtual Reality. *Conf. on Hypertext*, 93, 1993.

[Die94] A. Dieberger. *Navigation in Textual Virtual Environments Using a City Metaphor*. PhD thesis, Vienna Univ. of Tech., 1994.

[Die07] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer Verlag, Inc., 2007.

[DL05] S. Ducasse and M. Lanza. The Class Blueprint: Visually Supporting the Understanding of Classes. *IEEE Trans. on Soft. Eng.*, 31(1):75–90, 2005.

[DL06] M. D'Ambros and M. Lanza. Reverse engineering with logical coupling. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 189–198. IEEE, 2006.

[DLT01] S. Ducasse, M. Lanza, and S. Tichelaar. The Moose Reengineering Environment. *Smalltalk Chronicles*, 3(2), 2001.

[DOL02] M. Denford, T. O'Neill, and J. Leaney. Architecture-Based Visualisation of Computer Based Systems. In *IEEE Int'l Conf. on Eng. of Computer-Based Systems*, volume 2, 2002.

[DSGA+00] C.R. Dos Santos, P. Gros, P. Abel, D. Loisel, N. Trichaud, JP Paris, C.F. Telecom, and S. Antipolis. Mapping Information onto 3D Virtual Worlds. In *Int'l Conf. on Info. Vis.*, pages 19–21, 2000.

[DSP08] Karim Dhambri, Houari A. Sahraoui, and Pierre Poulin. Visual Detection of Design Anomalies. In *12th Euro. Conf. on Soft. Maintenance and Reeng.*, pages 279–283, 2008.

[DvZH02] M. Dahm, J. van Zyl, and E. Haase. The bytecode engineering library (BCEL), 2002.

[eAC98] F.B. e Abreu and J.S. Cuche. Collecting and Analyzing the MOOD2 Metrics. In *Proc. Workshop on Object Oriented Technology*, page 258. Springer, 1998.

[EBD99] L. Etzkorn, J. Bansiya, and C. Davis. Design and Code Complexity Metrics for OO Classes. *Journal of Object Oriented Programming*, 12:35–40, 1999.

[EGK+02] S.G. Eick, T.L. Graves, A.F. Karr, A. Mockus, and P. Schuster. Visualizing Software Changes. *IEEE Trans. on Soft. Eng.*, 2002.

[Eig03] M. Eiglsperger. *Automatic Layout of UML Class Diagrams: a Topology-Shape-Metrics Approach*. PhD thesis, Universität Tübingen, 2003.

[EKS92] J. Eder, G. Kappel, and M. Schrefl. Coupling and Cohesion in Object-Oriented Systems. In *Conference on Information and Knowledge Management*, volume 6, 1992.

[ESSJ92] SC Eick, JL Steffen, and EE Sumner Jr. Seesoft - a Tool for Visualizing Line Oriented Software Statistics. *IEEE Trans. Soft. Eng.*, 18(11):957–968, 1992.

[FBK06] A. Fronk, A. Bruckhoff, and M. Kern. 3D Visualisation of Code Structures in Java Software Systems. In *Symp. on Soft. Vis.*, 2006.

[Fen91] N.E. Fenton. *Software Metrics: a Rigorous Approach*. Chapman & Hall, Ltd. London, UK, 1991.

[Few04] S. Few. *Show Me the Numbers: Designing Tables and Graphs to Enlighten*. Analytics Press, 2004.

[FG06] M. Fischer and H. Gall. Evograph: A Lightweight Approach to Evolutionary and Structural Analysis of Large Software Systems. In *13th Conf. on Rev. Eng.*, pages 179–188, 2006.

[FN00] Norman E. Fenton and Martin Neil. Software Metrics: Roadmap. In *Proc. ACM Conference on The Future of Software Engineering (ICSE '00)*, pages 357–370, 2000.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[FP97] N. Fenton and S.L. Pfleeger. *Software Metrics: a Rigorous and Practical Approach*. PWS Publishing Co., 1997.

[FT96] W. Frakes and C. Terry. Software Reuse: Metrics and Models. *ACM Computing Surveys*, 28(2):415–435, 1996.

[Fur86] GW Furnas. Generalized Fisheye Views. *ACM SIGCHI Bulletin*, 17(4):16–23, 1986.

[FWD+03] J.D. Fekete, D. Wang, N. Dang, A. Aris, and C. Plaisant. Overlaying Graph Links on Treemaps. In *IEEE Symp. on Info. Vis. Conf. Compendium*, 2003.

[GHM05] K. Gallagher, A. Hatch, and M. Munro. A Framework for Software Architecture Visualisation Assessment. In *3rd IEEE Int'l Workshop on Vis. Soft. for Understanding and Analysis*, 2005.

[GHM08] Keith Gallagher, Andrew Hatch, and Malcolm Munro. Software Architecture Visualization: An Evaluation Framework and Its Application. *IEEE Trans. Soft. Eng.*, 34(2):260–270, 2008.

[GJK+03] C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel. A New Approach for Visualizing UML Class Diagrams. In *ACM Symp. on Soft. Vis.*, pages 179–188, 2003.

[GJR99] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *Int'l Conf. on Soft. Maintenance*, pages 99–108, 1999.

[GK98] J. Gil and S. Kent. Three Dimensional Software Modelling. In *20th IEEE Int'l Conf. on Soft. Eng.*, pages 105–114, 1998.

[GKSD05] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How Developers Drive Software Evolution. In *8th Int'l Workshop on Principles of Soft. Evolution*, pages 113–122, 2005.

[GLW05] O. Greevy, M. Lanza, and C. Wysseier. Visualizing feature interaction in 3-d. In *Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005. 3rd IEEE International Workshop on*, pages 1–6. IEEE, 2005.

[GME05] D. Gracanin, K. Matkovic, and M. Eltoweissy. Software Visualization. *Innovations in Syst. & Soft. Eng.*, 1(2), 2005.

[GR93] Mark Green and Jun Rekimoto. The Information Cube: Using Transparency in 3D Information Visualization. In *3rd Workshop on Info. Tech. & Syst.*, pages 125–132, 1993.

[Gra92] R.B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall, 1992.

[GRR99] M. Gogolla, O. Radfelder, and M. Richters. Towards Three-Dimensional Representation and Animation of UML Diagrams. *Lecture Notes in Computer Science*, pages 489–502, 1999.

[GYB04] H. Graham, H.Y. Yang, and R. Berrigan. A Solar System Metaphor for 3D Visualisation of Object-Oriented Software Metrics. In *Australasian Symp. on Info. Vis.*, volume 35, pages 53–59, 2004.

[Hat04] A. Hatch. *Software Architecture Visualisation*. PhD thesis, Univ. of Durham, 2004.

[HCN98] Rachel Harrison, Steve J. Counsell, and Reuben V. Nithi. An Evaluation of the MOOD Set of Object-Oriented Software Metrics. *IEEE Trans. Software Engineering*, 24(6):491–496, 1998.

[HCvW07a] D. Holten, B. Cornelissen, and J.J. van Wijk. Trace visualization using hierarchical edge bundles and massive sequence views. In *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pages 47 –54, june 2007.

[HCVW07b] D. Holten, B. Cornelissen, and J.J. Van Wijk. Trace visualization using hierarchical edge bundles and massive sequence views. In *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pages 47–54. IEEE, 2007.

[HDH04] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the presence of dynamic class loading. *ECOOP 2004–Object-Oriented Programming*, pages 96–122, 2004.

[HDM98] I. Herman, M. Delest, and G. Melancon. Tree Visualisation and Navigation Clues for Information Visualisation. In *Computer Graphics Forum*, volume 17, pages 153–165, 1998.

[HE98] CG Healey and JT Enns. Building Perceptual Textures To Visualize Multidimensional Datasets. In *Vis.*, pages 111–118, 1998.

[HJK+08] A. Hindle, Z.M. Jiang, W. Koleilat, M.W. Godfrey, and R.C. Holt. Yarn: Animating Software Evolution. In *IEEE Int'l Workshop on Vis. Soft. for Understanding and Analysis*, pages 25–26, 2008.

[HM95] M. Hitz and B. Montazeri. Measuring Coupling and Cohesion in Object-Oriented Systems. In *Proc. International Symp. on Applied Corporate Computing*, 1995.

[HMM00] I. Herman, M.S. Marshall, and G Melançon. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Trans. on Vis. & Computer Graphics*, 2000.

[Hol06] D. Holten. Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *Trans. on Vis. & Computer Graphics*, pages 741–748, 2006.

[HPM08] B. Hoffmann, J. Pérez, and T. Mens. A case study for program refactoring. In *Proc. of the GraBaTS Tool Context*, 2008.

[HS95] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1995.

[HVvW05] D. Holten, R. Vliegen, and JJ van Wijk. Visual Realism for the Visualization of Software Metrics. In *3rd IEEE Int'l Workshop on Vis. Soft. for Understanding and Analysis*, pages 1–6, 2005.

[HvW08] D. Holten and J.J. van Wijk. Visual Comparison of Hierarchically Organized Data. In *Computer Graphics Forum*, volume 27, pages 759–766. Blackwell Publishing Ltd, 2008.

[HWH97] AJ Hanson, EA Wernert, and SB Hughes. Constrained Navigation Environments. In *Scientific Vis. Conf.*, pages 95–95, 1997.

[IC03] W. Irwin and N. Churcher. Object-Oriented Metrics: Precision Tools and Configurable Visualisations. In *9th IEEE Int'l Soft. Metrics Symp.*, pages 112–123, 2003.

[ISS06] Pourang Irani, Dean Slonowsky, and Peer Shajahan. Human Perception of Structure in Shaded Space-Filling Visualizations. *Info. Vis.*, 5(1):47–61, 2006.

[ITW01] P. Irani, M. Tingley, and C. Ware. Using Perceptual Syntax to Enhance Semantic Content in Diagrams. *IEEE Computer Graphics & App.*, 21(5):76–84, 2001.

[IW00] P. Irani and C. Ware. Diagrams Based on Structural Object Perception. In *Working Conf. on Advanced Visual Interfaces*, 2000.

[IW03] P. Irani and C. Ware. Diagramming Information Structures using 3D Perceptual Primitives. *ACM Trans. on Computer-Human Interaction*, 10(1):1–19, 2003.

[JKC04] H.W. Jung, S.G. Kim, and C.S. Chung. Measuring Software Product Quality: a Survey of ISO/IEC 9126. *IEEE Software*, 21(5):88–92, 2004.

[JS91] B. Johnson and B. Shneiderman. Tree-Maps: a Space-Filling Approach to the Visualization of Hierarchical Information Structures. In *IEEE Conf. on Vis.*, pages 284–291, 1991.

[Kan02] S.H. Kan. *Metrics and Models in Software Quality Engineering.* Addison-Wesley Longman Publishing Co., 2002.

[KEM07] A. Kerren, A. Ebert, and J. Meyer. *Human-Centered Visualization Environments.* Springer-Verlag, 2007.

[KJ09] Andreas Kerren and Ilir Jusufi. Novel Visual Representations for Software Metrics Using 3D and Animation. In *Soft. Eng., Lecture Notes in Info.*, 2009.

[Kla03] Hilda B. Klasky. *A Study of Software Metrics.* PhD thesis, Univ. of New Jersey, 2003.

[KM99] C. Knight and M. Munro. Comprehension with[in] Virtual Environment Visualisations. In *7th Int'l Workshop on Prog. Comprehension*, pages 4–11, 1999.

[KM00] C. Knight and M. Munro. Virtual But Visible Software. In *4th IEEE Int'l Conf. on Info. Vis.*, pages 198–205, 2000.

[KM08] H. Kienle and H. Muller. The Rigi Reverse Engineering Environment. In *Int'l Workshop on Adv. Soft. Dev. Tools & Tech.*, 2008.

[Kni01] C. Knight. System and Software Visualisation. *Handbook of Soft. Eng. and Knowledge Eng.*, 4, 2001.

[Kos03] R. Koschke. Software Visualization in Software Maintenance, Reverse Engineering, and Re-Engineering: A Research Survey. *J. of Soft. Maintenance & Evo.: Research & Practice*, 15, 2003.

[Kul07] E. Kuleshov. Using ASM framework to implement common bytecode transformation patterns. *Proc. of the 6th AOSD, ACM Press*, 2007.

[KvdWVW01] E. Kleiberg, H. van de Wetering, and JJ Van Wijk. Botanical Visualization of Huge Hierarchies. In *Symp. on Info. Vis.*, 2001.

[Lan01] M. Lanza. The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques. In *4th Int'l Workshop on Principles of Soft. Evo.*, pages 37–42, 2001.

[Lan03a] M. Lanza. CodeCrawler - Lessons Learned in Building a Software Visualization Tool. In *IEEE Euro. Conf. on Soft. Maintenance and Reeng.*, 2003.

[Lan03b] Michele Lanza. *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization.* PhD thesis, Univ. of Bern, 2003.

[LBLH11] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[LCM+04] F. Losavio, L. Chirinos, A. Matteo, N. Levy, and A. Ramdane-Cherif. ISO Quality Standards for Measuring Architectures. *The Journal of Systems & Software*, 72(2):209–223, 2004.

[LD01] M. Lanza and S. Ducasse. A Categorization of Classes Based on the Visualization of Their Internal Structure: the Class Blueprint. *ACM SIGPLAN Notices*, 36:300–311, 2001.

[LD02] M. Lanza and S. Ducasse. Understanding Software Evolution Using a Combination of Software Visualization and Software Metrics. In *Langages et Modeles Objets*, pages 135–149, 2002.

[LD03] M. Lanza and S. Ducasse. Polymetric Views - a Lightweight Visual Approach to Reverse Engineering. *IEEE Trans. on Soft. Eng.*, 29:782–795, 2003.

[LD07] G. Langelier and K. Dhambri. Visual analysis of azureus using verso. In *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pages 163–164. IEEE, 2007.

[LDDB09] Jannik Laval, Simon Denier, Stéphane Ducasse, and Alexandre Bergel. Identifying Cycle Causes with Enriched Dependency Structural Matrix. In *Conf. on Rev. Eng.*, pages 113–122, 2009.

[LK94] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: a Practical Guide*. Prentice-Hall, Inc., 1994.

[LM06] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer-Verlag New York Inc, 2006.

[LN03] C. Lewerentz and A. Noack. CrocoCosmos - 3D Visualization of Large Object-Oriented Programs. *Graph Drawing Software. Springer-Verlag*, 2003.

[LNH07] V. Lakshmi Narasimhan and B. Hendradjaya. Some Theoretical Considerations for a Suite of Metrics for the Integration of Software Components. *Information Sciences*, 177(3):844–864, 2007.

[LP05] W. Lowe and T. Panas. Rapid Construction of Software Comprehension Tools. *J. of Soft. Eng. & Knowledge Eng.*, 15, 2005.

[LR96] J. Lamping and R. Rao. The Hyperbolic Browser: a Focus + Context Technique for VisualizingLarge Hierarchies. *J. of Visual Languages and Computing*, 7:33–35, 1996.

[LSP05] G. Langelier, H. Sahraoui, and P. Poulin. Visualization-Based Analysis of Quality for Large-Scale Software Systems. In *20th IEEE/ACM Int'l Conf. on Automated Soft. Eng.*, pages 214–223, 2005.

[LSP08] Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Exploring the Evolution of Software Quality with Animated Visualization. In *IEEE Symp. on Visual Lang. and Human-Centric Comp.*, 2008.

[LY99] T. Lindholm and F. Yellin. *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[Lyn60] K. Lynch. *The Image of the City*. MIT press, 1960.

[Mac86] J. Mackinlay. Automating the Design of Graphical Presentations of Relational Information. *Trans. On Graphics*, 5(2), 1986.

[Mar82] D. Marr. *Vision: A Computational Investigation Into the Human Representation and Processing of Visual Information*. Henry Holt and Co., 1982.

[Mar04] R. Marinescu. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In *Proc. 20th IEEE International Conference on Software Maintenance*, pages 350–359, 2004.

[MBAV09] P. Moret, W. Binder, D. Ansaloni, and A. Villazon. Visualizing calling context profiles with ring charts. In *Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on*, pages 33 –36, sept. 2009.

[McC76] TJ McCabe. A Complexity Measure. *IEEE Trans. on Software Engineering*, pages 308–320, 1976.

[ME98] B. Meyer and G. EiffelSoft. The Role of Object-Oriented Metrics. *Computer*, 31(11):123–127, 1998.

[MFM03a] A. Marcus, L. Feng, and J.I. Maletic. 3D Representations for Software Visualization. In *ACM Symp. Soft. Vis.*, 2003.

[MFM03b] A. Marcus, L. Feng, and JI Maletic. Comprehension of Software Analysis Data Using 3D Visualization. In *11th IEEE Int'l Workshop on Prog. Comprehension*, pages 105–114, 2003.

[MK92] J.C. Munson and T.M. Khoshgoftaar. The Detection of Fault-Prone Programs. *IEEE Trans. Software Engineering*, 18(5):423–433, 1992.

[MM05] PF Mihancea and R. Marinescu. Towards the Optimization of Automatic Detection of Design Flaws in Object-Oriented Software Systems. In *Proc. 9th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 92–101, 2005.

[MMC02] JI Maletic, A. Marcus, and ML Collard. A Task Oriented View of Software Visualization. In *1st Int'l Workshop on Vis. Soft. for Understanding and Analysis*, pages 32–40, 2002.

[Moc03] M. Mock. Dynamic analysis from the bottom up. In *WODA 2003 ICSE Workshop on Dynamic Analysis*, page 13. Citeseer, 2003.

[MP05] BA Malloy and JF Power. Using a Molecular Metaphor to Facilitate Comprehension of 3D Object Diagrams. In *2005 IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 233–240, 2005.

[MT04] T. Mens and T. Tourwe. A Survey of Software Refactoring. *IEEE Trans. Software Engineering*, 30(2):126–139, 2004.

[Mül86] Hans Albert Müller. *Rigi: a Model for Software System Construction, Integration, and Evolution Based on Module Interface Specifications.* PhD thesis, Rice Univ. in Houston, 1986.

[Mun97] T. Munzner. H3: Laying Out Large Directed Graphs in 3D Hyperbolic Space. In *IEEE Symp. on Info. Vis.*, pages 2–10, 1997.

[NL05] A. Noack and C. Lewerentz. A Space of Layout Styles for Hierarchical Graph Models of Software Systems. In *Proc. ACM Symp. on Soft. Vis.*, pages 155–164, 2005.

[OEGQ07] Hector M. Olague, Letha H. Etzkorn, Sampson Gholston, and Stephen Quattlebaum. Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes. *IEEE Trans. Soft. Eng.*, 33:402–419, 2007.

[OJH03] Alessandro Orso, James Jones, and Mary Jean Harrold. Visualization of program-execution data for deployed software. In *Proceedings of the 2003 ACM symposium on Software visualization*, SoftVis '03, pages 67–ff, New York, NY, USA, 2003. ACM.

[OKA11] I. Oikonomidis, N. Kyriazis, and A. Argyros. Efficient model-based 3d tracking of hand articulations using kinect. *BMVC, Aug*, 2, 2011.

[PAC01] H.C. Purchase, J.A. Allder, and D. Carrington. User Preference of Graph Layout Aesthetics: A UML Study. *Lecture Notes in Computer Science*, pages 5–18, 2001.

[PBG98] M. Petre, AF Blackwell, and TRG Green. Cognitive Questions in Software Visualization. *Soft. Vis.: Prog. as a Multimedia Experience*, pages 453–480, 1998.

[PBG03] T. Panas, R. Berrigan, and J. Grundy. A 3D Metaphor for Software Production Visualization. In *7th Int'l Conf. on Info. Vis.*, pages 314–319, 2003.

[PBS93] B.A. Price, R. Baecker, and I.S. Small. A principled taxonomy of software visualization. *J. of Visual Languages and Computing*, 4(3):211–266, 1993.

[PdQ06] M. Petre and E. de Quincey. A Gentle Overview of Software Visualisation. *The Computer Society of India Comm.*, 2006.

[PEQ+07] T. Panas, T. Epperly, D. Quinlan, A. Saebjornsen, and R. Vuduc. Communicating Software Architecture using a Unified Single-View Visualization. In *12th IEEE Int'l Conf. on Eng. Complex Computer Syst.*, pages 217–228, 2007.

[PFW98] G. Parker, G. Franck, and C. Ware. Visualization of Large Nested Graphs in 3D: Navigation and Interaction. *J. of Visual Languages and Computing*, 9(3):299–317, 1998.

[PGFL05] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing Multiple Evolution Metrics. In *Symp. on Soft. Vis.* ACM, 2005.

[PGJ05] M. Pinzger, H.C. Gall, and M. Jazayeri. ArchView-Analyzing Evolutionary Aspects of Complex Software Systems. *Vienna Univ. of Tech.*, 2005.

[PGKG08] Martin Pinzger, Katja Graefenhain, Patrick Knab, and Harald C. Gall. A Tool for Visual Understanding of Source Code Dependencies. In *16th Int'l Conf. on Prog. Comprehension*, pages 254–259. IEEE, 2008.

[PLL04] Thomas Panas, Jonas Lundberg, and Welf Löwe. Reuse in Reverse Engineering. In *Int'l Conf. on Prog. Comprehension*, 2004.

[PLL05] T. Panas, R. Lincke, and W. Löwe. Online-Configuration of Software Visualizations with Vizz3D. In *Symp. on Soft. Vis.*, 2005.

[Plo02] D. Ploix. Analogical Representations of Programs. In *1st Int'l Workshop on Vis. Soft. for Understanding and Analysis*, 2002.

[PRW03] M.J. Pacione, M. Roper, and M. Wood. A comparative evaluation of dynamic visualisation tools. In *10th Working Conference on Reverse Engineering*, pages 80–89, 2003.

[PV03] S. Purao and V. Vaishnavi. Product Metrics for Object-Oriented Systems. *ACM Computing Surveys*, 35(2):191–221, 2003.

[Rat03] D. Ratiu. Time-Based Detection Strategies. Master's thesis., University of Timisoara, 2003.

[RC92] G.C. Roman and KC Cox. Program Visualization: The Art Of Mapping Programs to Pictures. In *14th ACM Int'l Conf. on Soft. Eng.*, pages 412–420, 1992.

[RCM93] G.G. Robertson, S.K. Card, and J.D. Mackinlay. Information Visualization Using 3D Interactive Animation. *Commun. ACM*, 1993.

[RCR+93] G.C. Roman, K.C. Cox, D. Roman, et al. A Taxonomy of Program Visualization Systems. *IEEE Computer*, 1993.

[RDGM04] D. Ratiu, S. Ducasse, T. Girba, and R. Marinescu. Using History Information to Improve Design Flaws Detection. In *Proc. 8th European Conference on Software Maintenance and Reengineering (CSMR '04)*, pages 223–232, 2004.

[RDSGA+00] C. Russo Dos Santos, P. Gros, P. Abel, D. Loisel, N. Trichaud, and JP Paris. Metaphor-Aware 3D Navigation. In *IEEE Symp. on Info. Vis.*, pages 155–165, 2000.

[Rei95] S.P. Reiss. An Engine for the 3D Visualization of Program Information. *J. of Visual Languages and Computing*, 6, 1995.

[RG00] O. Radfelder and M. Gogolla. On Better Understanding UML Diagrams Through Interactive Three-Dimensional Visualization and Animation. In *Conf. on Advanced Visual Interfaces*, 2000.

[Rie96] A.J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1996.

[RM05] J. Rilling and SP Mudur. 3d visualization techniques to support slicing-based program comprehension. *Computers & Graphics*, 29(3):311–329, 2005.

[RR05] S.P. Reiss and M. Renieris. Jove: java as it happens. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 115–124. ACM, 2005.

[RWM03] J. Rilling, J. Wang, and SP Mudur. Metaviz–issues in software visualizing beyond 3d. In *Proc. 2nd International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), Amsterdam, Netherlands*, pages 92–97, 2003.

[SB91] RW Selby and VR Basili. Analyzing Error-Prone System Structure. *IEEE Trans. on Software Engineering*, 17(2):141–152, 1991.

[SB99] M.L. Staples and J.M. Bieman. 3D Visualization of Software Structure. *Advances in Computers*, 49:96–143, 1999.

[SBC00] V.C. Sreedhar, M. Burke, and J.D. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 196–207. ACM, 2000.

[SBM01] M.A. Storey, C. Best, and J. Michand. SHriMP Views: an Interactive Environment for Exploring Java Programs. In *9th Int'l Workshop on Prog. Comprehension*, pages 111–112, 2001.

[SCG05] M.A.D. Storey, D. Cubranic, and D.M. German. On the Use of Visualization to Support Awareness of Human Activities in Software Development: a Survey and a Framework. In *ACM Symp. on Soft. Vis.*, pages 193–202, 2005.

[SCGM00] J. Stasko, R. Catrambone, M. Guzdial, and K. McDonald. An Evaluation of Space-Filling Information Visualizations for Depicting Hierarchical Structures. *Int'l J. of Human Computer Studies*, 53:663–694, 2000.

[Sch92] NF Schneidewind. Methodology for Validating Software Metrics. *IEEE Trans. on Software Engineering*, 18(5):410–422, 1992.

[Sch02] NE Schneidewind. Body of Knowledge for Software Quality Measurement. *Computer*, 35(2):77–83, 2002.

[SCWP09] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 performance characterization. *Computer Performance Evaluation and Benchmarking*, pages 17–35, 2009.

[SDPK01] G. Sevitsky, W. De Pauw, and R. Konuru. An information exploration tool for performance analysis of java programs. In *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 38. Proceedings*, pages 85–101. IEEE, 2001.

[SFM99] M.D. Storey, FD Fracchia, and HA Müller. Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration. *J. of Syst. & Soft.*, 44, 1999.

[Shn92] B. Shneiderman. Tree Visualization with Tree-Maps: 2D Space-Filling Approach. *ACM Trans. On Graphics*, 11(1), 1992.

[SHR+00] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 264–280. ACM, 2000.

[SIG07] M. Shanmugasundaram, P. Irani, and C. Gutwin. Can Smooth View Transitions Facilitate Perceptual Constancy in Node-Link Diagrams? In *Graphics Interface*, page 78. ACM, 2007.

[SJSJ05] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using Dependency Models to Manage Complex Software Architecture. In *20th Conf. on OO Prog., Syst., Lang., and App.* ACM, 2005.

[SK03] Ramanath Subramanyam and M. S. Krishnan. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *IEEE Trans. Soft. Eng.*, 29, 2003.

[SL10] F. Steinbruckner and C. Lewerentz. Representing development history in software cities. In *Proceedings of the 5th international symposium on Software visualization*, pages 193–202. ACM, 2010.

[SM95] M.A.D. Storey and HA Müller. Manipulating and Documenting Software Structures Using SHriMP Views. In *Int'l Conf. on Soft. Maintenance*, pages 275–284, 1995.

[SM08] Inc. Sun Microsystems. Java platform standard ed. 6. package java.lang.instrument, 2008.

[SMB+11] A. Sarimbekov, P. Moret, W. Binder, A. Sewe, and M. Mezini. Complete and Platform-Independent Calling Context Profiling for the Java Virtual Machine. In *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, pages 1–15, 2011.

[SMK96] M.A.D. Storey, HA Müller, and Wong K. Manipulating and Documenting Software Structures. In *Soft. Vis.*, pages 244–263, 1996.

[Sõn09] S. Sõnajalg. Program analysis techniques for method call devirtualization in object-oriented languages. In *URL: http://www. cs. ut. ee/varmo/seminar/-sem09S/final/s6najalg. pdf*, 2009.

[Spe90] I. Spence. Visual Psychophysics of Simple Graphical Elements. *J. of Experimental Psychology: Human Perception and Performance*, 16:683–692, 1990.

[SR96] M. Scaifeand and Y. Rogers. External Cognition: How Do Graphical Representations Work? *Int'l J. of Human-Computer Studies*, 45:185–213, 1996.

[SS06] H.J. Schulz and H. Schumann. Visualizing Graphs - a Generalized View. In *IEEE Conf. on Info. Vis.*, pages 166–173, 2006.

[Sta93] John T. Stasko. Three-Dimensional Computation Visualization. In *IEEE Symp. on Visual Languages*, pages 100–107, 1993.

[Sta98] J. Stasko. *Software Visualization: Programming as a Multimedia Experience.* MIT press, 1998.

[Ste81] D. Steward. The Design Structure Matrix: A Method for Managing the Design of Complex Systems. *IEEE Trans. on Eng. Management*, 28:71–74, 1981.

[SW05] D. Sun and K. Wong. On Evaluating the Layout of UML Class Diagrams for Program Comprehension. In *13th Int'l Workshop on Prog. Comprehension*, pages 317–326, 2005.

[SWM97] M.A.D. Storey, K. Wong, and H.A. Müller. Rigi: A Visualization Environment for Reverse Engineering. In *19th ACM Int'l Conf. on Soft. Eng.*, pages 606–607, 1997.

[SWM00] M.A.D. Storey, K. Wong, and H.A. Müller. How Do Program Understanding Tools Affect How Programmers Understand Programs? *Science of Computer Prog.*, 36:183–207, 2000.

[SZ00] J. Stasko and E. Zhang. Focus + Context Display and Navigation Techniques for Enhancing Radial, Space-Filling Hierarchy Visualizations. In *IEEE Symp. on Info. Vis.*, pages 57–65, 2000.

[TA08] Alexandru Telea and David Auber. Code Flows: Visualizing Structural Evolution of Source Code. *Comput. Graph. Forum*, 27, 2008.

[TAD+10] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, December 2010.

[TBV09] Alexandru Telea, Heorhiy Byelas, and Lucian Voinea. A Framework for Reverse Engineering Large C++ Code Bases. *Electronic Notes in Theoretical Computer Science*, 233:143–159, 2009.

[TC09] Alfredo R. Teyseyre and Marcelo R. Campo. An Overview of 3D Software Visualization. *IEEE Trans. on Vis. & Comp. Graphics*, 15, 2009.

[TJ92] D. Turo and B. Johnson. Improving the Visualization of Hierarchies with Treemaps: Designissues and Experimentation. In *IEEE Conf. on Vis.*, pages 124–131, 1992.

[TKAM06] M. Tory, AE Kirkpatrick, MS Atkins, and T. Moller. Visualization Task Performance with 2D, 3D, and Combination Displays. *IEEE Trans. on Vis. & Computer Graphics*, 12(1), 2006.

[TKC99] M.H. Tang, M.H. Kao, and M.H. Chen. An Empirical Study on Object-Oriented Metrics. In *Proc. 6th International Software Metrics Symp.*, pages 242–249, 1999.

[TLTC05] M. Termeer, CFJ Lange, A. Telea, and MRV Chaudron. Visual Exploration of Combined Architectural and Metric Information. In *3rd IEEE Int'l Workshop on Vis. Soft. for Understanding and Analysis*, pages 1–6, 2005.

[TM04] M. Tory and T. Moller. Human Factors in Visualization Research. *IEEE Trans. Vis. and Computer Graphics*, 10, 2004.

[Tod] Dejan Todorovic. Gestalt Principles. `http://www.scholarpedia.org/article/Gestalt_principles`.

[TS07] Ying Tu and Han-Wei Shen. Visualizing Changes of Hierarchical Data using Treemaps. *Trans. on Vis. & Comp. Graphics*, 13, 2007.

[Tud03] M.E. Tudoreanu. Designing Effective Program Visualization Tools for Reducing User's Cognitive Effort. In *Symp. on Soft. Vis.*, 2003.

[TV08] Alexandru Telea and Lucian Voinea. An Interactive Reverse Engineering Environment for Large-Scale C++ Code. In *4th ACM Symp. on Soft. Vis.*, pages 67–76, 2008.

[TZ93] D.A. Troy and S.H. Zweben. *Measuring the Quality of Structured Designs.* McGraw-Hill, Inc., 1993.

[VBM09] A. Villazon, W. Binder, and P. Moret. Flexible Calling Context Reification for Aspect-Oriented Programming. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 63–74, 2009.

[vHvW03] F. van Ham and J.J. van Wijk. Beamtrees: Compact Visualization of Large Hierarchies. *Info. Vis.*, 2(1):31–39, 2003.

[VMV96] A. Von Mayrhauser and AM Vans. Identification of Dynamic Comprehension Processes During Large Scale Maintenance. *IEEE Trans. on Soft. Eng.*, 22(6):424–437, 1996.

[VRDBDR07] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *Software Engineering, IEEE Transactions on*, 33(12):800–817, 2007.

[VWVdW99] JJ Van Wijk and H. Van de Wetering. Cushion Treemaps: Visualization of Hierarchical Information. In *Symp. on Info. Vis.*, 1999.

[War04] C. Ware. *Information Visualization: Perception for Design.* Morgan Kaufmann, 2004.

[WB01] T.L. Woodings and G.A. Bundell. A Framework for Software Project Metrics. In *Proc. 12th European Conference on Software Control and Metrics (ESCOM '01)*, 2001.

[WC99] U. Wiss and DA Carr. An Empirical Study of Task Support in 3D Information Visualizations. In *Int'l Conf. on Info. Vis.*, 1999.

[WCG03] N. Wong, S. Carpendale, and S. Greenberg. Edgelens: An Interactive Method for Managing Edge Congestion in Graphs. In *IEEE Symp. on Info. Vis.*, pages 51–58, 2003.

[Whi02] A. White. SERP, an Open Source framework for manipulating Java bytecode. http://serp.sourceforge.net/, 2002.

[WL07a] R. Wettel and M. Lanza. Program Comprehension Through Software Habitability. In *Int'l Conf. on Prog. Comprehension*, 2007.

[WL07b] R. Wettel and M. Lanza. Visualizing Software Systems as Cities. In *4th IEEE Int'l Workshop on Vis. Soft. for Understanding and Analysis*, 2007.

[WL08a] R. Wettel and M. Lanza. Visual Exploration of Large-Scale System Evolution. In *IEEE 15th Working Conf. on Rev. Eng.*, 2008.

[WL08b] R. Wettel and M. Lanza. Visually Localizing Design Problems with Disharmony Maps. In *4th ACM Symp. on Soft. Vis.*, 2008.

[WLR11] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: A controlled experiment. In *Proceedings of ICSE*, volume 11, 2011.

[WS79] C. Wetherell and A. Shannon. Tidy Drawings of Trees. *Trans. Soft. Eng.*, 5:514–520, 1979.

[WS00] J. Wu and M.A.D. Storey. A Multi-Perspective Software Visualization Environment. In *Conf. of the Centre for Advanced Studies on Collaborative Research*, 2000.

[WWDW06] W. Wang, H. Wang, G. Dai, and H. Wang. Visualization of Large Hierarchical Data by Circle Packing. In *ACM SIGCHI Conf. on Human Factors in Computing Systems*, pages 517–520, 2006.

[XPM06] X. Xie, D. Poshyvanyk, and A. Marcus. Visualization of CVS Repository Information. In *13th Conf. on Rev. Eng.*, 2006.

[XSZC00] M. Xenos, D. Stavrinoudis, K. Zikouli, and D. Christodoulakis. Object-Oriented Metrics - a Survey. In *Proc. Federation of European Software Measurement Associations*, pages 1–10, 2000.

[YAR99] S.M. Yacoub, H.H. Ammar, and T. Robinson. Dynamic metrics for object oriented designs. In *Software Metrics Symposium, 1999. Proceedings. Sixth International*, pages 50–61. IEEE, 1999.

[YG03] H.Y. Yang and H. Graham. Software Metrics and Visualisation. Technical report, Univ. of Auckland, 2003.

[YM98] P. Young and M. Munro. Visualising Software in Virtual Reality. In *6th Int'l Workshop on Prog. Comprehension*, pages 19–26, 1998.

[You96] P. Young. Three Dimensional Information Visualisation. *Computer*, 1996.

[YR01] Z. Yu and V. Rajlich. Hidden Dependencies in Program Comprehension and Change Propagation. In *Proc. 9th International Workshop on Program Comprehension (IWPC)*, pages 293–299, 2001.

[YT07] H.Y. Yang and E. Tempero. Measuring the Strength of Indirect Coupling. In *Australian Software Engineering Conference*, 2007.

[Zha03] K. Zhang. *Software Visualization: from Theory to Practice*. Springer, Inc., 2003.

Résumé

Ce travail s'inscrit dans le cadre des recherches menées autour de l'analyse et la visualisation des logiciels, notamment les logiciels à objets, et en particulier Java. Très brièvement, on peut dire que le but de cette thèse revient à tenter de répondre à une question fondamentale: comment faire pour faciliter la compréhension du logiciel par ses développeurs et concepteurs ?

Ce travail de recherche est basé en grande partie sur deux axes principaux. Le premier consiste à analyser l'exécution des programmes, non seulement au niveau de la méthode, mais bien au niveau du bloc de base, pour recueillir des données d'exécutions avec un maximum de précision comme par exemple les différents types d'instances sur les sites d'appels. Le second axe considère l'utilisation des informations apportées par notre analyse dynamique de l'exécution pour permettre la visualisation de ces données. En effet, ces informations offrent des détails intéressants sur le fonctionnement du programme et aident à expliquer le comportement du logiciel, aussi bien pour déceler les problèmes de performance que les problèmes de codages.

Nous proposons une technique souple et efficace qui effectue une analyse dynamique de l'exécution de programmes Java. Nous introduisons ainsi une nouvelle technique et un nouvel outil permettant de recueillir des informations encore non proposées par d'autres analyseurs.

Cette approche trace l'exécution précise des programmes tout en ayant une baisse des performances d'exécution acceptable, laissant le programme final utilisable.

De plus, nous proposons et expérimentons une approche basé sur la visualisation des relations au sein d'une représentation du logiciel par une métaphore de ville. Nous introduisons une nouvelle technique de représentation des relations nommée *3D Hierarchical Edge Bundles* qui est basée sur une représentation 2D existante nommée *Hierarchical Edge Bundles*.

Cette approche conserve la puissance de visualisation du logiciel offerte par la métaphore de la ville tout en ajoutant la représentation des relations, et cela d'une façon lisible.

Ces travaux sont validés entre autres par le développement d'un outil d'analyse nommé VITRAIL JBInsTrace et d'un outil de visualisation nommé VITRAIL Visualizer. Ces outils sont la base de nos recherche actuelles sur l'étude de l'exécution des programmes objets.

Mots-clés: Analyse statique et dynamique des programmes objets, métriques logicielles, visualisation logicielle

Abstract

This work falls within the scope of research pertaining to the analysis and the visualization of software systems, especially for object oriented languages, and more precisely Java. In a nutshell, it can be said the aim of this thesis is to try to answer a fundamental question: what can we do to ease the understanding of software by its designers and developers ?

This research work is mainly based on two axes. The first axis consists in analyzing software runtime, not only at method level, but also at basic bloc level, so as to be able to get meaningful and precise information about the runtime. For instance, we can acquire the different types of instances on call sites at runtime. The second axis considers the use of information coming from our dynamic analyzer of software runtime and allowing the visualization of these data. Indeed, this kind of information offers important details about software functioning and provide a way to explain the behavior of software, so as to identify performance, coding and even design and architecture issues.

We propose a technique that allows flexible and efficient dynamic analysis of the execution of Java programs. We thus introduce a new technique and tool for gathering information not yet offered by other analyzers.

This approach precisely traces the execution of programs with acceptable performance penalty, that is while keeping the traced programs usable.

In addition, we propose and experiment an approach based on visualizing relationships within a software city representation. We introduce a new technique for representing relationships in 3D named the *3D Hierarchical Edge Bundles* that is based on an existing 2D technique, the *Hierarchical Edge Bundles*.

This approach keeps the power of the software city metaphor while adding the representation of the relationships within the software, in a readable way.

These works are validated by, among others things, the development of a tracer and analyzer tool called VITRAIL JBInsTrace and a visualization tool called VITRAIL Visualizer. These tools are used on our current researches which consist in studying runtime of object-oriented programs.

Keywords: Static and dynamic analysis of object-oriented programs, software metrics, software visualization