Université de Montréal

Formulation interactive des requêtes pour l'analyse et la compréhension du code source

par

Jamel Eddine Jridi

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

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

Novembre, 2010

Université de Montréal
Faculté des arts et des sciences

Ce mémoire intitulé :

Formulation interactive des requêtes pour l'analyse et la compréhension du code source

Présenté par :
Jamel Eddine Jridi

a été évalué par un jury composé des personnes suivantes :

Jian Yun Nie, président-rapporteur
Houari Sahraoui, directeur de recherche
Philippe Langlais, co-directeur
Julie Vachon, membre du jury

Mémoire accepté le :

RÉSUMÉ

Nous proposons une approche basée sur la formulation interactive des requêtes. Notre approche sert à faciliter des tâches d'analyse et de compréhension du code source. Dans cette approche, l'analyste utilise un ensemble de filtres de base (linguistique, structurel, quantitatif, et filtre d'interactivité) pour définir des requêtes complexes. Ces requêtes sont construites à l'aide d'un processus interactif et itératif, où des filtres de base sont choisis et exécutés, et leurs résultats sont visualisés, changés et combinés en utilisant des opérateurs prédéfinis. Nous avons évalués notre approche par l'implantation des récentes contributions en détection de défauts de conception ainsi que la localisation de fonctionnalités dans le code. Nos résultats montrent que, en plus d'être générique, notre approche aide à la mise en œuvre des solutions existantes implémentées par des outils automatiques.

Mots-clés : compréhension du code, analyse du code source, visualisation interactive, interrogation du code source.

ABSTRACT

We propose an interactive querying approach for program analysis and comprehension tasks. In our approach, an analyst uses a set of basic filters (linguistic, structural, quantitative, and user selection) to define complex queries. These queries are built following an interactive and iterative process where basic filters are selected and executed, and their results displayed, changed, and combined using predefined operators. We evaluated our querying approach by implementing recent state-of-the-art contributions on feature location and design defect detection. Our results show that, in addition to be generic; our approach helps improving existing solutions implemented by fully-automated tools.

Keywords : program comprehension, source code analysis, source code querying, Interactive visualization.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES

CBO	Coupling Between Objects
DIT	Depth of Inheritance Tree
LCOM	Lack of Cohesion in Methods
LOC	Lines Of Code
NMD	Number of Methods Declared
NACC	Number of ACCessor
OO	Orienté Objet
VERSO	Visualization for Evaluation and Re-engineering of object-oriented SOftware
WMC	Weighted Methods per Class
NINTERF	Number of Interface
NMPARAM	Number of PARAMeters
NAD	Number of Attributes Declared
LSI	Latent Semantic Indexing
TF	Term Frequency
IDF	Inverse Document Frequency
IQOP	Interactive Querying of Object-oriented Programs
SRI	Système de Recherche d'Information
JDK	Java Development Kit
SVD	Singular Values Decomposition

DÉDICACE

Je dédie ce travail à :

Mes parents Mohamed et Naziha pour leur soutien durant toutes mes années d'études : je ne serais être qu'infiniment reconnaissant quant aux sacrifices qu'ils ont consentis.

Mes beaux parents Mabrouk et Fethia pour leur amour et leur générosité.

Hajer, ma fiancée, qui m'a apporté soutien et réconfort dans les moments de doute. Je lui dédie ce travail.

Mon cher frère et ma petite sœur : Abdel Wadoud et Fatma pour leur amour et à qui je souhaite une vie pleine de réussites et de bonheur.

Tous mes amis(es) et collègues.

Jamel

REMERCIEMENTS

Je remercie tout d'abord mes directeurs Pr. Houari Sahraoui et Pr. Philippe Langlais pour avoir dirigé ce travail. Je les remercie pour leur patience et leur disponibilité dont ils ont fait preuve à mon égard. Leurs conseils et remarques constructifs m'ont permis d'améliorer grandement la qualité de mes travaux et de ce mémoire. Je les remercie également pour la confiance qu'ils m'ont témoignée tout au long de ces années.

J'adresse pareillement mes sincères remerciements aux membres de jury pour l'honneur qu'ils ont fait en participant à ce jury et d'avoir bien voulu apporter leurs conseils et leurs jugements sur ce modeste travail.

Je remercie les membres de laboratoire de génie logiciel GEODES pour leur gentillesse et leur respect : merci à Marouane pour ses conseils, ses encouragements et pour toutes les conversations que nous avons eues autour d'un bon repas, merci à Martin de mettre l'ambiance dans le laboratoire, merci à Dorsaf pour sa gentillesse et son soutien…

Mes remerciements s'adressent également à tous mes enseignants, chacun de son domaine, pour leur contribution à ma formation durant mes études universitaires.

Finalement, je remercie les membres de ma petite et grande famille, mon papa Mohamed, ma maman Naziha, mon frère Dido, ma belle petite sœur Fatouma, ma fiancée Hajoura, ainsi que mes amis de longue date, Issam Riahi et Majdi Guebebia pour leurs nombreux encouragements depuis le début de mes études universitaires.

À tous ceux qui ont contribué de près ou de loin à la réalisation de ce travail.

Chapitre 1.

Introduction générale

Dans ce chapitre, nous présentons dans un premier temps le contexte général de notre travail en détaillant l'importance d'avoir des outils qui aident à la maintenance et à la compréhension du code. Ensuite, nous discutons les principaux problèmes liés à la formulation de la requête, c'est-à-dire, comment le développeur peut formuler ses besoins en termes de requêtes? Après, nous décrivons les solutions proposées dans notre étude. Enfin, nous donnons un aperçu des différents chapitres de notre mémoire.

1.1 Contexte

La compréhension de programme est une activité cruciale du génie logiciel et de maintenance. Elle permet de faciliter la réutilisation, l'inspection, la rétro-ingénierie, la réingénierie, la refactorisation, etc. du code. Aujourd'hui, les systèmes logiciels sont de plus en plus complexes, ce qui rend leur maintenance très difficile. En effet, de nombreux experts conviennent que les efforts des programmeurs sont principalement consacrés à la maintenance des systèmes : 60 à 80% des coûts des projets ([17], [1]). Cela n'est pas surprenant quand on considère la quantité et la qualité du code à maintenir. Dans ce contexte, les tâches de maintenance exigent aux mainteneurs de passer la plupart de leur temps de travail à analyser et comprendre le code source. Disposer d'outils adéquats pour l'analyse et la compréhension des programmes faciliterait le travail du développeur et diminuerait les coûts de maintenance. Dans cette perspective, différents outils et techniques ont été proposés pour automatiser totalement ou partiellement les tâches de maintenance. Ces travaux pourraient être classés en deux familles : (1) des environnements génériques pour l'analyse et la compréhension des programmes, généralement basés sur la formulation des requêtes [2] [5] [6] [4] [31] [23] [19], et (2) des outils qui ciblent des tâches spécifiques telles que : la localisation de fonctionnalités [8] [22] [7] et la détection des défauts de conception [27] [29] [14].

1.2 Problématique

De nos jours, vu la complexité des programmes et le coût consacré à la maintenance, il devient crucial d'avoir des outils d'exploration et d'interrogation du code source des systèmes logicielles. Le problème majeur des développeurs réside dans les difficultés qu'ils ont à exprimer leurs besoins en termes de requêtes. Différentes techniques ont été proposées dans la littérature afin de réduire les efforts de maintenance. La majorité des travaux sont basés sur la formulation des requêtes (e.g, en langage naturel ou logiques) et certains d'entre eux sont destinés à résoudre des problèmes spécifiques en génie logiciel. Cependant, la formulation de ces requêtes peut être une tâche difficile et fastidieuse dans de nombreuses situations. Par exemple, lors de la détection des défauts de conception, il est difficile de formuler des requêtes permettant de réaliser cette tâche. En effet, de nombreux défauts ont les mêmes symptômes et il n'existe pas de définitions consensuelles qui peuvent être formulées sous forme de requêtes [29]. Ainsi, l'interaction avec l'utilisateur est nécessaire pour affiner les résultats après la formulation de certaines requêtes. Toutefois, les travaux existants ne prennent pas en considération cet aspect d'interactivité. Le processus d'exécution d'une requête est similaire à celui d'une boîte noire dont l'utilisateur n'a pas le contrôle. La formulation de requêtes pour de nombreuses tâches de maintenance est difficile pour deux raisons principales. Premièrement, il existe des requêtes permettant d'évaluer des conditions sur diverses entités d'un programme, par exemple dire qu'une classe ou un ensemble de classes sont trop complexes. Ce type des requêtes nécessite que le mainteneur spécifie une valeur de seuil, par exemple, le niveau de complexité admissible d'une classe ou bien le degré de similarité d'une méthode avec une requête en langage naturel [22] [7] [8]. Par exemple, selon Marinescu [29], une classe qui est considérée grande dans un programme pourrait être considérée moyenne dans d'autres. Cet aspect rend le processus de détection plus difficile à automatiser.

La deuxième raison qui rend la formulation des requêtes difficile est que plusieurs filtres de bases doivent être combinés de façon non triviale pour réaliser des tâches complexes. Par exemple, dans une approche de localisation de fonctionnalités, un filtre structurel

permettant d'explorer le graphe d'appel doit être combiné avec des filtres de recherche par mots clés [8]. Étant donné que ces requêtes complexes doivent s'appliquer à différents programmes, une méthode de combinaison fixe conduit généralement à de nombreux faux positifs [29] [27]. Ceci est révélateur du fait que des combinaisons de requêtes figées constituent une limite majeure à l'automatisation.

1.3 Contribution

Les problèmes de paramétrage de la recherche (valeurs seuils, etc.) et de composition de filtres pourraient être contournés si le mainteneur avait la possibilité de définir de façon interactive et itérative des requêtes complexes. La connaissance du contexte du programme aide aussi l'utilisateur à prendre de meilleures décisions dans sa recherche. En outre, en précisant la requête d'une façon itérative, un filtre après l'autre, il est plus facile de réviser et d'ajuster ces décisions. Par exemple, lors de la recherche d'une fonctionnalité, le mainteneur peut explorer le graphe d'appels, niveau par niveau, et décider à quel niveau il va s'arrêter en fonction des résultats déjà obtenus. C'est dans ce cadre que s'inscrit notre travail, qui consiste à mettre en place un outil d'interrogation du code pour la réalisation de tâches de maintenance.

Dans ce mémoire, nous proposons un environnement interactif pour explorer un ensemble de données extraites du code source. Notre environnement, nommé IQOP (pour Interactive Querying of Object-oriented Programs), offre la possibilité d'utiliser différents types de filtres de base (filtre structurel, filtre de recherche d'information, filtre quantitatif et filtre d'interactivité). Il offre également des opérateurs pour combiner ces filtres de base (opérateurs relationnels, itérateurs, etc.). La formulation de requêtes se fait de façon itérative et les résultats de chaque étape sont affichés en utilisant une métaphore de visualisation offerte par VERSO [10], un outil de visualisation de logiciel utilisant la métaphore de ville. Dans ce contexte, un logiciel est vu comme une ville organisée en quartiers (packages) qui représentent des regroupements de bâtiments (éléments logiciels tels que les classes) dans lesquels se réalisent des activités urbaines (comportement des éléments logiciels).

L'importance de la visualisation comme un complément pour les outils d'interrogation a été soulignée par Verbaere et al. [24]. La visualisation peut donner une vue globale du système, tout en mettant en évidence les résultats des filtres. Contrairement aux autres outils d'interrogation de code IQOP permet le plein contrôle du processus d'interrogation plutôt que d'appliquer un ensemble prédéfini de requêtes implantées de manière figée. Aussi, grâce à la métaphore de visualisation, l'utilisateur peut inspecter les résultats des filtres et possiblement modifier ces derniers par l'ajout et/ou la suppression d'éléments avant la composition avec d'autres filtres.

1.4 Structure du mémoire

Ce mémoire est organisé comme suit. Le chapitre 2 présente un survol des principaux travaux en relation avec notre sujet. Le chapitre 3 donne un aperçu général de notre approche. Cet aperçu inclut également une description de notre environnement interactif qui contient deux principaux modules: un module d'affichage de résultats et un autre de spécification de requêtes. Les détails de nos types de filtres sont présentés dans le chapitre 4. Dans le chapitre 5, nous détaillons notre processus de construction de requêtes complexes. Dans ce chapitre, nous décrivons les deux modes de formulation utilisés dans notre environnement interactif. Ensuite, les résultats d'une étude de cas sont présentés et discutés dans le chapitre 6. Enfin, dans le chapitre 7, nous récapitulons les principales idées introduites au sein de notre mémoire et nous proposons quelques perspectives d'amélioration de notre travail.

Chapitre 2.

État de l'art

2.1 Introduction

Dans le cadre de cette maîtrise, nous avons réalisé un environnement interactif d'interrogation du code source basé sur la formulation interactive et itérative de requêtes ainsi que sur la visualisation qui peut aider dans la prise de décisions. De nombreuses contributions d'exploration du code ont déjà été proposées. Elles visent toutes à aider les développeurs à bien explorer et à mieux comprendre le code, ainsi qu'à découvrir des informations qu'il serait difficile, voire impossible à extraire avec des outils standards. Nous catégorisons ces travaux en deux familles : (1) des environnements génériques pour l'analyse des programmes basés sur l'utilisation des requêtes logiques, et (2) des outils qui ciblent des tâches spécifiques. Nous commençons par présenter un aperçu des travaux génériques pour l'interrogation du code en discutant des limites de ces approches. Ensuite, nous présentons quelques travaux qui sont destinés à résoudre des problèmes spécifiques de maintenance.

2.2 Approches génériques d'interrogation

Plusieurs approches ont été proposées pour explorer le code à l'aide de requêtes. Ces approches utilisent des langages spécifiques pour exprimer ces requêtes. Ces langages sont, en général, fortement inspirés des langages de manipulation de base de données comme SQL ou Datalog. L'une des premières propositions est celle de Linton qui a proposé de décrire un programme comme une base de données [18]. Son environnement, nommé OMEGA, permet de décrire 58 relations qui représentent les détails les plus importants d'un programme. L'exploration du code a été réalisée en utilisant des requêtes en langage QUEL, similaire à SQL, qui ne supporte pas la récursivité par exemple pour afficher différents niveaux dans un graphe d'appel. On trouve aussi des outils tels que CodeQuest

[5], SemmleCode [31], GraphLog [23] et JQuery [6] [4] qui se basent respectivement sur Datalog, SQL ou Prolog.

CodeQuest est un outil qui combine la programmation logique et l'utilisation des systèmes de bases de données. Contrairement à OMEGA, CodeQuest supporte les requêtes récursives par la traduction de Datalog en SQL. De son côté, SemmleCode est offert comme un plugin Eclipse qui donne la possibilité au développeur de faire des requêtes en utilisant le langage « .QL » qui est un langage de requêtes similaire à SQL et fondé sur Datalog. JQuery [4] est une synthèse de tous les types d'outils précédents. Il utilise un langage de programmation logique nommé TyRuBa similaire à Prolog. En outre, JQuery a une interface utilisateur conviviale, où les résultats des requêtes peuvent être organisés dans une vue hiérarchique. Contrairement aux autres approches, GraphLog offre une façon de visualiser et représenter la structure des systèmes logiciels sous forme de graphes. Proposé en 1992, GraphLog présente un langage de requêtes avec un bon pouvoir d'expression pour décrire les propriétés des chemins dans les graphes. D'après Mariano et al. [23], les requêtes en GraphLog sont plus faciles à écrire que leurs équivalentes en Prolog.

Plus récemment, De Alwis et Murphy [2] ont proposé un système nommé Ferret. Ils se sont basés sur 36 différents types de requêtes répondant aux différentes questions que le développeur peut se poser quand il manipule le code source. Ces requêtes touchent principalement à la structure du code, aux informations structurelles statiques et aux informations des flux de contrôle, telles que les relations interclasses, intraclasses, l'héritage, les déclarations, etc. Cet ensemble de requêtes prédéfinies et implémentées de manière figée limite le choix de l'utilisateur. Ces outils ont en commun qu'ils se limitent à l'exploration de la structure et des propriétés des artefacts logiciels. Ils ne bénéficient pas de la puissance des requêtes en langage naturel et des techniques de recherche d'information. Chowdhury a déclaré que « la façon la plus facile pour l'utilisateur d'exprimer son besoin d'information est en langage naturel ». Dans ce cadre, l'approche la plus récente est celle de Würsch et al. [25]. Cette approche, basée sur des techniques du web sémantique, est implantée dans un outil nommé EVOLIZER. Ce dernier représente la structure d'un

logiciel avec une ontologie OWL et utilise des techniques de traitement de connaissances comme SPARQL pour interroger cette ontologie. La formulation des requêtes dans EVOLIZER est faite en langage naturel comme le montre la Figure 1.

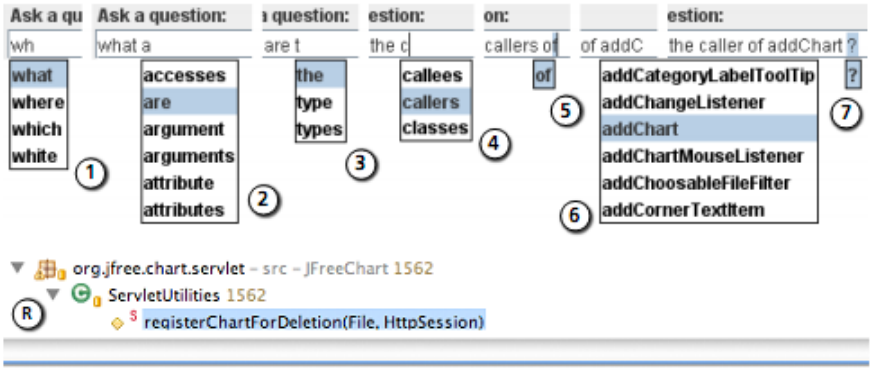


Figure 1. Saisie d'une requête et récupération des résultats dans EVOLIZER [25].

Dans cet outil, les auteurs ont utilisé l'ensemble de types de requêtes, qui se sont révélées les plus utiles pour les développeurs dans le système Ferret [2], celui proposé par De Alwis et Murphy [2]. De ce fait, EVOLIZER limite également le choix de l'utilisateur à l'ensemble des requêtes existantes. De plus, seul un ensemble de mots donnés du vocabulaire de la langue anglaise peut être utilisé pour formuler des requêtes.

Dans tous ces travaux, l'utilisateur peut exécuter une seule requête à la fois; il ne peut donc pas interagir avec les résultats pendant l'exécution de la requête puisqu'il n'a pas le contrôle du processus d'interrogation. Par ailleurs, l'utilisateur ne peut pas combiner plusieurs types de requêtes. En effet, les environnements décrits plus haut se concentrent uniquement sur des informations structurelles extraites statiquement du code telles que les relations entre les entités, etc. Ils ne permettent pas d'exprimer des requêtes de recherche d'information ou d'évaluation quantitative du code (conditions sur des métriques logicielles). En outre, le processus d'interrogation est monolithique et ne permet pas à l'usager d'intervenir avant la présentation des résultats finaux.

Les approches décrites dans cette section aident à explorer et à comprendre la structure d'un programme. Cependant, parfois, il y a un besoin pour résoudre des problèmes précis de maintenance qui nécessitent des connaissances spécifiques. C'est pour ce type de besoins

que plusieurs approches ont été proposées afin de résoudre des problèmes tels que la détection de défauts de conception ou l'évaluation de la qualité des logiciels. Dans la section suivante, nous allons décrire quelques travaux de cette famille.

2.3 Approches destinées à résoudre des problèmes spécifiques

Le génie logiciel est un domaine dédié à l'étude, la conception et la réalisation de systèmes logiciels (programmes informatiques). Les opérations de maintenance d'un logiciel se poursuivent pendant toute sa durée de vie et coûtent souvent plus cher que son développement. Pour faciliter la maintenance, en plus des approches génériques décrites dans la section précédente, plusieurs outils ont été proposés pour explorer, comprendre le système logiciel et trouver des critères d'évaluation de sa qualité. Ces outils sont destinés à résoudre une variété de tâches ou de problèmes spécifiques d'ingénierie de logiciels. L'un des problèmes intéressants traités par la communauté du logiciel depuis la dernière décennie est la détection de défauts (appelés aussi anomalies) de conception. Les anomalies sont de mauvais choix de conception ou de programmation, qui, sans être des bogues, peuvent causer indirectement des comportements incorrects [26].

Marinescu [29] propose une stratégie de détection de défauts de conception basée sur la composition de règles, en utilisant les métriques logicielles. Les règles permettent de choisir parmi les entités (classes et méthodes), celles répondant à des critères absolus ou relatifs. Ce mécanisme peut prendre en paramètre, pour chaque métrique, une valeur indiquant le seuil tolérable ou un nombre déterminant le minimum d'entités à retrouver (par exemple 20% des classes) ayant les plus grandes ou plus petites valeurs d'une métrique donnée.

Moha et al. [27] proposent une approche, nommée DECOR, qui consiste à décrire les défauts de conception sous la forme de règles en utilisant un langage déclaratif de haut niveau. La Figure 2 présente, par exemple, les règles spécifiques pour la détection d'un *spaghetti code* dans DECOR [27].

```
1   RULE_CARD : SpaghettiCode  {
2     RULE : SpaghettiCode
            { INTER LongMethod  NoParamete  NoInheritance
            NoPolymorphism  ProceduralName  UseGlobalVariable };
3     RULE : LongMethod        { METRIC LOC_METHOD VERY_HIGH 10.0 };
4     RULE : NoParameter       { METRIC NMNOPARAM VERY_HIGH 5.0 };
5     RULE : NoInheritance     { METRIC DIT 1 0.0 };
6     RULE : NoPolymorphism    { STRUCT NO_POLYMORPHISM };
7     RULE : ProceduralName    { LEXIC CLASS_NAME
                                    (Make, Create, Exec...) };
8     RULE : UseGlobalVariable { STRUCT USE_GLOBAL_VARIABLE };
9   };
```

Figure 2. Carte de règles pour la détection de spaghetti code par Moha et al. [27].

Les approches mentionnées ci-haut permettent la détection automatique sous forme d'un ensemble fixe de combinaisons des règles. Chaque combinaison permet de détecter un type de défauts de conception. La majorité des règles concernent des métriques pour lesquelles il faut spécifier des valeurs seuils. Ces dernières sont très difficiles à définir de manière uniforme compte tenu de la diversité des systèmes logiciels et des pratiques de développement. En d'autres termes, une valeur qui fonctionnerait pour un système, ne fonctionnerait probablement pas pour un autre.

Pour remédier à ce problème, Dhambri et al [14] proposent une approche semi-automatique de détection de défauts de conception basée sur la visualisation. Ils ont décrit une stratégie de détection pour chaque type de défaut (classe mal placée, blob…). Leur processus de détection se déroule en trois étapes : la première consiste à choisir l'ensemble de métriques pertinentes qui peuvent être utilisées pour détecter une anomalie particulière. Les valeurs de ces métriques seront associées aux attributs (couleur, orientation, hauteur) des objets graphiques représentant les entités du système. Ensuite, l'analyste essaye de localiser les classes qui peuvent être considérées comme des défauts, en se basant sur l'apparence de ces classes en fonction du mapping de métriques choisi. De plus, l'analyste peut utiliser un filtre permettant de visualiser la distribution des valeurs d'une métrique sur les entités, ce qui permet de localiser facilement des classes ayant des valeurs de métriques anormales. La dernière étape consiste à inspecter les classes choisies.

Ces approches destinées à détecter les défauts de conception, même si elles donnent des résultats plus ou moins bons, sont spécifiques à ce problème. Leur adaptation à des problèmes plus ou moins similaires n'est pas triviale et nécessite un effort important.

Toujours dans la catégorie des outils spécifiques, il existe d'autres travaux qui se sont intéressés à la traçabilité entre les éléments d'analyse et de conception, et les entités du code. Par exemple, la localisation de fonctionnalités dans le code a fait l'objet de plusieurs contributions. Ces contributions ont pour objectif d'aider les mainteneurs à localiser rapidement les parties à modifier suite à un changement dans les requis d'un système.

Marcus [22] a proposé une approche d'exploration basée sur une technique de recherche d'information nommée LSI (*latent semantic indexing*). Le but de Marcus est de trouver des entités du code source qui implémentent une fonctionnalité particulière décrite par une requête en langage naturel. Le mécanisme permet de chercher l'ensemble des entités qui sont sémantiquement proches de la requête entrée par l'utilisateur.

Hill et al. [8] proposent et évaluent un outil nommé DORA qui utilise à la fois les informations structurelles et lexicales du code source. Ils ont montré les limites d'une utilisation indépendante des informations structurelles et lexicales (recherche d'information). Cette approche consiste, à partir d'un point de départ dans le graphe d'appels, à choisir l'ensemble des voisins les plus pertinents par rapport à une requête en langue naturelle. Le choix des voisins se fait par rapport à une valeur seuil de pertinence par rapport à la requête. Le processus est ensuite récursivement répété sur les nœuds choisis. Une des limites de cette approche est la valeur seuil utilisée et le niveau du graphe d'appels à atteindre. Cette approche automatisée ne permet pas à l'analyste de moduler la valeur seuil en fonction des niveaux ni de décider en fonction des niveaux précédents où s'arrêter dans l'exploration du graphe d'appels.

2.4 Synthèse

Nous avons vu dans cette section les principales contributions d'interrogation du code qui ont été proposées. La plupart des approches génériques proposées dans la littérature

sont destinées à l'exploration de la structure du code et ne peuvent pas résoudre des problèmes précis comme la détection de défauts de conception. D'autres approches automatiques ont été proposées pour résoudre ce genre de problèmes comme Marinescu [29] et Moha et al. [27]. Les difficultés liées à ces approches de détection résident, d'une part, dans la définition de valeurs de seuil associées aux métriques logicielles utilisées, et d'autre part, dans la définition d'un ensemble de combinaisons de règles de détection d'une façon figée.

Dans ces approches automatiques, qu'elles soient génériques ou bien spécifiques, l'utilisateur n'a pas le contrôle de la situation. Le mainteneur n'a ni la possibilité d'inspecter ni la possibilité d'interagir avec les résultats. Par contre, Dhambri et al. [14] ont proposés une approche de détection semi automatique basée sur la visualisation, qui aide le mainteneur à inspecter les résultats. Mais cette approche reste toujours destinée à résoudre un problème précis.Pour remédier à toutes ces limites, nous proposons un outil générique dont l'utilisateur est l'acteur principal. Cet outil permet d'une part d'explorer et de comprendre le code source, et d'autre part, de résoudre des problèmes de maintenance précis. Cet outil offre également, la possibilité au mainteneur de combiner plusieurs types de requêtes d'une façon personnalisée. La construction de la requête est faite progressivement, c'est-à-dire, l'utilisateur applique un filtre à la foie et le combine avec les résultats précédents de la requête en utilisant les opérateurs. Cette construction permet d'inspecter les résultats de chaque filtre ajouté et possiblement de les modifier. .

Notre approche peut être une étape complémentaire aux approches automatiques. Elle permet d'une part de raffiner les résultats de ces dernières et d'autre part d'exécuter pas à pas leurs algorithmes pour déterminer les parties à améliorer. Par exemple, dans le cas des approches de détection de défauts de conception, une exécution progressive permet de déterminer les étapes qui génèrent les faux positifs et de les améliorer en conséquence. La détermination des étapes problématiques et leurs améliorations sont rendues possibles par la visualisation offerte par l'outil VERSO. Dans le reste du mémoire, nous détaillerons le principe de fonctionnement de notre environnement ainsi que ces composants.

2.5 Conclusion

Dans ce chapitre, nous avons fait une revue des travaux reliés à l'approche présentée dans ce mémoire. Nous avons commencé par présenter l'ensemble des approches et outils génériques proposés pour améliorer l'exploration du code source et la compréhension des programmes. Ensuite, nous avons décrit les travaux qui sont destinés à résoudre des problèmes spécifiques en maintenance du logiciel. Nous avons d'abord présenté les travaux qui touchent à la détection des anomalies de conception. Par la suite, nous avons énuméré d'autres techniques proposées destinées à la localisation des fonctionnalités. Enfin, nous avons proposé une synthèse des limites d'approches proposées dans la littérature. Dans le chapitre suivant, nous détaillons notre approche d'interrogation du code.

Chapitre 3.

Approche interactive d'interrogation de code

3.1 Introduction

Dans ce chapitre, nous commençons par la description de notre approche d'interrogation de code en utilisant la métaphore du cuisinier. Cette interrogation est guidée par un environnement interactif. Ensuite, nous décrivons le principe de notre environnement interactif. Cet environnement est composé de deux modules : un module d'affichage ou de visualisation et un module de spécification de requêtes. La structure de ce chapitre est la suivante : dans la section 3.2, nous décrivons notre approche en utilisant la métaphore du cuisinier et son panier. Ensuite dans la 3.3, nous présentons les principaux modules qui composent notre environnement interactif.

3.2 Description de l'approche

Pour illustrer le processus de construction d'une requête, nous considérons la métaphore du cuisinier et du panier. Le cuisinier doit réaliser une série de recettes. Pour cette raison, il doit trouver les ingrédients nécessaires dans un supermarché. Comme tout bon cuisinier, la liste finale des ingrédients utilisés dépend de ce qui est disponible. En outre, certains principes doivent être considérés; par exemple éviter les plats riches en matières grasses, etc. Donc, le cuisinier commence par la recherche des ingrédients importants en limitant la recherche aux départements du magasin correspondants à ces critères de recherche. Chaque fois qu'un ou plusieurs éléments sont trouvés, ils sont ajoutés au panier. À tout moment, le cuisinier peut accéder au contenu du panier. Il peut enlever un ingrédient s'il est incompatible avec d'autres ingrédients sélectionnés ou le remplacer afin de maintenir un taux acceptable de gras. Il peut également ajouter un nouvel ingrédient qui n'a pas été initialement prévu. À la fin de l'achat, le panier contient les résultats des activités de recherche effectuées ainsi que plusieurs autres éléments résultants de décisions prises au cours de la recherche. Notez que le cuisinier n'a pas utilisé de valeurs seuils spécifiques

pour le gras, le sel ou les glucides d'un ingrédient. Généralement, la construction de requêtes complexes pour certaines tâches d'analyse et de compréhension est similaire à la métaphore du cuisinier et de son panier. Dans notre environnement, l'interrogation est réalisée comme un ensemble de cycles successifs comme le montre la Figure 3.

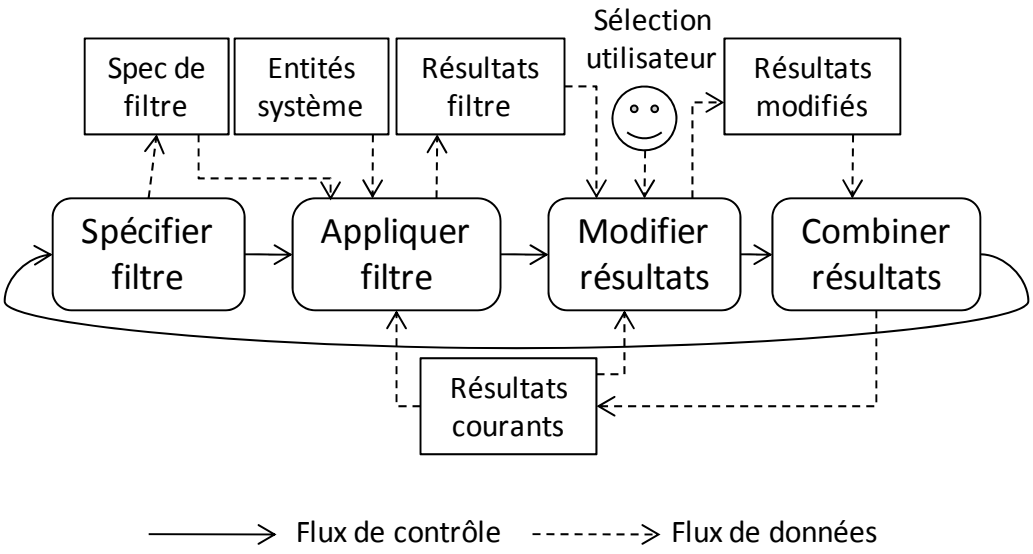


Figure 3. Processus de construction de requête.

Chaque cycle, représenté par les flèches de flux de contrôle, consiste à appliquer un filtre, inspecter les résultats de ce filtre, modifier ces résultats, et les combiner avec ceux des cycles précédents (résultats courants). Un filtre peut être appliqué à toutes les entités du système ou seulement aux résultats des cycles précédents. Si les résultats de ce filtre ne sont pas satisfaisants, alors il sera supprimé. À tout moment du processus de recherche, l'utilisateur de notre système peut juger de la pertinence d'un filtre. À la suite de chaque changement au niveau de l'environnement (ajout d'un filtre, modification de résultats, etc.), les résultats sont mis à jour et sont affichés en utilisant notre module de visualisation. Ceci permet d'aider le mainteneur dans ses tâches d'inspection et de modification des résultats.

3.3 Environnement interactif

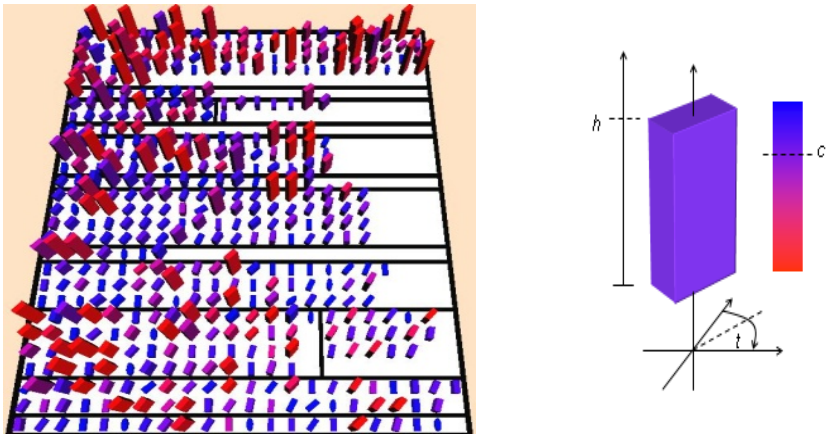
Comme souligné précédemment, les tâches de maintenance exigent des mainteneurs de passer la majorité de leur temps de travail à analyser le code source et à comprendre les fonctionnalités du programme. La visualisation est reconnue comme une des solutions qui permettent d'améliorer l'exploration et la compréhension du code. Partant de ce constat, notre environnement utilise la métaphore de visualisation développée dans le projet VERSO [10]. Cette métaphore permet d'afficher plusieurs caractéristiques du logiciel selon différents niveaux de granularité. Elle offre une vue globale du système nécessaire à la visualisation des résultats de requêtes.

En plus de la visualisation, notre environnement doit inclure des outils permettant d'interroger le code en utilisant différents types de requêtes. Ce module est basé sur la spécification et la composition de plusieurs types de filtres. Nous intégrons donc dans notre environnement un module de visualisation VERSO et un module interactif d'interrogation du code.

3.3.1 Module de visualisation

VERSO (**V**isualization for **E**valuation and **R**e-engineering of object-oriented **SO**ftware) est un outil de visualisation de logiciels de grande taille. Les entités d'un logiciel sont représentées par des éléments graphiques en trois dimensions qui sont disposés sur un plan à deux dimensions en respectant la structure des packages d'un programme. Le placement des entités est réalisé par l'algorithme de placement *TreeMap* [16] (Figure 4-(a)). *TreeMap* prend comme point d'entrée un rectangle qui représente le programme. Ensuite, ce rectangle est découpé en plusieurs tranches verticales, une pour chaque package principal. La taille de chaque zone dépend de la taille du package qui correspond au nombre de classes contenues dans ce dernier. De la même façon, les nouveaux rectangles associés aux packages principaux sont découpés horizontalement en fonction des sous packages respectifs. Cette opération, qui alterne les découpages verticaux et horizontaux, est répétée récursivement, au fur et à mesure que l'on parcourt l'arbre des packages. Les entités du

programme (classes, interfaces) qui sont contenues dans un package donné sont placées sur le rectangle correspondant.



(a). Vue global d'un système (b). Représentation d'une classe

Figure 4. Visualisation de logiciels dans VERSO.

Une classe est représentée sous forme d'une boîte 3D (Figure 4-(a)) dont la hauteur, l'orientation et la couleur sont associées à des métriques de classes (Figure 4-(b)). Différentes métriques peuvent être associées selon l'objectif de la visualisation. Par exemple, la hauteur peut correspondre à la métrique de complexité WMC (**W**eighted **M**ethod per **C**lass), l'orientation à la métrique de cohésion LCOM5 (**L**ack **C**ohesion **O**f **M**ethod), et la couleur (qui varie entre le bleu et rouge) à la métrique de couplage CBO (**C**oupling **B**etween **O**bjects) [10]. Les interfaces sont représentées par des cylindres pour les distinguer des classes. Les choix des formes graphiques ont pour but de facilité la perception tel qu'expliqué par Langelier [16].

La navigation dans la scène trois dimensions est gérée par une caméra. La caméra peut être déplacée n'importe où dans l'espace et définit la direction de vue. Il est aussi possible de s'approcher et de s'éloigner du plan pour mieux voir certains détails, avoir une meilleure vue d'ensemble ou bien descendre à un niveau de granularité plus bas (vue de méthodes). Plusieurs exemples de positions de caméra sont fournis à la Figure 5 ci-dessous :

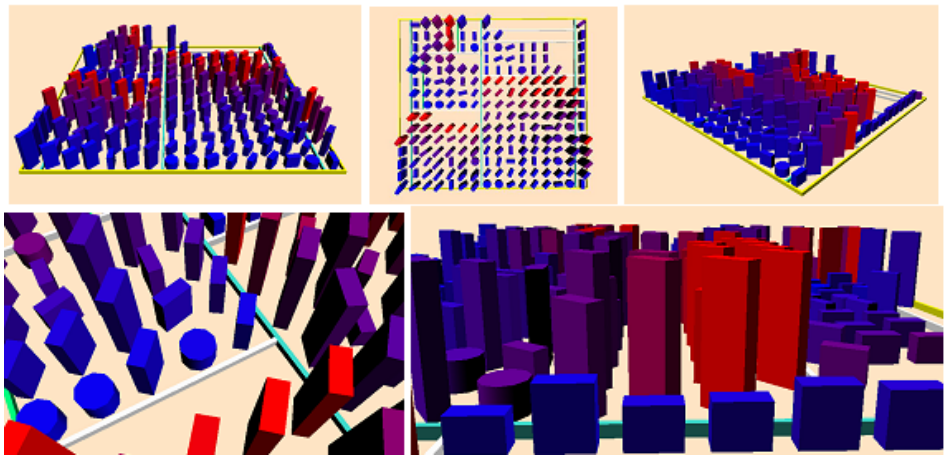


Figure 5. Exemples de différentes vues d'un système (Mouvement de caméra) [16].

Afin d'améliorer les aspects de navigabilité et de perception dans VERSO, nous avons ajouté diverses façons d'afficher les noms des éléments. Nous avons ajouté une liste défilante contenant la hiérarchie des classes du système. Nous avons ajouté également la possibilité d'afficher les détails d'une entité. Ces deux mécanismes sont déclenchés en fonction des mouvements de la souris. Quand la souris s'arrête sur une entité, la liste défile jusqu'au nom de l'entité et la fiche de détails est mise à jour également (Figure 6).

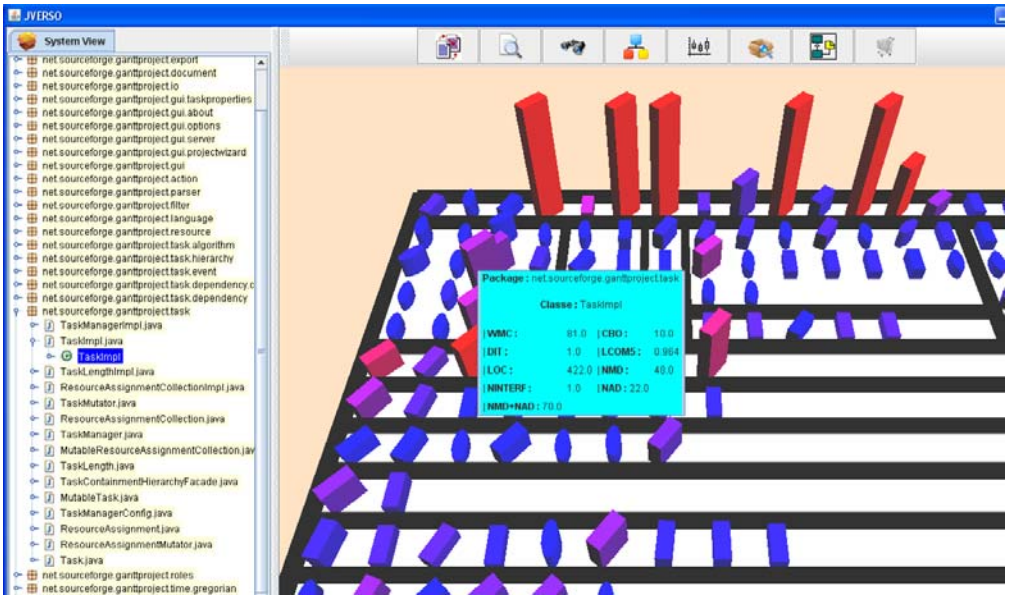


Figure 6. Nommage des éléments.

3.3.2 Module de spécification de filtres

Comme nous l'avons vu dans la section précédente, l'outil VERSO est destiné à la visualisation de logiciels complexes et de grande taille. Ceci nous permet de faire des requêtes sur des systèmes aussi volumineux que celui montré à la Figure 7 extraite de Langelier [16]. Cette figure représente JDK1.5 (Java Developement Kit) avec le placement du Treemap. Plus de 5000 classes sont affichées dans ce logiciel. L'exploration d'un logiciel de cette envergure n'est pas possible en utilisant uniquement des outils de visualisation. Pour réaliser des tâches complexes, il est nécessaire d'avoir un module permettant d'interroger les données.

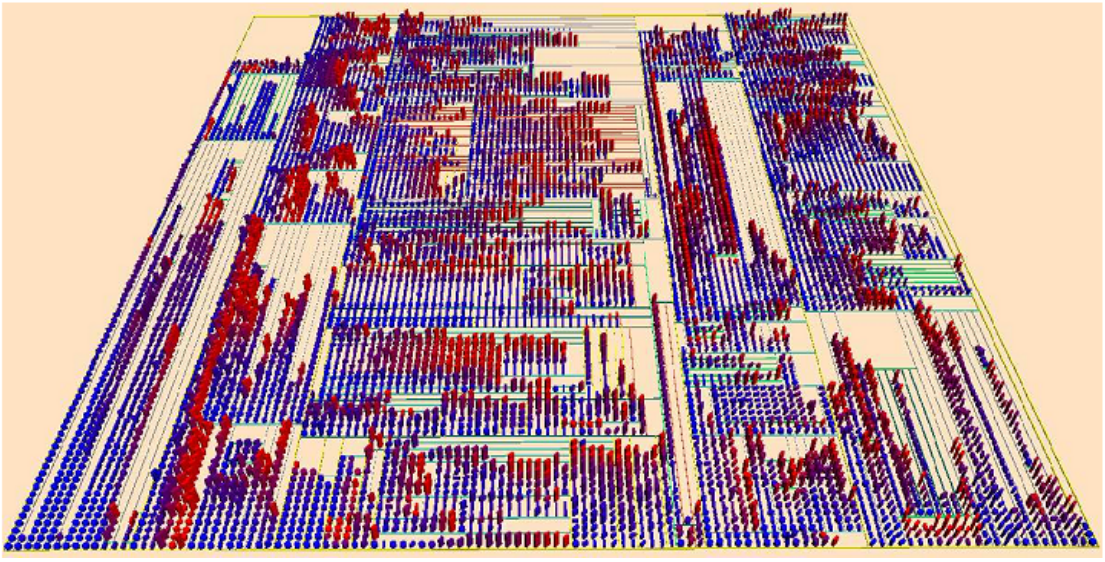
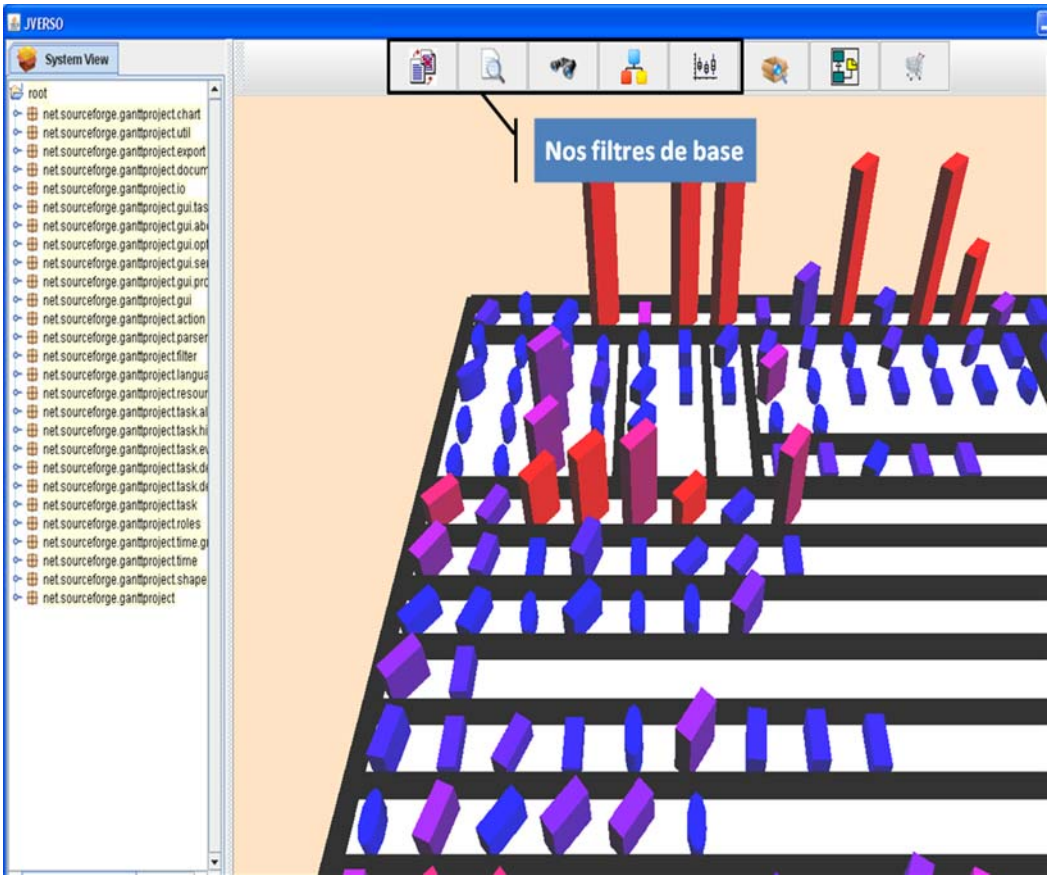


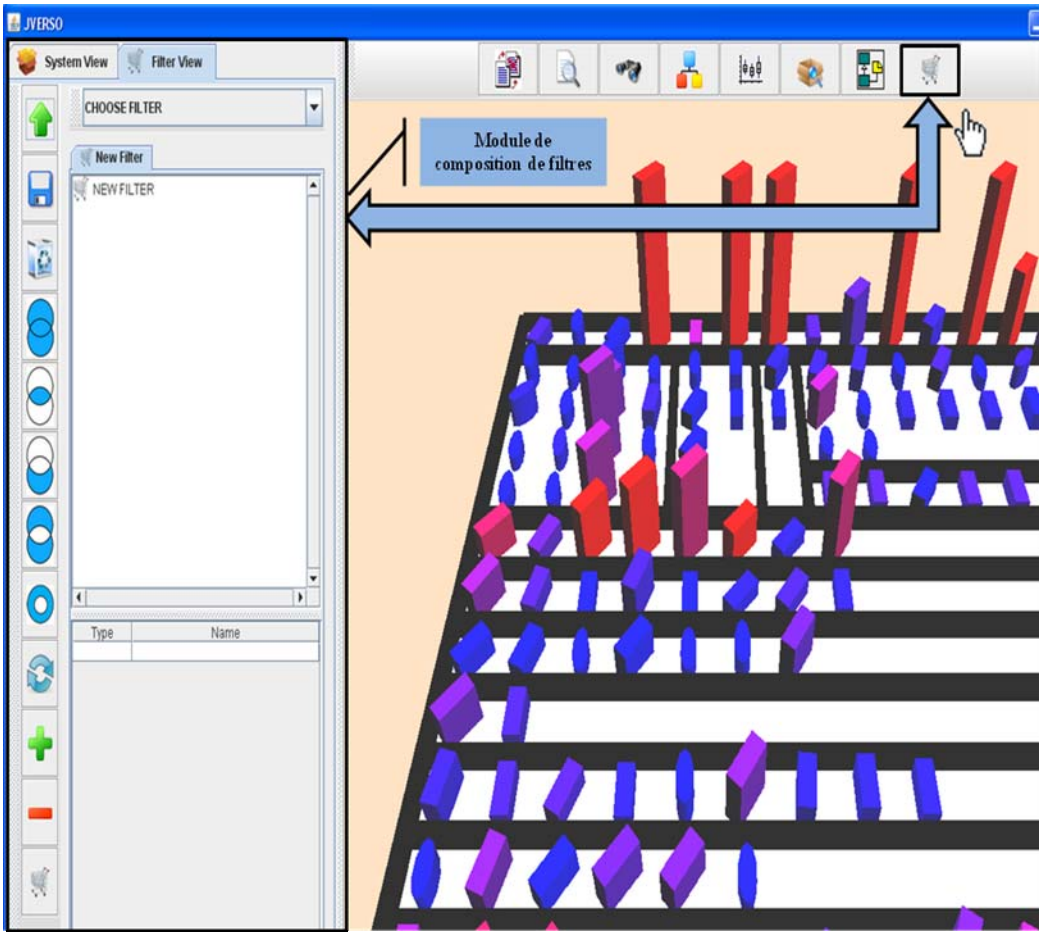
Figure 7. Visualisation de JDK 1.5 à l'aide de Treemap [16].

Dhambri [14] a proposé une extension de l'outil VERSO pour améliorer l'analyse de code. Cette extension a été conçue spécifiquement pour la détection des défauts de conception. Nous nous basons sur ce travail pour proposer une approche plus générique qui permet de résoudre un éventail plus large de problèmes tels que la localisation de fonctionnalités, la détection de clones, la compréhension des dépendances, etc. Ceci est fait par l'intégration d'un module d'interrogation de code qui permet de réduire l'espace de recherche pour permettre à des utilisateurs d'effectuer les tâches de maintenance sur des

logiciels volumineux. Ce module est composé de deux parties principales : une partie permettant de spécifier des filtres de base et une autre pour la composition de ces filtres. Lors de la réalisation de nos filtres de base, nous avons considéré quatre principales catégories d'information : sémantique, relationnelle, architecturale, et quantitative. Le fonctionnement de chaque filtre sera détaillé dans le chapitre 4. Comme le montre la Figure 8 (a), nous avons cinq différents filtres.



(a). Module de base



(b). Module de composition de filtres

Figure 8. Module d'interrogation de code.

La composition de filtres est gérée par un module de formulation interactive de requêtes (Figure 8 (b)). Ce module aide l'analyste à construire des requêtes complexes à partir de différents filtres de base ainsi que d'opérateurs de composition. Pour les filtres de base et les requêtes composées, quand une classe est sélectionnée, elle conserve sa couleur d'origine (Figure 9, à gauche). En revanche, les classes qui ne sont pas choisies sont toujours affichées, mais avec la couleur grise. Le module de visualisation permet aussi d'afficher les méthodes. Quand un filtre est appliqué à des méthodes, des boîtes de classes contenant les méthodes choisies deviennent transparentes et leurs méthodes apparaissent également comme des boîtes 3D (Figure 9, à droite).

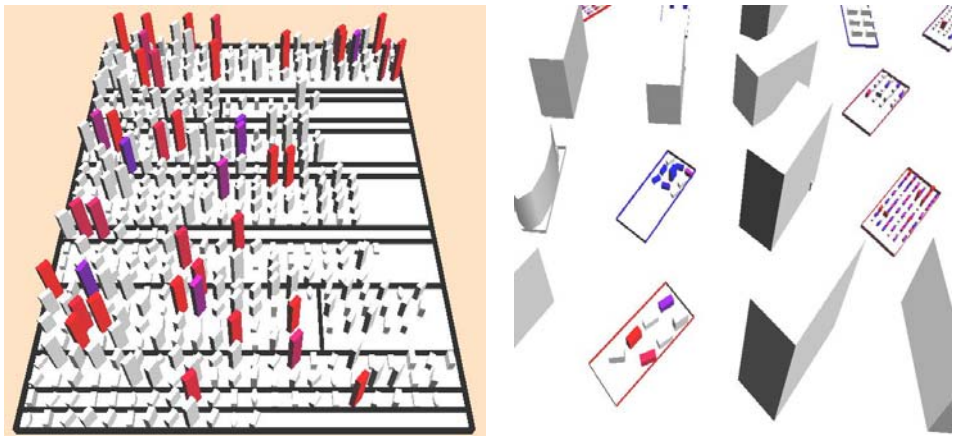


Figure 9. Résultat d'un filtre : Visualisation des classes (Gauche), Visualisation des méthodes (Droite).

Dans le module de composition, en plus de la visualisation 3D, il est également possible d'afficher les résultats sous forme de listes d'éléments. Les éléments sont classés par pertinence selon le type de filtres utilisés. Par exemple, pour le filtre de recherche par mots clés, les résultats seront classés par ordre décroissant de pertinence par rapport à la requête entrée par l'utilisateur.

À la fin du processus de recherche, l'analyste a également la possibilité de générer le diagramme de classe correspondant à l'ensemble des éléments du résultat de la recherche. Ceci permet, entre autres, de voir les relations entre ces éléments.

3.4 Conclusion

Nous avons présenté dans ce chapitre les principes de base ainsi que les modules qui composent notre environnement interactif. Notre environnement est composé de deux modules complémentaires : un module de visualisation et un autre de spécification de requêtes. Ce dernier contient à son tour deux sous-modules : un premier de spécification de filtres de base qui sont exécutés séparément et un deuxième de composition permettant de combiner plusieurs types de filtres. Dans le chapitre suivant, nous allons donner des détails concernant le fonctionnement et la réalisation de nos filtres de base.

Chapitre 4.

Les filtres de base (Requêtes simples)

4.1 Introduction

Les éléments de base de notre environnement d'interrogation sont les filtres. Après avoir examiné la littérature sur l'analyse et la compréhension de logiciels, nous avons identifié trois familles de filtres actuellement utilisées : (1) filtres basés sur des requêtes en langage naturel, (2) filtres basés sur la structure du code (héritage, appels entre méthodes ou classes, etc.), et (3) filtres basés sur les métriques logicielles. Comme l'interactivité est un élément essentiel dans notre environnement, nous avons ajouté un quatrième type basé sur les actions effectuées par l'utilisateur (ajout et suppression d'éléments). Dans cette section, nous allons décrire le principe de chaque filtre utilisé dans notre approche et nous donnons pour chacun un exemple d'utilisation. Dans la partie 4.2, nous décrivons nos filtres basés sur des requêtes en langage naturel (recherche dans les noms, recherche par mots clés et recherche de similarité entre les éléments). Dans la partie 4.3, nous présentons les filtres structurels qui se basent sur les informations extraites à partir d'une analyse statique du code. Ensuite, dans la partie 4.4, nous abordons les filtres quantitatifs qui se basent sur les métriques logicielles. Enfin, dans la 4.5, nous détaillons les filtres d'interaction qui permettent d'intégrer les actions de l'utilisateur faites à l'aide de notre visualisation interactive.

4.2 Les filtres basés sur les requêtes en langage naturel

Dans cette partie, nous décrivons nos filtres qui sont basés sur des requêtes en langage naturel. Dans la partie 4.2.1, nous donnons les principes de base de la recherche d'information ainsi que les principales phases dans ce processus de recherche que sont l'indexation et la recherche.

4.2.1 Principe de base de la recherche d'information

i. Définition de la recherche d'information

La recherche d'information (RI) est une branche de l'informatique qui s'intéresse à l'acquisition, l'organisation, le stockage, la recherche et la sélection d'information [15]. L'intérêt de la recherche d'information est de sélectionner dans une collection des données les informations pertinentes répondant aux besoins d'un utilisateur. De manière générale, le processus de recherche d'information fait intervenir trois entités : un utilisateur, une collection de documents et un système de recherche d'information (SRI). L'utilisateur, ayant un besoin en information, interroge le système de RI en lui soumettant une requête q, souvent sous la forme d'une liste des mots clés, et s'attend à recevoir comme réponse une liste des documents de la collection répondant potentiellement à cette requête, qu'on appelle documents pertinents. Les SRI se distinguent entre eux par la façon d'interpréter et de représenter les documents et les requêtes. L'utilisation des techniques sophistiquées de recherche d'information consistent à passer par plusieurs étapes comme le prétraitement du corpus de documents ainsi que l'indexation. La figure ci-dessous décrit le fonctionnement d'un SRI :

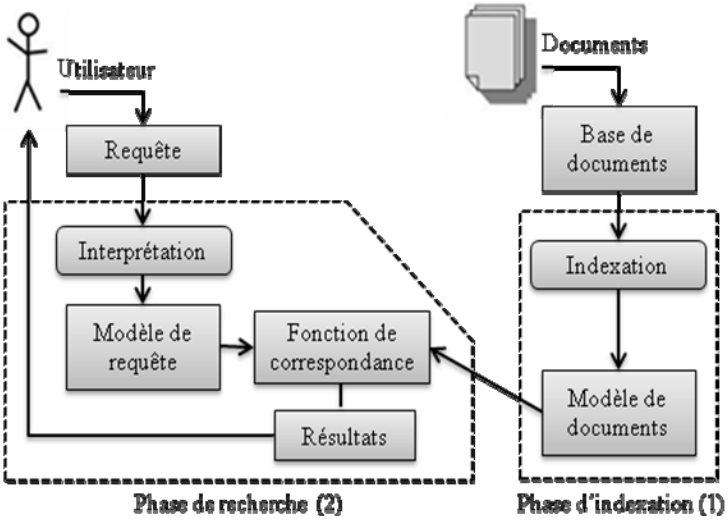


Figure 10. Fonctionnement d'un système de recherche d'information (SRI).

Dans la partie suivante, nous décrirons les deux phases importantes du processus de recherche d'information : (1) l'indexation et (2) la recherche.

ii. Phase d'indexation

L'indexation consiste traditionnellement à dégager l'ensemble des mots clés qui décrivent le document. Un poids est associé à chaque terme pour déterminer son importance dans le document ainsi que son pouvoir discriminant dans la collection de documents résultats. L'index est un ensemble des termes pondérés qui reflètent le contenu du document. L'indexation textuelle repose sur les techniques de *traitement automatique des langues (TAL)*. Dans notre cas, un document représente le code source de chaque méthode ou classe dans un système logiciel. Afin d'indexer le contenu des classes et des méthodes, il y a une phase de prétraitement décrite dans la Figure 11 ci-dessous.

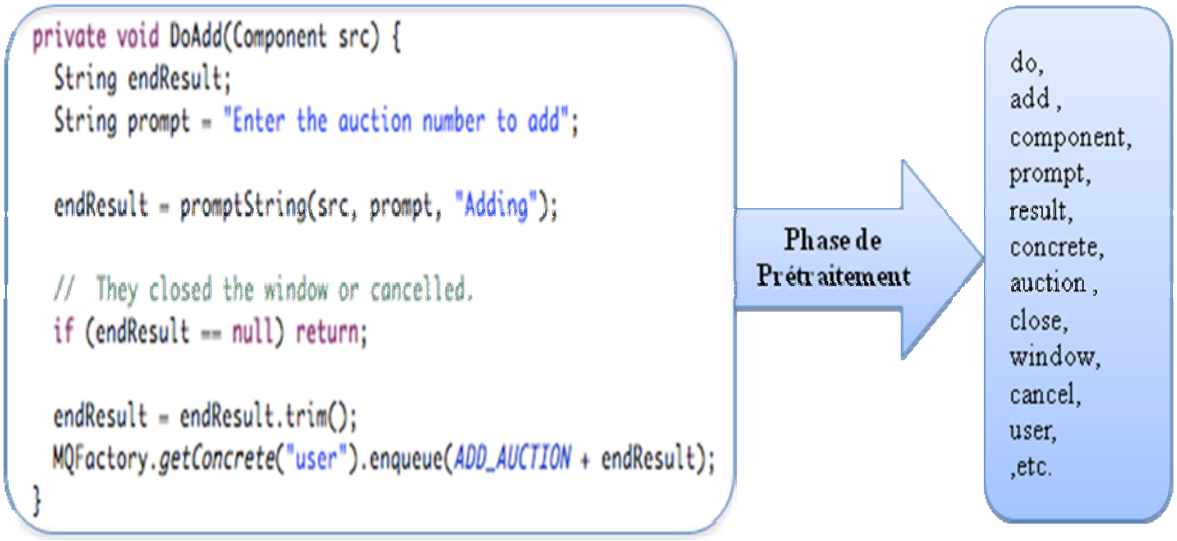


Figure 11. Prétraitement d'un exemple de document : la méthode *DoAdd(Component)* dans le système JBidWatcher v1.0pre6 [12].

La réalisation de cette phase de prétraitement est gérée en plusieurs étapes que nous décrivons ci après:

- **Exploration du code :** Ceci consiste en la création d'un analyseur (*parseur*) qui permet d'explorer les fichiers java d'un code source afin d'extraire le contenu de chacune

des classes et chacune des méthodes. Ce contenu représente le corps d'une méthode ou d'une classe (signature, variables, commentaires, instructions, etc.). L'extraction du contenu nous permet de construire la base des méthodes et des classes.

- **Suppression des caractères spéciaux :** Pour chaque document qui existe dans notre base de méthodes ou bien de classes, cette sous-étape consiste à analyser le fichier contenant le code et à éliminer l'ensemble des caractères spéciaux de leur contenu, par exemple (, ; . { } [ ] ! = etc.).

- **Décomposition des termes :** Dans cette sous étape, nous nous intéressons au traitement de chaque mot (*token*) lu. Ce traitement consiste à vérifier si le token en cours est composé (c.-à-d., voir s'il contient des lettres majuscules), car parfois, les noms de méthodes, de variables ou bien de classes peuvent être composés. Par exemple, le nom de la méthode « setUndoRecord » est composé de 3 termes set, undo et record. Dans ce cas, on ajoute les mots simples « set, undo, record », ainsi que les mots composés « setUndoRecord » dans la liste des tokens existants.

- **Élimination des mots vides :** Après l'étape de traitement des tokens, nous filtrons les termes en utilisant une anti-liste (*Stop List*). Cette liste contient un ensemble de termes qui sont considérés comme des mots vides, en anglais et en français, ainsi qu'une liste de mots clés du langage et de la bibliothèque java, comme string, public, void , etc.). Comme la plupart des méthodes ou bien des classes peuvent contenir ces termes, on les considère comme des mots vides, car ils ne permettent pas discriminer un élément d'un autre.

- **Lemmatisation ou radicalisation :** La lemmatisation ou radicalisation est la réduction d'une forme fléchie en sa forme canonique, par exemple le terme "adding" devient "add".

- **Indexation :** Finalement, après la préparation du contenu ainsi que le prétraitement effectué sur le corpus de données existantes, l'étape suivante correspond au processus d'indexation. L'étape d'indexation permet de choisir les termes représentant les documents. Cette étape d'indexation est effectuée une seule fois pour être utilisée dans l'étape d'interrogation (requêtes). Dans les différents SRI, un texte (document) est décrit par un ensemble de mots-clés représentatifs appelés termes d'indexation. Un terme d'indexation est un élément dont la sémantique décrit (partiellement) le contenu d'un texte et fournit une clé

d'accès à ce texte. Un poids est généralement associé à chaque terme d'indexation pour refléter son importance dans la représentation du texte et son pouvoir discriminant. Cette étape consiste le plus souvent à pondérer l'ensemble des termes dans un document (classe ou bien méthode) selon l'une des deux techniques : l'indexation avec *TF-IDF* et *l'indexation sémantique latente (LSI)*.

a) TF-IDF

TF-IDF [9] est une technique de pondération utilisée dans le domaine de la recherche d'information. Elle permet d'évaluer l'importance d'un terme dans un document ainsi que dans toute la collection.

TF signifie "*Term Frequency*" et *IDF* signifie "*Inverse Document Frequency*". TF est une mesure qui a un rapport avec l'importance d'un terme dans un document. En général, cette valeur est déterminée par la fréquence du terme dans le document. Par contre, IDF est une mesure permettant de déterminer l'aspect discriminatoire d'un terme. Un terme qui a une valeur de TF*IDF élevée doit être à la fois important dans ce document et apparaît peu dans les autres documents. C'est le cas des termes correspondant à des caractéristiques importantes et uniques d'un document. Le poids d'un terme dans un document sera calculé comme suit :

$$Poids\ (t_i\ ,\ d_j) = TF\ (t_i\ ,\ d_j)\ *\ IDF\ (t_i)$$

Avec :

$TF(t_i, d_j) = \frac{N_{i,j}}{NbTerm(d_j)}$ avec $N_{i,j}$ est le nombre d'apparition du terme t_i dans le document d_j et $nbTerm(d_j)$ est le nombre total des termes dans le document d_j.

$IDF(t_i) = \log \frac{|D|}{|\{d_j : t_i \in d_j\}|}$ avec : $|D|$ est le nombre total de documents dans la collection et $|\{d_j : t_i \in d_j\}|$ est le nombre de documents de la collection contenant le terme t_i.

b) Indexation sémantique latente (LSI)

L'indexation sémantique latente [15] est une technique qui permet d'extraire et de représenter la similitude sémantique entre les termes ainsi que les concepts contenus dans une collection de données non structurées. *LSI* est basée sur le principe que les mots qui sont utilisés dans les mêmes contextes ont tendance à avoir des significations semblables. Cette méthode part d'une technique mathématique nommée *décomposition en valeurs singulières (SVD)* [15]. Cette technique permet de résoudre le problème de synonymie et de polysémie par la recherche des relations sémantiques entre les termes. La synonymie signifie que deux termes ont le même sens, mais ont des formes différentes, par contre la polysémie signifie qu'un même terme peut avoir plusieurs sens selon le contexte dans lequel il apparaît.

L'indexation sémantique latente se déroule en deux étapes. La première étape consiste à représenter le texte sous forme d'une matrice. Les lignes décrivent l'ensemble de termes dans le corpus, les colonnes sont les contextes qui existent dans le corpus et les cellules représentent les fréquences des termes dans les documents. L'analyse de la structure sémantique latente commence par cette matrice de termes/documents. Cette matrice est alors analysée par la décomposition en valeurs singulières afin d'extraire de notre modèle la structure sémantique latente. La décomposition en valeurs singulières est elle-même basée sur la factorisation des matrices rectangulaires complexes. Afin d'implémenter cette technique, nous avons utilisé une librairie existante nommée SVDLIBC proposée par Berry et al. [13].

iii. Phase de recherche

La procédure de recherche d'information se base sur une comparaison entre les index des documents et celui de la requête.

Le modèle vectoriel [9] représente les documents et les requêtes par des vecteurs dans un espace à *n* dimensions, les dimensions étant constituées par les termes du vocabulaire d'indexation. La fonction de correspondance mesure la similarité entre le vecteur requête et

les vecteurs documents ou bien document/documents. Une mesure classique est le cosinus de l'angle formé par les deux vecteurs. La formule utilisée est la suivante :

$$Similarity(\vec{d_i}, \vec{d_j}) = \frac{\vec{d_i}.\vec{d_j}}{\| \vec{d_i} \| . \| \vec{d_j} \|}$$

Cette formule peut servir à calculer la similarité entre deux documents ou bien entre une requête et un document. Cette mesure peut avoir une valeur entre 0 et 1, tel que:

$$\vec{d_i}.\vec{d_j} = \sum_{k=1}^{n} poids(t_k, \vec{d_i}).poids(t_k, \vec{d_j}), \qquad \| \vec{d_i} \| . \| \vec{d_j} \| = \sum_{k=1}^{n} \sqrt{(poids(t_k, \vec{d_i}))^2 . (poids(t_k, \vec{d_j}))^2}$$

4.2.2 Filtre de recherche dans les noms

La recherche dans les noms, nommée *NAME_SEARCH,* est le filtre de base qui consiste à effectuer une recherche simple d'une chaîne de caractères (*TXT_QUERY*) dans les noms de classes ou de méthodes. Si la chaîne est présente dans le nom d'un élément, cet élément est inclus dans le résultat. Ce filtre peut être appliqué à tous les éléments du système (*SYSTEM*) ou à ceux déjà choisis par des filtres précédents (*CURRENT*). Ce filtre est spécifié en utilisant l'interface utilisateur (Figure 12) ou par la syntaxe suivante :

NAME_SEARCH (**QUERY**: TXT_QUERY, **LEVEL**: CLASS|METHOD|BOTH, SYSTEM|CURRENT).

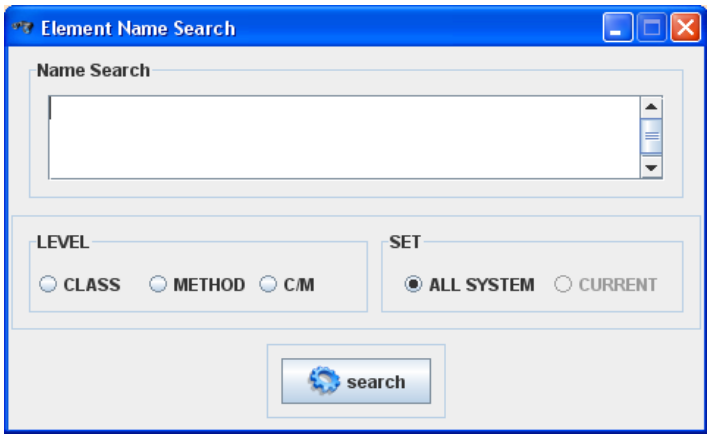


Figure 12. Interface utilisateur de filtre *NAME_SEARCH*.

Par exemple, on peut utiliser ce filtre pour chercher les classes contenant le terme «options» dans leurs noms. Dans GanttProject v1.10.2 [11] deux classes sont retournées: GanttOptions et CSVOptions. Il est à souligner que ce premier filtre n'utilise pas les techniques d'indexation et de recherche décrites dans la section précédente.

4.2.3 Filtre de recherche par mots clés

La recherche par mots-clés, nommée *KEYWORD_SEARCH*, est utilisée pour déterminer à quel point, les éléments d'un code source (des classes ou des méthodes) sont pertinents par rapport à une requête en langage naturel saisie par l'utilisateur (*TXT_QUERY*). Pour déterminer cette pertinence, nous utilisons la mesure cosinus pour calculer le degré de similarité entre le vecteur de termes de la requête et celui des classes ou bien des méthodes. Ce filtre peut s'appliquer sur tous les éléments du système (*SYSTEM*) ou bien sur les résultats de filtres précédents (*CURRENT*).

Une des deux techniques de recherche documentaire sera utilisée selon le besoin de l'analyste du code, l'Indexation Sémantique Latente (*COSINE_LSI*) ou bien TF-IDF (*COSINE_TF-IDF*). Une valeur seuil peut être spécifiée pour déterminer le score minimal exigé (*VALUE*) afin de considérer les éléments comme pertinents par rapport à la requête et les inclure dans le résultat du filtre. Ce filtre est aussi décrit via l'interface de la Figure 13 ou selon la syntaxe suivante:

KEYWORD_SEARCH (**QUERY**: TXT_QUERY, **METHOD**: COSINE_LSI|COSINE_TF_IDF, **THRESHOLD**: VALUE, **LEVEL**: CLASS|METHOD, SYSTEM|CURRENT)

Ce filtre est plus sophistiqué que celui de la recherche dans les noms, car il prend en considération l'aspect sémantique en analysant l'ensemble des termes qui décrivent une méthode ou une classe.

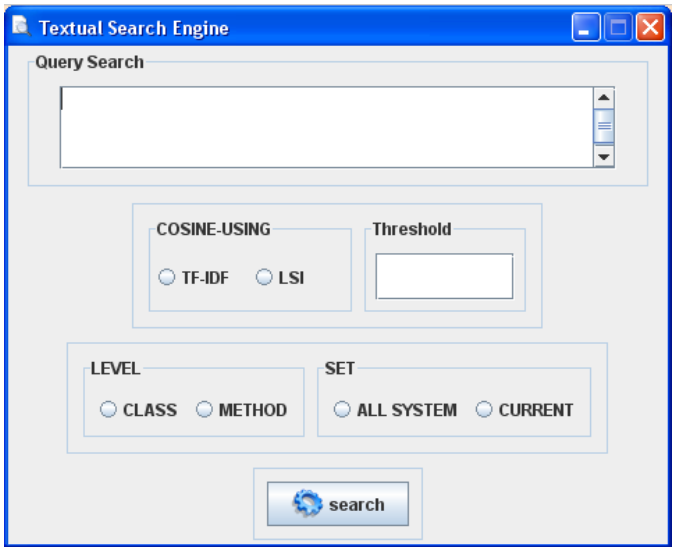


Figure 13. Interface utilisateur de filtre *KEYWORD_SEARCH*.

Ce filtre est utilisé pour la compréhension du code et plus particulièrement pour la localisation des fonctionnalités dans le code. Il permet d'identifier les éléments du code qui implémentent une fonctionnalité particulière dans un programme de grande taille. La description du problème peut être exprimée en langage naturel en se basant sur la force de techniques de recherche d'information. Par exemple Marcus [22] a proposé une approche de compréhension de code basée sur la technique de recherche d'information *LSI* (*latent semantic indexing*). Cette approche consiste à identifier les entités du code source qui implémentent une fonctionnalité particulière décrite par une requête en langage naturel.

4.2.4 Filtre de recherche de similarité entre les éléments

La recherche de similarité entre les éléments, nommée *SIMILARITY_SEARCH*, est utilisée pour déterminer le degré de similarité sémantique entre un élément du code avec tout le système (*SYSTEM*) ou bien le résultat des filtres précédents (*CURRENT*). Les éléments du code peuvent être des classes ou des méthodes. En utilisant ce filtre, l'utilisateur va entrer le nom de l'élément (*TXT_QUERY*) à utiliser. Ensuite, pour déterminer la liste des éléments semblables, nous utilisons la mesure cosinus pour calculer le degré de similarité entre le vecteur de termes (index) de l'élément choisi et celui des classes ou bien des méthodes. Une des deux techniques de recherche documentaire sera

utilisée, soit l'indexation sémantique latente (*COSINE_LSI*), soit TF-IDF (*COSINE_TF-IDF*).

Une valeur de seuil peut être spécifiée pour déterminer le score minimal exigé (*VALUE*) afin de considérer qu'un élément est similaire à l'élément sélectionné. Ce filtre sera décrit selon la syntaxe suivante:

SIMILARITY_SEARCH (**ELT_NAME**: TXT_QUERY, **ALGORITHM**: COSINE_LSI|COSINE_TF-IDF, **THRESHOLD**: VALUE, **LEVEL**: CLASS|METHOD, SYSTEM|CURRENT)

L'interface utilisateur permettant de spécifier ce filtre est montrée dans la Figure 14.

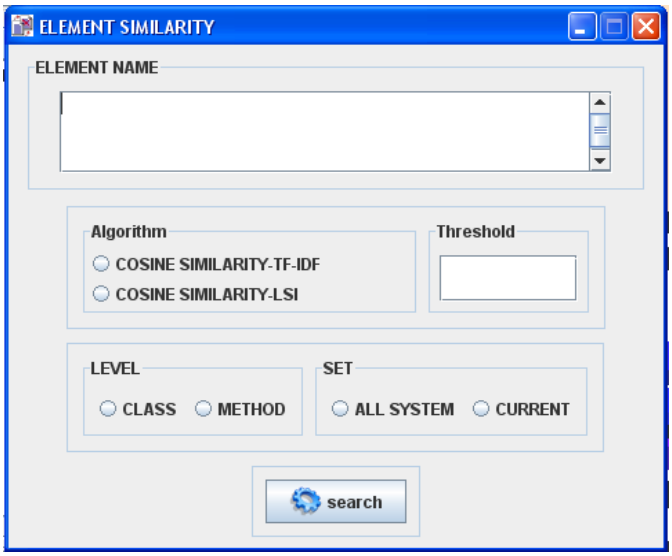


Figure 14. Interface utilisateur de filtre *SIMILARITY_SEARCH*.

Ce filtre peut servir dans plusieurs problèmes tels que la détection des clones. Les clones représentent les éléments du code qui dupliquent le même comportement. Ce problème est dû à l'action de copier-coller faite par les développeurs. Dans le cas où deux éléments ont un degré de similarité proche de 1, on peut conclure qu'ils partagent les mêmes termes.

4.3 Le filtre structurel

Dans notre environnement, les dépendances entre les éléments de code (méthodes - méthode, méthode-classe, classes-classes, etc.) sont extraites par l'analyse statique du code. Ces dépendances comprennent les invocations de méthodes, l'héritage, l'inclusion, etc. Le filtre structurel, nommé *STRUCTURAL*, est utilisé pour collecter les éléments liés à un élément donné par une relation de dépendance particulière. Dans le cas des appels entre classes ou bien des invocations de méthodes, le système est considéré comme un graphe où les nœuds correspondent aux éléments (classes ou méthodes) et les arcs représentent les relations entre ces éléments (Calls ou Called By). Un exemple de question à laquelle on peut répondre par ce filtre est : quelles sont les méthodes qui ont reçu des appels d'une méthode donnée? Dans ce cas, nous utilisons la relation « Called By » qui prend en paramètre le nom de la méthode à utiliser. Ce filtre peut être utilisé sans donner comme paramètre un élément particulier (option ALL). Par exemple, il est possible de chercher l'ensemble des classes qui implémentent au moins une interface. Dans cette situation, il recueille toutes les classes en utilisant l'option ALL avec la relation « Implemented Interface ». L'interface, qui décrit ce filtre, est présentée dans la Figure 15. La syntaxe est la suivante :

STRUCTURAL (**ELEMENT** : NAME|ALL, **TYPE** : CLASS|METHOD, **RELATION**: LIST_OF_RELATIONSHIP)

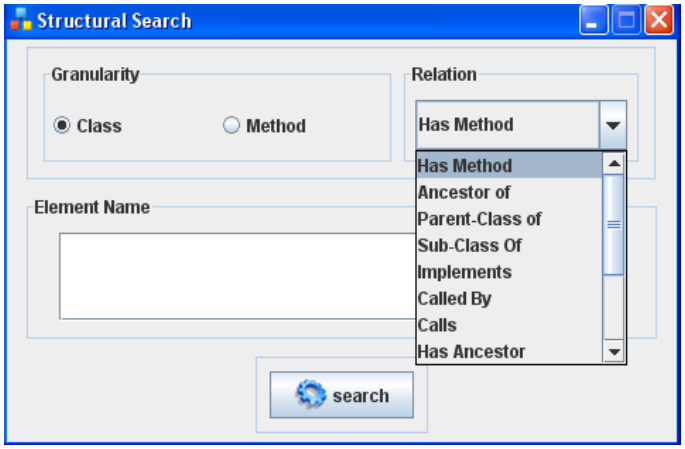


Figure 15. Interface utilisateur de filtre *STRUCTURAL*.

La liste des relations à utiliser dans le filtre structurel est donnée dans le Tableau 1.

.TYPE	RELATIONS	DESCRIPTION
CLASSE	Has Method	Toutes les classes ayant une certaine méthode.
	Ancestor Of	Les ancêtres d'une certaine classe.
	Parent Class Of	Les parents d'une classe choisie.
	Sub Class Of	Les sous-classes d'une classe choisie.
	Implements	Les classes qui implémentent une interface choisie.
	Called By	Les classes appelées par une classe choisie.
	Calls	Les classes appelant une classe choisie.
	Has Ancestor	Toutes les classes ayant des ancêtres.
	Has Parent	Toutes les classes ayant des parents.
	Has Sub Class	Toutes les classes ayant des sous-classes.
	Implemented Interface	Toutes les classes qui implémentent une interface. Les résultats seront triés par rapport au nombre d'interfaces implémentées par chaque classe.
	Are called	Toutes les classes qui sont appelées. Les résultats seront triés par rapport au nombre d'appels reçus.
	Has Calls	Toutes les classes qui appellent d'autres classes. Les résultats seront triés par rapport au nombre d'appels émis.
MÉTHODE	Has Return Type	Toutes les méthodes ayant un type de retour particulier.
	Called By	Les méthodes appelées par une méthode particulière.
	Calls	Les méthodes appelant une méthode donnée.

Tableau 1. La liste des relations utilisée dans le filtre *STRUCTURAL*.

Pour montrer l'utilité de ce type de filtres, prenons le cas d'un développeur qui veut effectuer une tâche de maintenance au niveau de la signature d'une méthode (modification de type, ajout ou suppression de paramètre(s), etc.). Dans ce cas, les méthodes qui appellent cette dernière vont être affectées. Avant la réalisation de ces changements, le développeur pourra utiliser le filtre *STRUCTURAL* afin de lister l'ensemble de méthodes susceptibles d'appeler cette méthode. Il doit spécifier la relation *Calls* en donnant comme paramètre le nom de la méthode à changer.

4.4 Le filtre quantitatif

Les métriques logicielles sont des outils puissants pour synthétiser les propriétés des éléments du code (la taille, la complexité, l'héritage, le couplage et la cohésion). Ils sont utilisés dans de nombreuses tâches de maintenance ou bien pour évaluer la qualité d'un système. L'importance des métriques logicielles a motivé l'inclusion d'un filtre quantitatif dans notre environnement d'interrogation. Le filtre quantitatif, appelé aussi filtre des métriques (*QUANTITATIVE*), permet de sélectionner, pour une métrique particulière (*METRIC_NAME*), un ensemble d'éléments de code dont la valeur est dans un intervalle donné. L'intervalle peut être précisé par un seuil qui détermine la limite inférieure ou supérieure (*HIGHER_THAN* ou *LOWER_THAN*). Alternativement, il peut être défini relativement à la distribution des valeurs pour l'ensemble des éléments à filtrer que ce soit tous les éléments du système (*SYSTEM*) ou bien les résultats d'un filtre précédent (*CURRENT*). Dans ce cas, nous utilisons la technique de *boîte à moustache* (*RANGE_BOXPLOT*). La boîte à moustache permet de partitionner les valeurs d'une métrique pour un ensemble d'éléments en cinq tranches comme le montre la Figure 16.

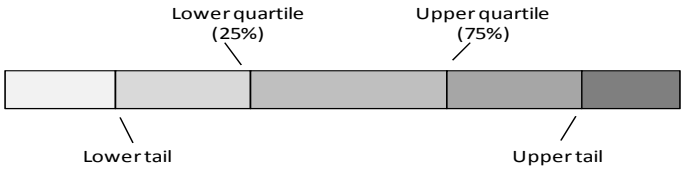


Figure 16. Définition des intervalles de valeurs d'une métrique avec une boîte à moustache (*box plot*).

La liste de métriques pour chaque type d'éléments (classe et méthode) qui sont disponibles dans notre environnement est décrite dans le Tableau 2.

TYPE	METRIQUES	DESCRIPTION
CLASSE	CBO	Couplage entre les objets.
	WMC	La somme de la complexité de méthodes dans une classe.
	DIT	Profondeur dans l'arbre d'héritage.
	LOC	Nombre de lignes de code.
	NINTERF	Nombre d'interfaces implémentées.
	NAD	Nombre d'attributs dans une classe.
	LCOM5	Degré de cohésion entre méthodes dans une classe.
MÉTHODE	CohMETHOD	Degré de cohésion d'une méthode.
	LOCMETHOD	Nombre de lignes de code dans une méthode.
	CouplingMETHOD	Degré de complexité d'une méthode.
	NMPARAM	Nombre de paramètres dans une méthode.

Tableau 2. Métriques [33] [30] utilisées dans le filtre *QUANTITATIVE*.

Le filtre quantitatif pourrait être utilisé à travers une interface graphique (Figure 17) ou textuellement en utilisant la syntaxe suivante :

QUANTITATIVE (**METRIC**: METRIC_NAME, **BOXPLOT** RANGE_SELECTED | **HIGHER_THAN** VALUE | **LOWER_THAN** VALUE**, LEVEL**: CLASS|METHOD, SYSTEM|CURRENT)

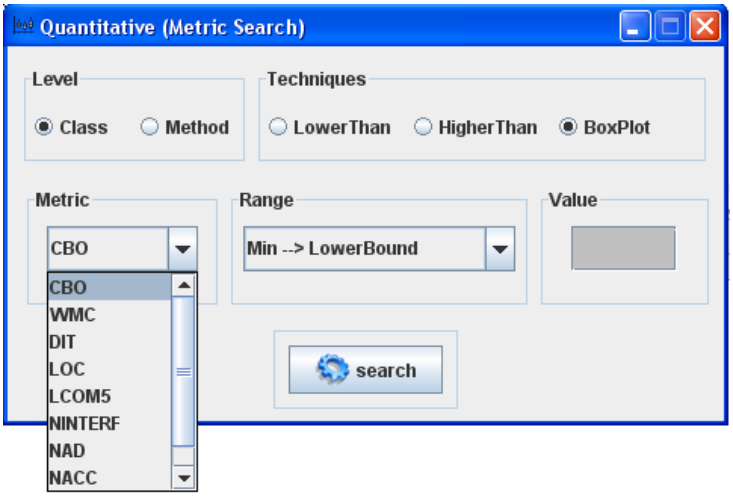


Figure 17. Interface utilisateur de filtre *QUANTITATIVE*.

Dans un système logiciel, afin de localiser les classes qui dépendent le plus des interfaces (implémentant un grand nombre d'interfaces), on peut utiliser, comme métrique *NINTERF* qui décrit le nombre d'interfaces implémentées par une classe, et comme quartile (> *UpperTail*).

4.5 Le filtre d'interactivité

Tout au long du processus d'interrogation, l'utilisateur peut combiner plusieurs filtres des types discutés dans les sections précédentes. Après l'utilisation de chaque filtre, il peut inspecter les résultats retournés et décider à quel point les résultats d'un certain filtre sont pertinents. Prenons par exemple l'une des versions du programme ArtOfIllusion qui est un logiciel open source de modélisation 3D. Si un développeur est en train de rechercher dans ce programme les classes participant à la fonction d'encodage d'images (JPEG, GIF, etc.), il peut commencer par l'utilisation de filtre *NAME_SEARCH* pour rechercher le mot « encoder » dans les noms de classes. Le résultat de ce filtre sera un ensemble de classes (JPEGEncoder, BMPEncoder, etc.). Ensuite, il peut utiliser le filtre structurel pour ajouter des classes qui sont utilisées par ces classes. Lors de l'inspection des classes retournées par la recherche dans les noms, l'utilisateur peut prendre trois décisions : (1) la majorité des classes retournées ne sont pas pertinentes, ce qui conduit à annuler le filtre et/ou d'affiner les critères de recherche, (2) toutes les classes retournées sont considérées comme

pertinentes, ce qui conduit à poursuivre l'interrogation par le filtre structurel, et (3) la majorité des classes sont pertinentes, mais certaines sont manquantes (pas de classe trouvée pour l'encodage des GIF) et/ou ne sont pas pertinentes (par exemple l'existence d'une classe nommée WidgetEncoder parmi les résultats) (Figure 18). Dans ce dernier cas, l'utilisateur peut choisir une classe, sachant qu'elle encode les images GIF, et l'ajoute manuellement aux résultats. Il peut également supprimer WidgetEncoder du résultat. Une fois ces modifications faites, il pourra continuer avec le filtre suivant.

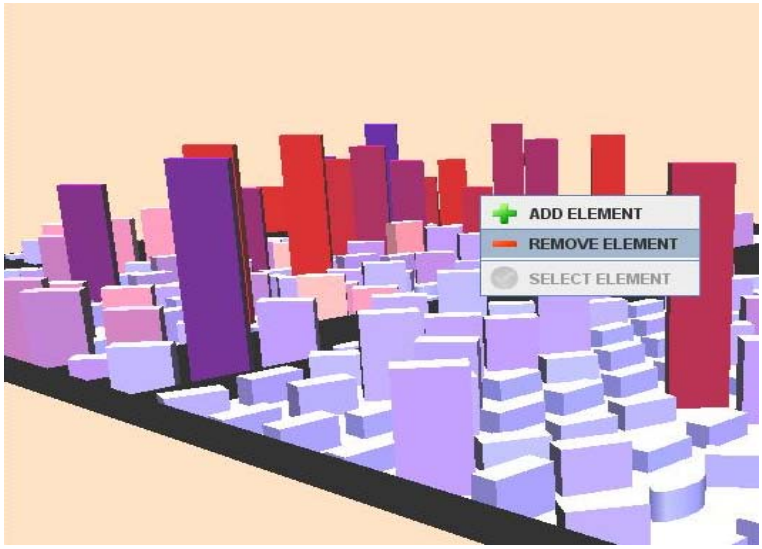


Figure 18. Actions destinées à l'interactivité avec les résultats.

4.6 Conclusion

Dans ce chapitre, nous avons détaillé nos filtres de base. Nous avons classé ces filtres selon quatre catégories : filtres linguistiques basés sur des requêtes en langage naturel, filtres structurels basés sur la structure de code, filtres quantitatifs basés sur les métriques logicielles, et filtres d'interactivité basés sur les actions de l'utilisateur durant la recherche. Dans le prochain chapitre, nous présentons la combinaison de filtres de base afin de créer des requêtes complexes et décrivons en particulier le processus itératif de construction de requêtes.

Chapitre 5.

Requêtes complexes (Composition de filtres)

5.1 Introduction

Lors de la réalisation des tâches d'analyse et de compréhension, des requêtes complexes seront exprimées en combinant des filtres de base. Les filtres de base permettent d'explorer des critères de recherche de différentes natures : linguistiques, structurels et quantitatifs. Leur combinaison permet d'accroître la précision de l'analyse et de mieux affiner la recherche. Il y a deux modes de composition de filtres : la combinaison de résultats de filtres en utilisant des opérateurs ensemblistes classiques ou bien l'entrelacement entre les filtres en utilisant des opérateurs plus sophistiqués (les itérateurs). Dans ce chapitre, nous allons décrire le principe de construction de requêtes complexes en utilisant notre module d'interrogation de code. Dans la partie 5.2, nous commençons par la description de composition de résultats de filtres. Ensuite dans 5.3, nous détaillons le principe d'entrelacement de filtres.

5.2 Composition des résultats des filtres

Nous proposons un système de formulation interactive de requêtes dont l'utilisateur est l'acteur principal. Pour formuler les requêtes, l'utilisateur dispose des différents types de filtres proposés dans le chapitre 4. En plus de l'utilisation séparée de ces filtres de base, l'utilisateur a la possibilité de composer les résultats de plusieurs filtres. Dans cette perspective, nous utilisons différents opérateurs ensemblistes pour combiner les résultats de filtres. Comme le montre la Figure 19, dans ce mode de composition, nous avons deux types des opérateurs : les opérateurs unaires (représentés en rouge) et binaires (représentés en noir).

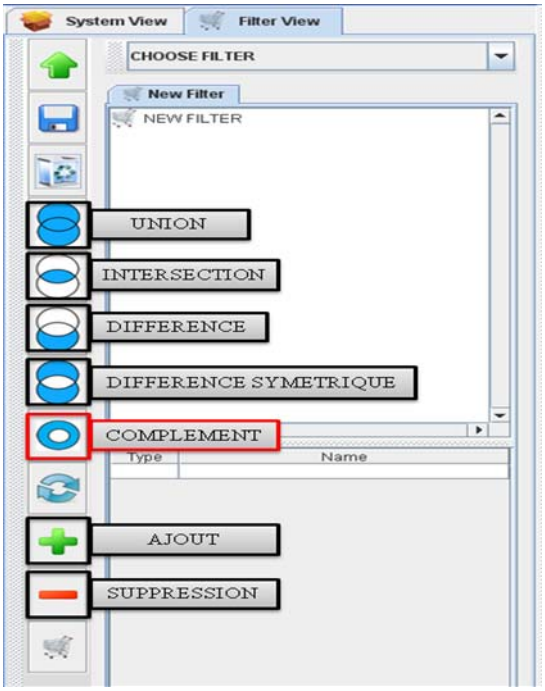


Figure 19. Les opérateurs utilisés pour la composition de résultats de filtres.

Le seul opérateur unaire utilisé est le complément d'un ensemble (nommé COMPLEMENT dans la Figure 19). Cet opérateur est utilisé pour exprimer la négation de certains critères de recherche. Il renvoie tous les éléments qui ne sont pas inclus dans le résultat d'un filtre ou bien d'une requête complexe. Par exemple, si l'on veut exclure de notre analyse l'ensemble des classes de gestion dans le système, on pourrait appliquer le filtre de recherche dans les noms, *NAME_ SEARCH*, prenant en paramètre la chaîne de caractère "*manager*". Ce filtre retourne l'ensemble des classes contenant cette chaîne dans leurs noms ou bien dans les noms de leurs méthodes. L'étape suivante consiste à appliquer l'opérateur de complément pour exclure toutes ces classes des résultats de la requête. Cela est exprimé formellement comme suit :

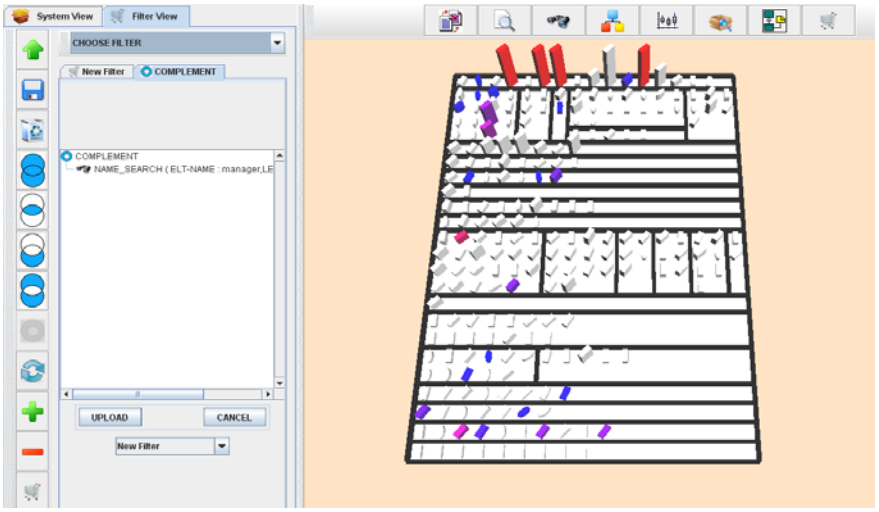
COMPLEMENT **[**

NAME_SEARCH (QUERY: "*manager*", LEVEL: BOTH, SYSTEM)

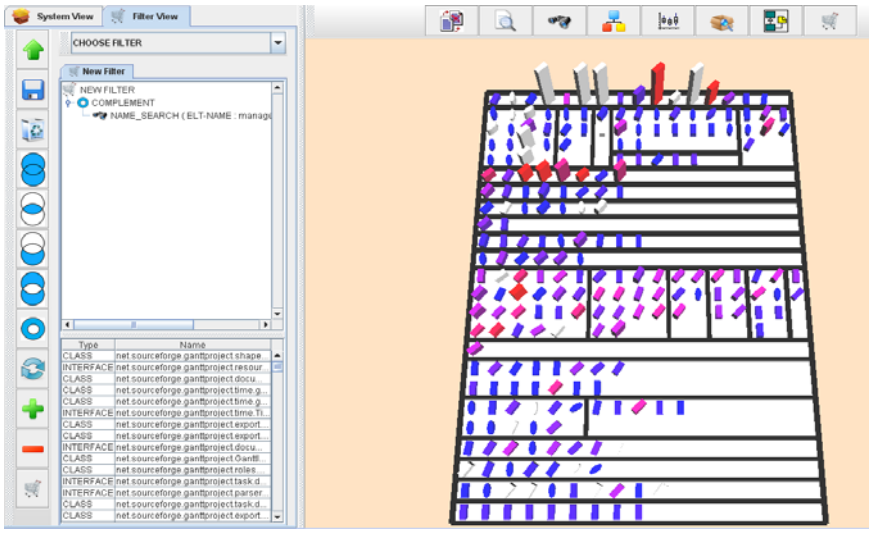
]

En utilisant notre framework, le scénario à exécuter est le suivant. Premièrement, l'utilisateur clique sur le bouton qui correspond à l'opérateur COMPLEMENT (celui

encadré en rouge dans la Figure 19). Ensuite, comme le montre la Figure 20-(a), un nouvel onglet s'affiche dans lequel l'utilisateur peut formuler sa requête. Dès qu'il termine la formulation de sa requête, il peut cliquer sur le bouton *upload* d'une part pour donner le signal d'exécution de l'opération de complément et d'autre part pour afficher les résultats dans l'onglet principal (Figure 20-(b)). Si l'on applique ce scénario dans le système GanttProject v 1.10.2 [11], on obtient le résultat montré dans la Figure 20.



(a). Exemple de Formulation d'une requête dans l'onglet COMPLEMENT



(b). Télécharger les résultats dans l'onglet principal.

Figure 20. Scénario d'utilisation de COMPLEMENT dans GanttProject v 1.10.2.

En plus de l'opérateur unaire, quatre opérateurs binaires sont fournis pour combiner les résultats des filtres. Ces opérateurs sont l'union, l'intersection, la différence, et la différence symétrique. Par exemple, supposons que nous cherchons les classes ayant une complexité très élevée et qui contiennent, dans leurs noms ou bien dans le nom de leurs méthodes, la chaîne "*manager*". Alors on pourrait appliquer le filtre quantitatif à l'aide de la métrique WMC (Weighted Method per Class), en utilisant la technique de *box plot* et en spécifiant le quartile supérieur. Ensuite, nous appliquons le filtre de recherche dans les noms avec la chaîne "*manager*". Finalement, nous combinons les deux résultats avec l'opérateur INTERSECT. Textuellement, cette requête complexe est exprimée comme suit :

QUANTITATIVE (METRIC: WMC, BOXPLOT: >= UpperTail, LEVEL: CLASS, SYSTEM)
INTERSECT
NAME_SEARCH (QUERY: "*manager*", LEVEL: CLASS, SYSTEM)

En utilisant l'interface graphique de notre environnement d'interrogation, le scénario de cette requête est montré dans la Figure 21. Les étapes à suivre sont les suivantes :

- Premièrement (1), l'analyste choisis le filtre QUANTITATIVE dans la barre des filtres.
- Ensuite (2), l'opérateur INTERSECT qui existe dans la barre des opérateurs.
- Enfin (3), choisir le deuxième filtre NAME_SEARCH.

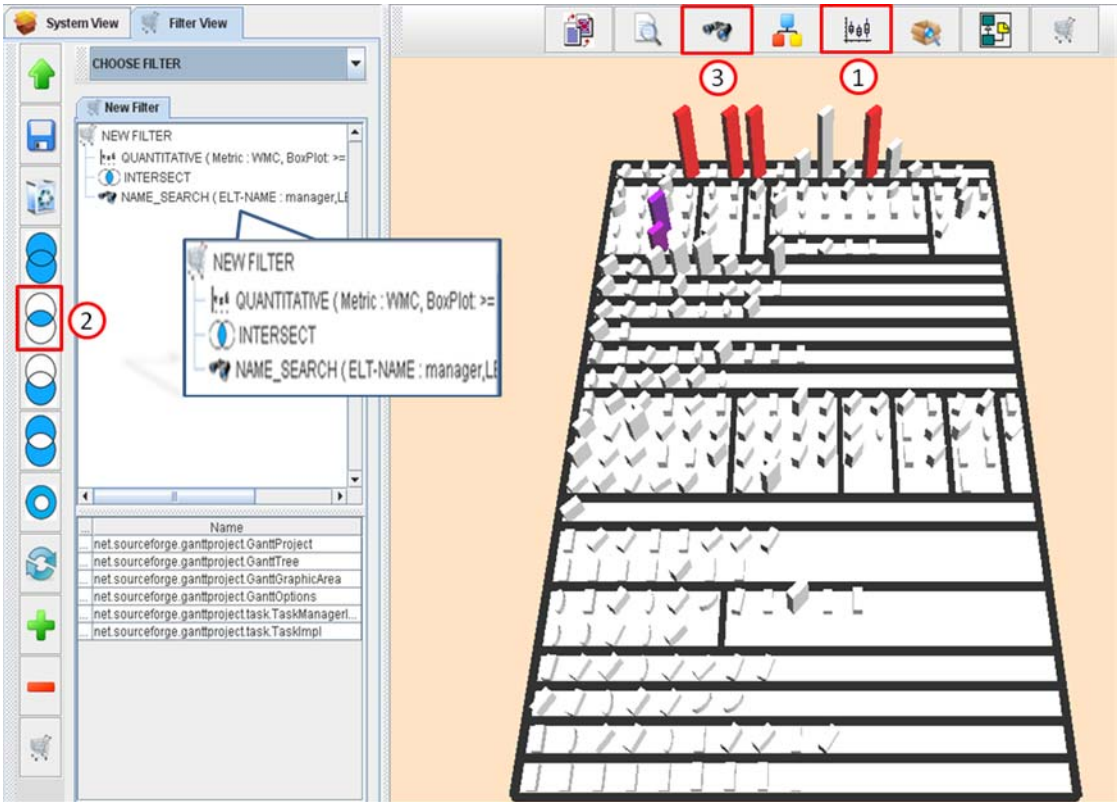


Figure 21. Scénario d'utilisation de l'opérateur binaire INTERSECT sur GanttProject v 1.10.2 [11].

Pour faciliter l'expression de requêtes et réduire l'interaction, nous avons implanté d'autres opérateurs tels que MINUS, qui permet à l'analyste de supprimer des éléments de résultat actuel. En outre, afin d'aider l'analyste à garder la trace de ses interactions, à tout moment du processus, il peut afficher une formulation textuelle de la partie déjà réalisée de la requête. Cette formulation inclut les filtres de base (y compris les sélections manuelles) et les opérateurs.

5.3 Composition par entrelacement de filtres

Ce mode de composition permet d'appliquer un filtre sur les résultats d'une requête précédente. Dans notre environnement, nous avons défini deux types d'itérateurs qui se basent sur l'imbrication de filtres; un itérateur de base et un itérateur conditionnel.

5.3.1 Itérateur de base

L'itérateur de base permet d'appliquer le filtre de structure d'une façon récursive. En effet, à partir d'un point d'entrée ou un ensemble de points d'entrée, il sélectionne les éléments liés à ce(s) point(s) d'entrée(s) en utilisant une certaine relation. Ensuite, il considère chacun des éléments sélectionnés comme point d'entrée et lui réapplique le même filtre, et ainsi de suite jusqu'à un niveau donné. Tous les éléments sélectionnés à tous les niveaux sont ajoutés au résultat final. Nous avons implanté quatre principales relations prises en compte par notre itérateur de base. Ces relations sont résumées dans le Tableau 3.

Par exemple, pour trouver tous les descendants d'une classe, on peut simplement appliquer un itérateur sur cette classe en utilisant un filtre de structure ayant comme relation *ITERATE-INHERITENCE_GRAPH*.

TYPE DE RELATIONS	DESCRITPITON
ITERATE-ALL.CALL-GRAPH LEVEL	• Elle permet de parcourir tous les niveaux qui correspondent aux appels sortants, d'une ou plusieurs classes, dans un graphe d'appel.
ITERATE-IN/OUT.CALL_GRAPH	• Elle est similaire à la première relation, mais la différence est qu'elle prend en considération les appels entrants.
ITERATE-INHERITENCE_GRAPH	• Elle consiste à parcourir tout le graphe d'héritage à partir d'une classe.

Tableau 3. La liste des relations utilisée avec l'itérateur de base.

5.3.2 Itérateur conditionnel

Cet itérateur permet de parcourir conditionnellement soit un graphe d'appel, soit l'arbre d'héritage d'une ou plusieurs classe(s). Cet itérateur est similaire à l'itérateur de base à l'exception qu'à chaque niveau d'itération, une condition est appliquée aux éléments

du résultat. Seuls les éléments vérifiant la condition sont considérés dans la prochaine itération. La condition peut être un filtre de recherche par mots clés, de recherche dans les noms, de recherche de similarité entre les éléments ou bien un filtre quantitatif. L'itérateur conditionnel est appliqué automatiquement à tous les niveaux sans interruption, ou bien il peut être exécuté niveau par niveau. Ce type de parcours permet à l'analyste l'inspection et la modification de résultats à chaque itération. Dans le cas des graphes qui comportent des cycles, pour chaque élément dans le graphe, l'itérateur vérifie si cet élément n'a pas été visité durant le processus. Cette vérification est faite pour éliminer le problème des cycles.

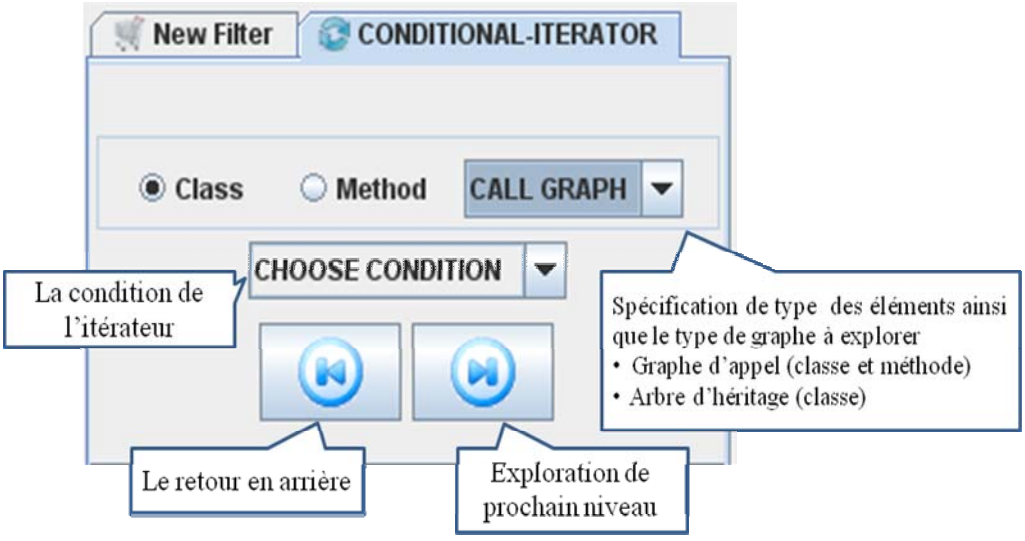


Figure 22. La barre de navigation dans l'itérateur conditionnel.

La Figure 22 montre l'interface utilisée pour l'exécution de l'itérateur conditionnel. En utilisant les boutons de navigation, l'analyste peut avancer ou reculer dans les niveaux du graphe considéré. Pour illustrer l'utilisation de cet itérateur, prenons le cas où un analyste est en train de chercher dans l'arbre d'héritage des classes qui contiennent des méthodes qui traitent la fonctionnalité "add". Nous nous intéressons uniquement aux chemins dans le graphe d'héritage où toutes les classes ont un "add". Dans l'exemple de l'itérateur de base, le mainteneur peut explorer tous les descendants d'une certaine classe. Par contre, avec notre itérateur conditionnel, nous utilisons une condition qui va s'appliquer à chaque niveau dans l'arbre d'héritage. Dans ce cas, la condition utilisée est le filtre de recherche dans les noms prenant en paramètre la chaîne "add". Tout d'abord, cette condition s'applique au

résultat initial de la requête. Ensuite, à chaque niveau dans l'arborescence, les classes qui respectent la condition mentionnée verront leurs descendants explorés par la suite.

5.4 Conclusion

Dans ce chapitre, nous avons décrit le principe de formulation des requêtes complexes. Nous avons montré les deux modes de composition. Le premier est basé sur la composition des résultats de filtres en utilisant les opérateurs unaires et binaires. Le deuxième mode est basé sur l'imbrication de filtres. Pour chaque mode de formulation, nous avons donné un exemple d'utilisation.

Chapitre 6.

Mise en œuvre et évaluation

6.1 Introduction

Dans ce chapitre, nous évaluons notre approche en reproduisant deux travaux permettant de résoudre deux problèmes différents de maintenance du logiciel. Le premier travail est lié au problème de détection des défauts de conception et le deuxième cible le problème de la localisation des fonctionnalités. Pour les deux travaux, nous avons implanté les solutions proposées en utilisant la formulation interactive de requêtes offerte par notre environnement IQOP. Les deux travaux existants utilisés sont DECOR [27] pour la détection des défauts de conception, et DORA [8] pour la localisation des fonctionnalités dans le code. L'objectif est de montrer que notre environnement permet de reproduire de manière interactive les solutions, mais également que l'interactivité permet d'améliorer la qualité des résultats des solutions automatisées. Pour effectuer notre étude, nous avons choisi deux projets open source en Java, GanttProject v1.10.2 [11] et JBidWatcher v1.0pre6 [12], sur lesquels nous appliquons les solutions. GanttProject est un outil de création des calendriers de projets. Il contient environ 21.000 lignes de code et 181 classes. JBidwatcher gère les différentes sortes de ventes sur eBay (achat immédiat, enchères, etc.) ainsi que les monnaies. Il contient environ 23.000 lignes de code et 183 classes.

Dans la suite de ce chapitre, nous décrivons par la suite notre étude sur chacun des travaux. Dans la partie 6.2, nous décrivons les détails de l'implantation de notre système. Ensuite, nous étudions avec un exemple la mise en œuvre de la détection des défauts de conception. Par la suite, dans la 6.4, nous examinons, avec un exemple également, l'application à la localisation des fonctionnalités.

6.2 Implantation

Dans notre travail, nous avons mis en place un module interactif d'interrogation de programmes. Ce module peut être considéré comme une extension de l'environnement de visualisation VERSO [10]. En effet, pour des systèmes de grande taille comme celui à la Figure 7, l'exploration d'un logiciel est très difficile en utilisant uniquement la visualisation. Dans ce cadre, nous avons implanté différents types de filtres permettant de réduire l'ensemble de données à explorer. Nous avons également implanté des mécanismes de composition de ces filtres par l'utilisation de différents types d'opérateurs. L'intégration des filtres a nécessité l'ajout de 21 classes dont 5 correspondent aux filtres de base en plus de la modification des classes existantes de VERSO.

Plus précisément, nos filtres de base forment 4 catégories : linguistique, structurel, quantitatif, et d'interactivité. Les filtres linguistiques sont KEYWORD_SEARCH, ELEMENT_SEARCH, et SIMILARITY_SEARCH. Ils se basent sur des algorithmes de recherche d'information. Nous avons donc implanté différents types d'algorithmes comme TF-IDF et LSI (voir Section 4.2.1). De plus, nous avons implanté des mécanismes de prétraitement des données comme l'extraction de contenu des méthodes et des classes et l'indexation. Afin de réaliser ces étapes, nous avons modifié différentes classes dans VERSO. Ceci a nécessité par exemple l'ajout d'un attribut à la classe « *Entity* » qui identifie le contenu de chaque élément (classe, méthode ou interface), ainsi que des méthodes qui gèrent l'extraction et le prétraitement. Pour l'indexation, nous avons ajouté un module contenant 8 classes permettant de gérer l'indexation avec TF-IDF ou avec LSI.

D'autre part, nous avons mis en place une classe responsable de la composition des filtres, nommée « *Filter_Combiner* ». Cette classe contient plusieurs méthodes qui gèrent l'interface graphique, ainsi que la composition des résultats. En plus, nous avons mis en place des classes qui gèrent l'entrelacement entre les filtres comme la classe « *Conditional_Iterator* » et la classe « *Iterator* ».

Pour l'affichage des résultats des filtres, nous avons ajouté à VERSO, l'affichage de la hiérarchie des packages (comme le *Package Explorer* d*'Eclipse)*. Nous avons également modifié les classes de la visualisation pour y inclure le nommage des éléments. Ces options

aident l'utilisateur à se localiser dans la scène de VERSO. D'autre part, nous avons implanté diverses stratégies pour mettre en évidence les éléments retournés par les filtres.

6.3 Application à la détection de défauts de conception

Les défauts de conception sont des problèmes qui peuvent affecter le développement des logiciels. Ces défauts sont causés par une mauvaise programmation ou conception. Plusieurs solutions ont été proposées pour remédier à ce problème. Dans la section 5.3.2, nous avons décrit le principe ainsi que les techniques d'un ensemble de ces approches.

Parmi ces travaux, nous nous sommes intéressés à celui de Moha et al. [27]. Dans leur approche, nommée DECOR, ils proposent une classification de défauts de conception en fonction des symptômes. Ainsi, ils décrivent les défauts de conception en utilisant un langage à base de règles abstraites correspondant aux symptômes. Comme le montre la Figure 23, la définition de base du défaut de conception *Blob* dans DECOR correspond à une classe contrôleur, associée à un ensemble de classes de type données. Cette définition est formulée à l'aide de quatre métriques logicielles: LCOM, NACC, NMD, NAD [30], ainsi que des critères sur les noms des classes et des méthodes. Chaque métrique, décrivant un symptôme, a une valeur seuil à partir de laquelle on considère que le symptôme est avéré. La combinaison entre les métriques/seuils et les recherches sur les noms est faite à l'aide de deux opérateurs, l'union et l'intersection. Les métriques, les seuils, les recherches dans les noms et opérateurs constituent les paramètres d'une règle.

Une des limites de cette approche est la définition d'une valeur seuil qui doit s'appliquer à toute circonstance. En effet, cette valeur devrait dépendre du contexte du système à évaluer pour tenir compte de la diversité des logiciels existants et des pratiques de conception/programmation. En d'autres termes, il a été montré qu'une valeur qui fonctionnerait pour un système, ne fonctionnerait probablement pas pour un autre [29]. Par exemple, la détection du blob a besoin d'information quantitative comme la taille de la classe (une grande classe). Bien que l'on puisse mesurer la taille d'une classe, il n'est pas

trivial de définir une valeur seuil appropriée. Une classe peut être considérée grande dans un programme donné, mais moyenne dans un autre [14].

Cette limite explique en bonne partie le grand nombre de faux positifs détectés avec DECOR, car les règles mettent en œuvre de nombreux seuils. Les faux positifs sont des éléments du système qui ont été détectés comme des anomalies, bien que ces éléments ne soient pas des défauts en réalité. Par ailleurs, le rappel est généralement élevé, car les valeurs seuil sont moins restrictives. Dans la page suivante, nous donnons un exemple de détection de blob avec DECOR et nous calculons son rappel ainsi que sa précision avec le système GanttProject v1.10.2. Nous conjecturons que l'utilisation de la visualisation interactive peut aider à prévenir certains faux positifs.

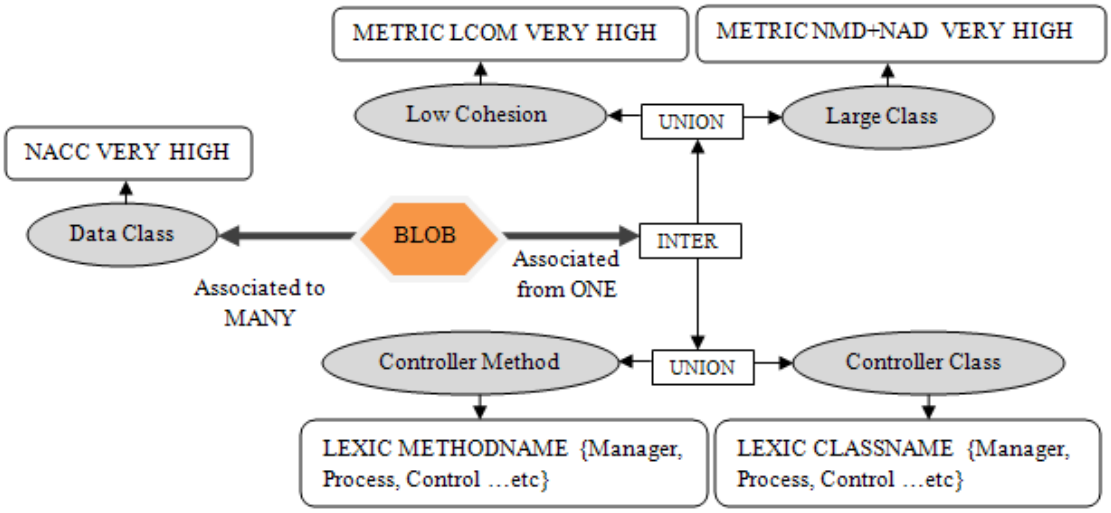


Figure 23. La définition d'un blob dans DECOR [27].

La Figure 24 montre la liste des blobs détectés à l'aide de DECOR dans le système GanttProject. Seulement 4 classes (celles en fond blanc) parmi 10 correspondent à de vrais blobs. Ceci donne dans ce cas une précision de 40% avec un rappel de 100%. En utilisant notre environnement de composition de filtres et la définition d'un blob dans DECOR, nous arrivons à reproduire les résultats de détection d'un blob par DECOR. Grâce à notre environnement interactif et à l'aide de la métaphore de visualisation utilisée, nous avons détecté tous les blobs qui sont des vrais positifs mais en eliminant les faux positifs au fur et à mesure de la recherche comme le montre la Figure 25.

Class	Blob-DECOR
GanttGraphicArea	x
GanttLookAndFeelInfo	x
GanttOptions	x
GanttProject	x
GanttStatusBar	
GanttTaskPropertiesBean	x
GanttTree	x
GanttXFIGSaver	x
GregorianTimeUnitStack	x
ResourceLoadGraphicArea	x
TaskImpl	x

Figure 24. La liste des blobs détectés par DECOR.

Prenons l'exemple de la classe TaskImpl qui est un faux positif de DECOR, pour illustrer notre démarche. Comme le montre la Figure 25, l'utilisateur peut remarquer que la boîte correspondant à cette classe n'est pas très haute (la hauteur de la boîte correspond à la métrique WMC qui représente la complexité) par rapport à d'autres classes dans le système. Il peut donc conclure que ce n'est pas un blob et le supprimer de l'ensemble des résultats avant de poursuivre avec les autres symptômes. Il peut même, en cas de doute, regarder le code ou une partie du code de cette classe.

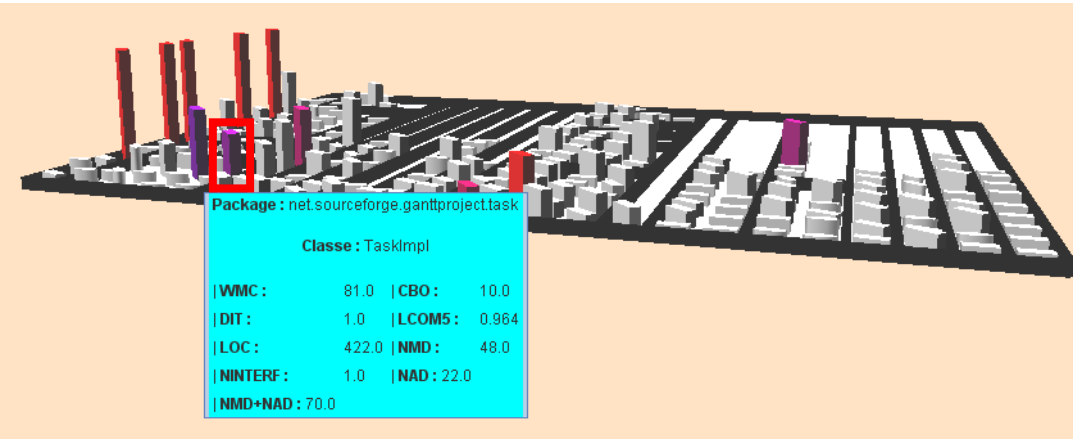


Figure 25. La classe TaskImpl dans GanttProject v 1.10.2.

En outre, comme une partie de l'analyse des symptômes dépend du fait que la classe est un contrôleur, l'utilisateur peut essayer des techniques linguistiques plus sophistiquées

comme l'indexation sémantique latente [22][15] pour déterminer les classes contrôleurs au lieu d'une recherche simple dans le nom avec une liste finie de synonymes. Toutefois, dans DECOR, ces termes sont introduits manuellement et correspondent parfois à des noms de certaines classes ou méthodes qui ne sont pas en fait des contrôleurs. Par exemple, si nous introduisons le terme «*control*» comme terme dans DECOR, seulement deux classes sont détectées et ces classes ne correspondent pas à des blobs. Lorsque nous avons utilisé le filtre linguistique *KEYWORD_SEARCH* qui prend en paramètre *COSINE-LSI*, nous avons détecté 9 classes dont 3 sont des blobs (Figure 26).

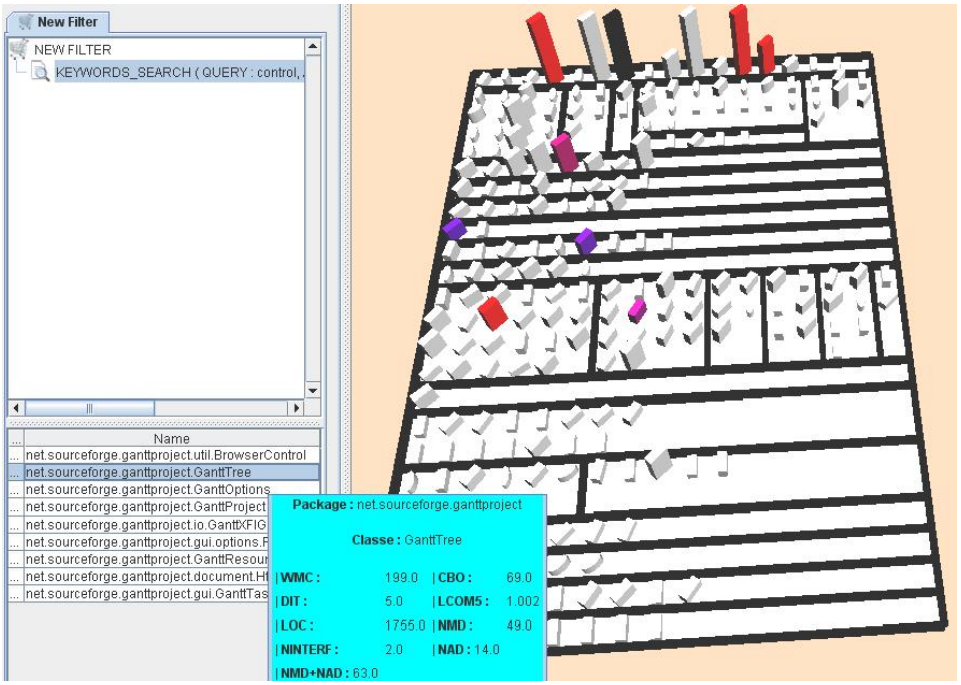


Figure 26. Détection des classes contrôleur dans GanttProject avec le filtre *KEYWORD_SEARCH*.

Nous avons essayé à reproduire les étapes décrites par Moha et al. [27] pour la détection de blob dans un programme. En utilisant la composition de filtres offerte par notre outil, nous arrivons à reproduire les résultats de DECOR mais nous avons remarqué que dans une certaine étape de processus de recherche, il existe des classes comme la classe TaskImpl, à la Figure 25, qui ne représente pas un blob dans GanttProject. En utilisant la visualisation ainsi que l'aspect interactivité, nous pouvons supprimer ces classes qui sont

des faux positifs. Dans un deuxième lieu, puisque notre outil offre une formulation personnalisée des requêtes, nous pouvons changer la recherche dans les noms proposée par DECOR par une recherche plus sophistiquée avec TF-IDF ou LSI. Comme nous montre la Figure 26, pour chercher les classes contrôleurs dans un programme, la précision est meilleure que celle de DECOR.

6.4 Application à la localisation de fonctionnalités

La compréhension d'un programme est une activité importante de la maintenance. Elle est utilisée pour faciliter l'entretien, la réutilisation, la réingénierie, l'exploration de code, la localisation de fonctionnalités, etc. La localisation de fonctionnalités en particulier permet d'identifier la partie de code source qui implémente une fonctionnalité spécifique du système. Un des outils proposés pour la localisation de fonctionnalités est celui de Hill et al. [8], nommé DORA. Il utilise une technique qui exploite à la fois la structure du programme et les informations lexicales pour déterminer les éléments du code en relation avec la fonctionnalité recherchée.

Prenons l'exemple de la fonctionnalité d'ajout des enchères dans le programme JBidWatcher v1.0pre6 [12]. Nous nous intéressons à trouver les méthodes qui interviennent principalement dans cette fonctionnalité. Cette localisation peut-être motivée par le fait que l'exécution de la fonctionnalité qui nous intéresse génère un bogue. Les méthodes pertinentes qui implémentent cette fonctionnalité sont en principe *DoAdd()* et *DoPasteFromClipboard()*. À partir d'un point de départ particulier (la méthode *DoAction()*), Hill et al. [8] ont montré la limite de l'utilisation du graphe d'appel, et des techniques de recherche d'information séparément.

Avec l'exploration du graphe d'appels, la précision est faible. En effet, le DORA trouve 40 méthodes dont seulement deux sont effectivement pertinentes. D'un autre côté, avec la technique de recherche d'information, les deux méthodes pertinentes sont retournées avec 48 autres qui ne le sont pas en réponse à la requête "*add auction*". La précision est également très faible dans ce cas.

À partir de cette observation, Hill et al. [8] utilisent une combinaison entre les informations structurelles et lexicales afin de dégager l'ensemble de voisins pertinents à partir d'un point de départ. Cette approche se déroule en 4 étapes:

1. Extraction de la liste des voisins directs d'un point de départ.
2. Calcul du score de pertinence lexicale des voisins.
3. Retrait des méthodes ayant un score inférieur à un certain seuil.
4. Ré-exploration récursive.

Une des limites de cette approche automatique est la définition des valeurs seuils. L'utilisation de ces seuils peut ignorer des méthodes qui sont pertinentes à la requête. En fait, cette valeur dépend également du contexte du système à évaluer. En outre, le système n'offre pas la possibilité de changer la valeur de seuil en cours de recherche. Ainsi, l'utilisateur doit explorer tout le graphe d'appel et ne peut pas s'arrêter quand il trouve les méthodes adéquates.

Avec notre environnement interactif, nous avons mis en œuvre cette solution. Contrairement à DORA, dans IQOP, l'utilisateur contrôle la recherche. Cette option est due à l'interaction avec les résultats, offerte par IQOP et la manière interactive de formuler les requêtes. En outre, nous pouvons appliquer cette approche à des classes et pas seulement aux méthodes.

En utilisant IQOP, le scénario pour la mise en œuvre de cette approche contient quatre étapes. Ce scénario est montré dans la Figure 27.

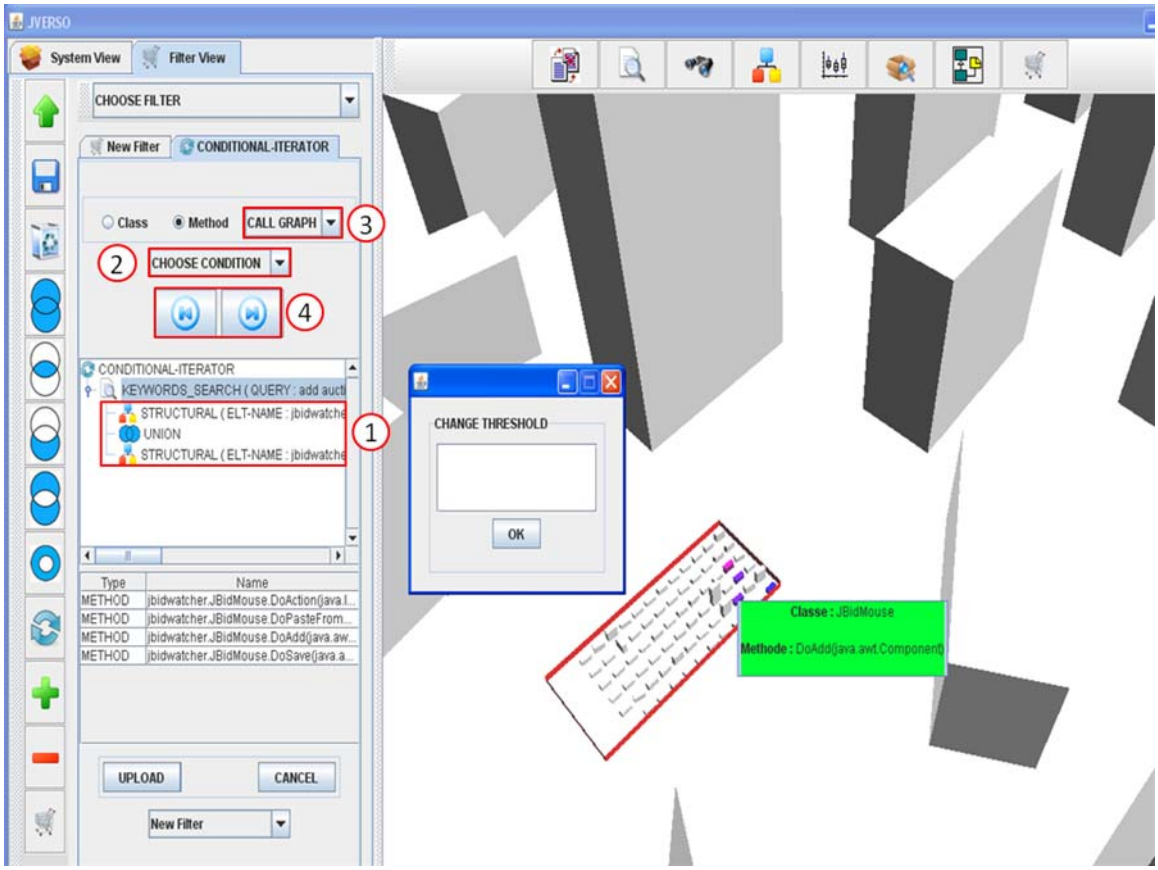


Figure 27. Visualisation de résultats de DORA.

Avant de commencer, nous choisissons le *CONDITIONAL_ITERATOR* permettant d'explorer de façon itérative le graphe d'appels. Cette exploration obéit aux conditions mentionnées dans la section 5.3.2. La première étape consiste à choisir un ou plusieurs point(s) de départ à utiliser. Dans notre exemple, le point de départ est la méthode *DoAction()* ayant la signature suivante: *JBidMouse.DoAction (Object, String, AuctionEntry)*. Ensuite, la deuxième étape consiste à spécifier le type des éléments ainsi que le type de graphe à explorer. Pour le cas des méthodes, seul le graphe d'appel sera utilisé. Dans la troisième étape, nous choisissons le filtre *KEYWORD_SEARCH* comme une condition de l'itérateur. Ce filtre prend en paramètre une valeur seuil et une requête contenant ces mots clés "*add auction*". Finalement, une fois la condition choisie, nous réexplorons de manière récursive le graphe en utilisant la barre de navigation dans notre itérateur conditionnel. À chaque changement de niveau dans le graphe d'appels, la requête

garde uniquement les méthodes qui respectent la condition choisie. Pendant la recherche, nous pouvons tester différentes valeurs de seuil (Figure 27), et inspecter les résultats à tout moment du processus. Grâce à notre outil de visualisation, on peut faire des zoom pour accéder aux méthodes pertinentes contenues dans la classe *JBidMouse*.

6.5 Conclusion

Dans ce chapitre, nous avons pris deux exemples d'application de notre environnement interactif de formulation de requêtes. Nous avons montré qu'avec l'utilisation de cet environnement interactif, le programmeur a la possibilité d'implanter une famille des travaux existants qui touchent plusieurs problèmes spécifiques en maintenance. Avec IQOP ainsi qu'avec une connaissance du contexte du programme à évaluer, on peut améliorer et raffiner les résultats donnés par les approches automatiques surtout en utilisant l'interactivité et la visualisation de résultats.

Chapitre 7.

Conclusion générale

Les systèmes logiciels sont vastes et complexes, que ce soit en terme des entités logicielles qu'en terme de leur relation. Par conséquent, comprendre le fonctionnement d'un système logiciel nécessite d'avoir la capacité de poser différents types de questions sur les entités du système. La compréhension du code source est une activité vitale pour de nombreuses tâches en génie logiciel. Les développeurs ont besoin de comprendre le système, sur lequel ils travaillent, dans le but d'améliorer la qualité et les fonctionnalités, ainsi réduire le coût de maintenance.

Les outils d'interrogation de code sont conçus pour aider les développeurs à comprendre les relations complexes entre les différentes entités. Les outils existants souffrent des problèmes de paramétrage de la recherche (valeurs seuils, etc.). Dans ce cas, la visualisation peut être une solution permettant de résoudre ce problème en donnant une vue d'ensemble de logiciels. Dans le cas d'un système de grande taille, l'utilisateur peut se perdre dans la scène de visualisation devant un grand nombre de composants logiciels anonymes. La solution consiste à concevoir un outil de recherche qui permet de bien préciser le(s) composant(s) à maintenir ou à utiliser. Par contre, le problème majeur des systèmes d'exploration de code est que l'utilisateur a des difficultés à transformer ses besoins en des requêtes ce qui provoque l'existence d'un grand écart entre l'énoncé du problème et la description des solutions. Le but principal de notre travail est de rendre la tâche de recherche plus interactive entre le développeur et le code source. D'où l'idée est d'implanter un système d'interrogation de code qui permet de combiner un ensemble de filtres de différentes natures: linguistiques, structurels et quantitatifs.

Plus concrètement, dans ce mémoire, nous avons présenté une approche semi-automatique d'interrogation de code. Cette approche est basée sur la formulation interactive

et itérative de requêtes basée sur la visualisation. L'interrogation de code est réalisée par l'implantation de différents types de filtres.

Nous avons essayé d'offrir un outil qui rend la tâche des développeurs plus souple et les aide à résoudre différentes tâches de maintenance. L'idée est d'économiser le temps des développeurs, c'est-à-dire, au lieu de réimplanter les techniques utilisées dans certaines approches automatiques, ils peuvent utiliser notre outil afin de reproduire les résultats de ces approches. De plus, notre outil permet une grande variété de possibilités d'exploration des logiciels existants. Par comparaison, des outils comme Ferret [2] limitent les possibilités d'exploration à un nombre prédéfini de requêtes. Par ailleurs, nous offrons des mécanismes de combinaison de filtres qui vont au-delà des simples opérateurs ensemblistes sur les résultats. Ceci n'est pas le cas par exemple des outils comme OMEGA [18] ou EVOLIZER [25].

La structure de ce mémoire reflète notre démarche méthodologique. D'abord, nous avons fait une revue de l'état de l'art dans le domaine d'interrogation de code que ce soit les approches génériques proposées ainsi que celles destinées à résoudre des problèmes bien spécifiques. Ensuite, nous avons discuté de notre environnement interactif en décrivant les deux modules existants : (1) un module d'interrogation et (2) un module de visualisation de résultats. Par la suite, nous avons décrit l'ensemble des filtres de base de notre système que nous avons illustrés à l'aide d'exemples. Nous avons ensuite montré le processus de construction des requêtes complexes dans notre système. Finalement, nous avons présenté les résultats de deux études de cas réalisées lors de cette maîtrise. Dans ces deux cas d'étude, nous avons décrit l'applicabilité de notre environnement en comparaison à des approches automatiques proposées dans la littérature, et avons mis en avant le bénéfice apporté par l'aspect interactif de la formulation de requêtes d'une part, et par la visualisation d'autre part.

Comme travaux futurs, nous comptons réaliser une étude empirique afin de tester l'utilisabilité de notre outil. Par exemple, nous demandons à divers utilisateurs de faire différentes tâches de maintenance, afin d'étudier leur comportement face à notre système, en mesurant les résultats produits, ainsi que le temps mis pour les produire, etc. En outre,

nous prévoyons de concevoir une meilleure façon de visualiser les résultats de filtres par l'utilisation de composition et de dégradation des couleurs. En revanche, notre procédé de visualisation actuelle consiste à garder les couleurs originales des classes qui ont été sélectionnées par un filtre, alors que les autres classes sont montrées en couleur grise. Ceci nous permettrait par exemple, dans le cas d'un filtre de recherche par mots clés, d'utiliser un dégradé de couleurs afin de rendre compte de la pertinence des résultats retournés à l'utilisateur. Nous pensons aussi à étendre l'ensemble des artefacts logiciels utilisés (e.g, la documentation de code, les traces d'exécution, etc) afin de toucher à d'autres problèmes comme la traçabilité.

BIBLIOGRAPHIE

[1] Alkhatib G., "*The Maintenance Problem of Application Software: An Empirical Analysis*", Journal of Software Maintenance – Research and Practice, 4(2):83-104, 1992.

[2] B. de Alwis and G. C. Murphy, "*Answering conceptual queries with ferret*", In Proc. Int'l Conf. Softw. Eng., pp. 21–30, New York, USA, 2008. ACM.

[3] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, "*Introduction to Information Retrieval*", Cambridge University Press. 2008.

[4] D. Janzen and K. de Volder, "*Navigating and querying code without getting lost*", In 2nd International Conference on Aspect-Oriented Software Development, pages 178–187, 2003.

[5] E. Hajiyev, M. Verbaere, and O. de Moor, "*CodeQuest: scalable source code queries with Datalog*", In D. Thomas, editor,Proceedings of ECOOP, volume 4067 of Lecture Notes in Computer Science, pp. 2–27, Springer, 2006.

[6] E. McCormick and K. D. Volder. "*JQuery: finding your way through tangled code*". In Companion to OOPSLA, pp. 9–10. ACM Press, 2004.

[7] Emily Hill, "*Developing natural language-based program analyses and tools to expedite software maintenance*", ICSE'08*,* May 10–18, 2008, Leipzig, Germany.

[8] Emily Hill, and K. Vijay-Shanker, "*Exploring the neighborhood with dora to expedite software maintenance*", ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.

[9] G. G. Chowdhury, "*Introduction to Modern Information Retrieval*", Facet, London, 2nd edition, 2004.

[10] Guillaume Langelier, Houari Sahraoui, and Pierre Poulin, "*Visualization–based Analysis of Quality for Large–Scale Software*", 20th IEEE/ACM International Conference on Atomated Software Engineering (ASE), 2005.

[11] http://ganttproject.biz/index.php

[12] http://sourceforge.net/projects/jbidwatcher/

[13] http://tedlab.mit.edu/~dr/SVDLIBC/#formats

[14] Karim Dhambri, Houari Sahraoui, and Pierre Poulin. "*Visual Detection of Design Anomalies"*. In Proceedings of the 12th European Conference on Software Maintenance and Reengineering, Tampere, Finland, pages 279-283, April 2008. IEEE Computer Society.

[15] Landauer, T. K., Foltz, P. W., & Laham, D. "*Introduction to Latent Semantic Analysis*". Discourse Processes, vol. 25, pp.259-284, 1998.

[16] Langelier G., "*Visualisation de la qualité des logiciels de grandes tailles*", M.Sc. Thesis, Département d'informatique et de recherche opérationnelle, Université de Montréal, 2006. B. de Alwis and G. C. Murphy, "Answering conceptual queries with ferret", In Proc. Int'l Conf. Softw. Eng., pp. 21–30, New York, USA, 2008. ACM.

[17] Lientz, B. P., Swanson, B. E., "*Software Maintenance Management*", Addison-Wesley, Reading, MA, 1980.

[18] M. A. Linton. "*Implementing relational views of programs*". In P. B. Henderson, editor, Software Development Environments (SDE), pages 132–140, 1984.

[19] M. Consens, A. Mendelzon, and A. Ryman, "*Visualizing and Querying Software Structures*". In Proceedings of the 14th International Conference on Software Engineering, pages 138-156, May 1992.

[20] M. Marin, A. Van Deursen, and L. Moonen. "*SoQueT: Query-based documentation of crosscutting concerns"*. In 29th International Conference on Software Engineering (ICSE 2007), pages 758–761, 2007.

[21] M.-A. D. Storey, C. Best, and J. Michaud. "*Shrimp views: An interactive and customizable environment for software exploration*". In Proc. of International Workshop on Program Comprehension (IWPC '2001), 2001.

[22] Marcus A., Sergeyev A., Rajlich V., and Maletic J., "*An Information Retrieval Approach to Concept Location in Source Code*" in the Proceedings of the 11th IEEE

Working Conference on Reverse Engineering (WCRE2004), Delft, The Netherlands, November 9-12, pp. 214-223, 2004.

[23] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. "*Visualizing and querying software structures*". In ICSE '92: Proceedings of the 14th international conference on Software engineering, pages 138–156, New York, NY, USA, 1992. ACM Press.

[24] Mathieu Verbaere, Michael W. Godfrey and Tudor Gîrba, "*Query Technologies and Applications for Program Comprehension*", In Proceedings of ICPC 2008, pp. 285-288.

[25] Michael Würsch, Giacomo Ghezzi, Gerald Reif, and Harald C. Gall," *Supporting Developers with Natural Language Queries",* ICSE '10, May 2-8 2010, Cape Town, South Africa.

[26] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, 2nd edition,1997.

[27] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. "*DECOR: A Method for the Specification and Detection of Code and Design Smells*". IEEE Transactions on Software Engineering, vol. 36, no. 1, pp. 20-36, 2010.

[28] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble, " *Keynote Address: .QL for Source Code Analysis*", SCAM 2007: pp. 3-16, 2007.

[29] Radu Marinescu, "*Detection Strategies: Metrics-Based Rules for Detecting Design Flaws*", Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004), pp. 350 -359, 2004.

[30] S. R. Chidamber and C. F. Kemerer, "*A metrics suite for object oriented design*," IEEE Transactions on Software Engineering, vol. 20, no. 6, pp.476–493, 1994.

[31] Semmle Ltd. Company website with free downloads, documentation,and discussion forums. http://semmle.com, 2007.

[32] Wei Zhao and Lu Zhang, "*SNIAFL: Towards a Static Noninteractive Approach to Feature Location*", ACM Transactions on Software Engineering and Methodology, Vol. 15, No. 2, pp. 195–226,2006.

[33] Y.-S. Lee, B.-S. Liang, S.-F. Wu, and F.-J. Wang, "*Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow*," Proc. Int'l Conf. Software Quality, Maribor, Slovenia, 1995.