Université de Montréal

Génération automatique de configurations et de scénarios d'utilisation d'outils de visualisation à partir de spécifications de tâches d'analyse de logiciels

par

Ahmed Sfayhi

Département d'informatique et de recherche opérationnelle

Faculté des Arts et des Sciences

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

Avril, 2015

© Ahmed Sfayhi, 2015

Résumé

Nous proposons une approche qui génère des scénarios de visualisation à partir des descriptions de tâches d'analyse de code. La dérivation de scénario est considérée comme un processus d'optimisation. Dans ce contexte, nous évaluons différentes possibilités d'utilisation d'un outil de visualisation donnée pour effectuer la tâche d'analyse, et sélectionnons le scénario qui nécessite le moins d'effort d'analyste. Notre approche a été appliquée avec succès à diverses tâches d'analyse telles que la détection des défauts de conception.

Mots-clés : rétro ingénierie, visualisation du code source, visualisation de logiciels, tâches d'analyse

Abstract

We propose an approach that derives interactive visualization scenarios from descriptions of code analysis tasks. The scenario derivation is treated as an optimization process. In this context, we evaluate different possibilities of using a given visualization tool to perform the analysis task, and select the scenario that requires the least effort from the analyst. Our approach was applied successfully to various analysis tasks such as design defect detection and feature location.

Keywords: Reverse engineering, source code visualization, software visualization, analysis task

Table des matières

Résumé i

Abstract ii

Table des matières iii

Liste des tableaux v

Liste des figures vi

Liste des sigles vii

Remerciements ix

CHAPITRE 1 : INTRODUCTION 10

1.1 Contexte 10

1.2 Problématique 11

1.3 Objectifs 12

1.4 Structure de la mémoire 13

CHAPITRE 2 : ÉTAT DE L'ART 14

2.1 Généralités 14

2.2 La visualisation des logiciels 14

2.3 Formalisation de la description des tâches de maintenance et des actions de visualisation 18

2.4 La visualisation assistée 20

2.5 Conclusion 23

CHAPITRE 3 : APPROCHE 24

3.1 Introduction 24

3.2 Description des tâches d'analyse 25

3.2.1 Méta-modèle 26

3.2.2 Utilisation du langage proposé 30

3.2.3 Exemple de description d'une tâche d'analyses 30

3.3 Description des outils de visualisation.. 32
3.3.1 Exemple de description d'un outil de visualisation.. 34
3.4 Modélisation des scénarios de visualisation et des vues... 36
3.5 Génération des scénarios de visualisation.. 38
3.5.1 Génération des Scénarios comme un problème d'optimisation 38
3.6 Codage du scénario de visualisation 41
3.7 Obtention des nouveaux scénarios 42
3.8 Évaluation des scénarios de visualisation 45
3.8.1 Les contraintes des scénarios irréalisables (C1) 46
3.8.2 Les contraintes d'utilisation des interactions de visualisation (C2) 48
3.8.3 Les contraintes de mappings des métriques à des attributs graphiques (C3)........... 49
3.9 Fonctionnement de l'Algorithme 50
CHAPITRE 4 : ÉTUDE DE CAS 52
4.1 Génération des vues et des scénarios de visualisation 52
CHAPITRE 5 : CONCLUSION 57
Bibliographie i

Liste des tableaux

Tableau 1: Description des primitives 28

Tableau 2: Description de la tâche de détection de Blob 31

Tableau 3 : Interaction de visualisation 38

Tableau 4 : Exemples de transformations possibles de la primitive filtre (Filter) 39

Tableau 5: Classification des indices visuels selon l'échelle d'expressivité [12] 50

Tableau 6 : Associations (mappings) des données pour le meilleur scénario de détection de Blob 53

Tableau 7 : Le meilleur scénario obtenu pour la détection de Blob 54

Tableau 8 : Coût des meilleurs scénarios dans chaque génération 56

Liste des figures

Figure 1 : Visualisation du web dans un espace hyperbolique [27]. 15

Figure 2 : Une vue de l'outil proposé par Wettel et Lanza [39] 16

Figure 3 : Une vue de l'outil proposé par Langelier et al. [18]. 18

Figure 4: Historiques des conditions météorologiques pour le mois de janvier dans tous les états unis d'Amérique 21

Figure 5 : Aperçu de notre approche 25

Figure 6 : Vue simplifié du méta-modèle des tâches d'analyses 27

Figure 7 : Vue simplifié du méta-modèle des outils de visualisation 34

Figure 8 : Vue et représentation graphique de Verso 35

Figure 9 : Vue simplifié du méta-modèle des vues et des scénarios de visualisation 37

Figure 10 : Codage du scénario de visualisation – Population initial (solutions 1 à n) 43

Figure 11: Opération de croisement 44

Liste des sigles

CC : Classes contrôleuses

WMC: Weighted Methods per Class

LCOM5: Lack of Cohesion in Methods

DIT: Depth in Inheritance Tree

À ma famille

À mes amis

Remerciements

Je remercie les membres du jury pour évaluer ma recherche. Je tiens aussi à remercier mon directeur de recherche ainsi que tous les membres de l'équipe GEODES et du DIRO qui m'ont aidé durant mes recherches et mes travaux.

CHAPITRE 1 : INTRODUCTION

1.1 Contexte

De nos jours, les logiciels connaissent une croissance fulgurante. Ils deviennent de plus en plus grands et complexes, sont très variés et touchent plusieurs secteurs de notre vie [28].

Face à ces changements et à l'avancement technologique rapide, les concepteurs et les développeurs des logiciels doivent de plus en plus tenir compte de la qualité et de la productivité, c.-à-d., la réduction du temps/effort de développement et de maintenance. Pour répondre aux nouvelles demandes de modification et d’amélioration d'un logiciel existant, les développeurs doivent constamment choisir entre développer un nouveau logiciel ou effectuer des changements sur les logiciels disponibles [17]. Dans les deux cas, ils ont besoin de comprendre le logiciel existant, détecter ses défauts et déterminer ce qui manque pour répondre aux demandes et proposer une solution. Pour cela, plusieurs méthodes et outils de compréhension de code et de rétro-ingénierie ont été proposés. Nous citons comme méthode de rétro-ingénierie le regroupement « clustering », la détection de clone, la décomposition de code « program slicing » et la visualisation de code [34].

Toutes ces approches font l'extraction de plusieurs informations sur les logiciels. Par la suite, elles permettent de les analyser, les modifier ou simplement montrent les résultats de l’extraction. L'analyse peut être manuelle, automatique ou semi-automatique. L'analyse manuelle est fastidieuse et demande beaucoup de temps et d'expertise [7]. L'analyse automatique se base généralement sur les systèmes de règles et les systèmes d'apprentissage [34]. Ces systèmes doivent être faits par un expert de tous les problèmes qu'ils vont résoudre [29]. Les résultats de ces derniers doivent être vérifiés manuellement pour s'assurer de leurs exactitudes, ce qui ralentit le développement de ces systèmes.

L'une des méthodes hybrides, entre le manuel et l’automatique, est la visualisation des logiciels [17].

Cette méthode permet de présenter les informations collectées sous une forme qui facilite leur compréhension. Elle facilite aussi la réalisation de tâches de compréhension de code.

La visualisation de logiciel est un outil puissant pour explorer d'énormes quantités de données multidimensionnelles. Par conséquent, la visualisation de logiciel devient de plus en plus importante, comme le démontre le grand nombre d'outils de visualisation qui sont produits. Cette importance a également été soulignée dans une enquête sur des chercheurs dans la maintenance des logiciels, réingénierie, et rétro-ingénierie [16]. Dans cette étude, 82% des chercheurs interrogés considèrent que les logiciels de visualisation sont au moins important dans leurs projets de recherche.

Plusieurs systèmes et outils de visualisation ont été proposés [35] [8]. La majorité de ces derniers ont été développés dans un environnement donné et pour résoudre un but bien précis, ce qui restreint leur utilisation. Face à la diversité des problèmes, la réalisation d'une tâche de maintenance nécessite un choix de l'outil le plus approprié à cette tâche. Ceci n'est pas évident pour les non-experts du domaine de la visualisation.

1.2 Problématique

Certains problèmes d'analyse ou de compréhension de code ne peuvent pas être résolus avec les approches automatiques et nécessitent une approche interactive comme la visualisation. Plusieurs outils de visualisation ont été proposés pour faciliter ces tâches d'analyse comme nous nous le décrirons dans le chapitre 2.

Cependant, utiliser un outil de visualisation pose de nombreux problèmes: comprendre cet outil, le paramétrer, le configurer et l'utiliser pour réaliser la tâche voulue. Ces étapes demandent des connaissances des sciences cognitives, de la perception visuelle humaine et une expertise de l'outil de visualisation.

À ces difficultés, s'ajoute un autre problème inhérent à l'intervention humaine dans la visualisation des logiciels. Ce problème est que les tâches de compréhension de code à être effectuées par le biais des outils de visualisation sont généralement exposées par des descriptions non formelles et leurs réalisations par un outil peuvent varier d'une personne à une autre. En supposant que nous définissons un langage rigoureux pour décrire les tâches de maintenance, il faut aussi trouver un moyen de transformer ces descriptions en des vues et des actions compatibles avec un outil de visualisation. Ceci n'est pas trivial puisque les outils sont différents et n'offrent pas les mêmes possibilités.

1.3 Objectifs

Notre objectif est de proposer un cadre générique pour aider les analystes lors de l'utilisation des outils de visualisation.

Nous proposons de développer un générateur automatique de configurations et de scénarios d'utilisation d'outils de visualisation de code qui pourra relier deux types de connaissances: l'analyse de logiciel et la visualisation interactive.
Pour cela nous proposons de définir :

- un langage pour décrire les tâches de compréhension ou d'analyse de code
- un langage pour décrire le scénario de visualisation : les vues et les actions qui seront appliquées dans l'outil de visualisation pour réaliser cette tâche.
- Un algorithme de recherche heuristique pour trouver le meilleur scénario de visualisation à partir de la description de la tâche d'analyse et la description de l'outil de visualisation

1.4 Structure de la mémoire

Ce mémoire est structuré comme suit. Dans le chapitre 2, nous décrivons l'état de l'art sur la visualisation des logiciels ainsi que la visualisation assistée. Dans le chapitre 3, nous présentons notre approche. Nous commençons par proposer un langage de description des tâches d'analyse. Puis nous proposons un méta-modèle pour modéliser les outils de visualisation. Ensuite, nous proposons une modélisation des scénarios de visualisation et des vues. Nous terminons ce chapitre par une explication de notre mécanisme de génération d'un scénario de visualisation à partir de la description d'une tâche d'analyse et d'un outil de visualisation. Dans le chapitre 4, nous présentons une étude de cas pour illustrer notre approche. Finalement, nous concluons ce mémoire en résumant notre travail de recherche et en présentant les limites de notre approche et des améliorations possibles.

CHAPITRE 2 : ÉTAT DE L'ART

2.1 Généralités

Le problème que nous voulons résoudre est nouveau et n'a pas de solutions connues, même partielles. Cependant, il existe beaucoup de travaux qui touchent à nos centres d'intérêt. Par exemple, nous trouvons de nombreux travaux sur la visualisation et plus précisément des outils de visualisation qui permettent d'explorer les logiciels, leur code, leurs évolutions et leurs caractéristiques [36][19][33]. D'autres s'intéressent à trouver une méthode pour exprimer les tâches d'analyse en général et les actions de visualisation appliquées sur un outil de visualisation. Les travaux qui ressemblent le plus à nos recherches ce sont les approches proposées pour assister la visualisation en automatisant la génération des vues.

2.2 La visualisation des logiciels

Plusieurs outils de visualisation ont été développés pour différents objectifs. Ils permettent la visualisation du web [35] [25] [27], du code source des logiciels [39] [20] [6] [36] et de leurs évolutions [19] ainsi que de leurs caractéristiques [33]. Nous allons, dans ce qui suit, discuter de quelques outils des points de vue des différentes métaphores, attributs graphiques et actions de visualisation offerts.

Munzner [27] a proposé de visualiser le web dans un espace hyperbolique. Cette métaphore permet de voir les détails d'une partie de la vue en gardant une vue globale sur le système. La Figure 1 nous montre un exemple de cette approche.

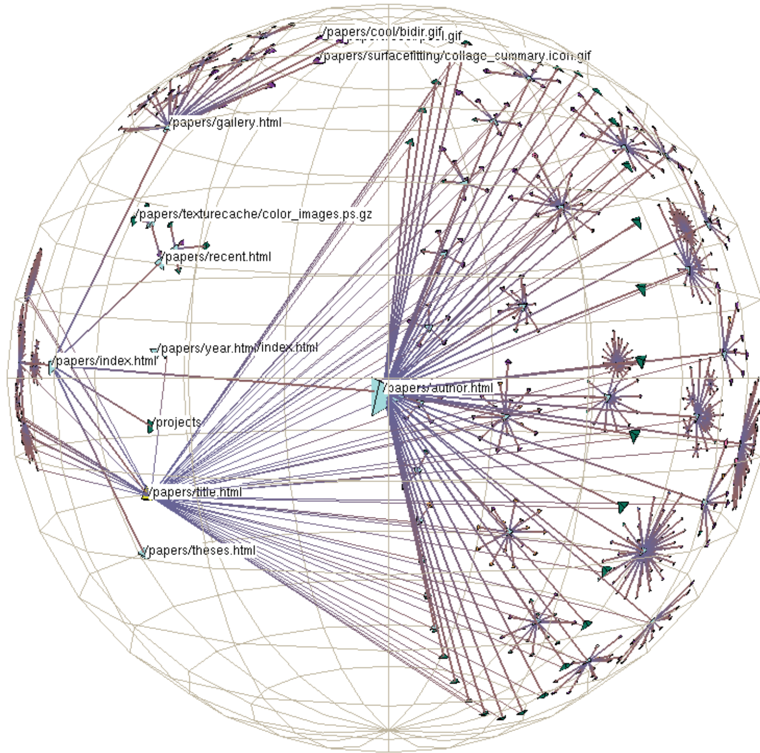

Figure 1 : Visualisation du web dans un espace hyperbolique [27].

Wettel et Lanza [39] proposent un outil de visualisation qui représente les composantes du logiciel comme des édifices dans une ville. Plusieurs attributs graphiques sont caractérisés dans cet édifice comme la dimension, la couleur, la position, la transparence et la saturation. Cet outil permet aussi de regrouper des édifices et de les mettre à une certaine altitude pour mettre en évidence la hiérarchie. Il permet aussi de sélectionner un édifice, de voir une partie de la ville dans une autre vue pour avoir plus de précision, d'appliquer des filtres ou de naviguer dans la ville. La Figure 2 nous montre une vue de cet outil. Les édifices (boîtes) représentent les classes. La hauteur et la largeur de ces édifices peuvent représenter des métriques de classes telles que le nombre de ligne de code ou la complexité.

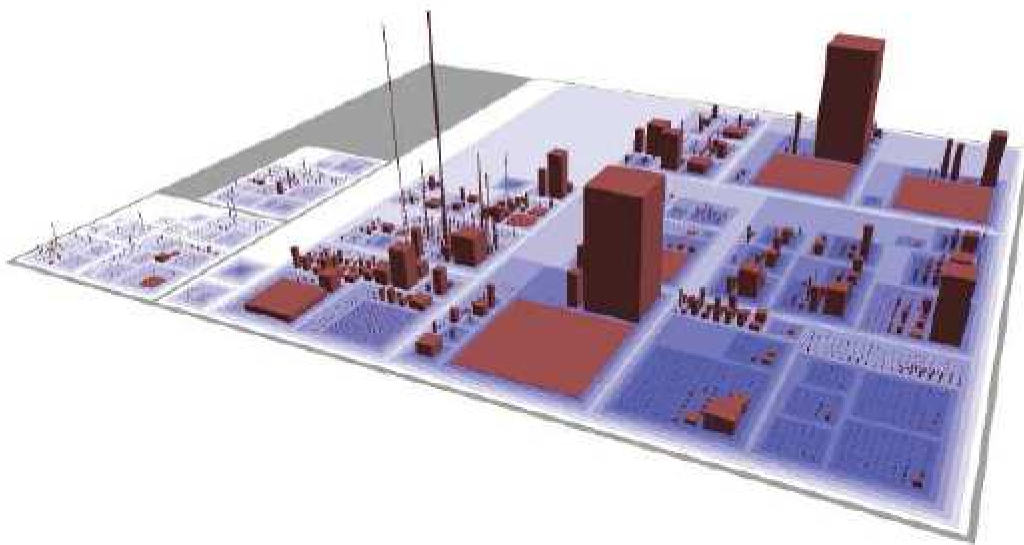

Figure 2 : Une vue de l'outil proposé par Wettel et Lanza [39]

Eick et al. [32] ont développé l'outil SeeSoft qui permet de visualiser des métriques du code source d'un logiciel. Ils présentent chaque ligne de code par un segment horizontale. Chaque segment a une couleur qui correspond aux caractéristiques de la ligne correspondante. Cet outil permet de voir les caractéristiques de la totalité des lignes de code en superposant tous les segments.

Marcus et al. [23] ont proposé une représentation 3D pour visualiser les logiciels de grande taille. Les entités à visualiser sont représentées comme des parallélépipèdes rectangles placés sur un plan. Les métriques de ces entités peuvent être représentées par la couleur, la hauteur, la profondeur et la position des parallélépipèdes. Cet outil permet d'avoir une vue globale du logiciel, de naviguer dans toutes les directions, de faire un zoom sur une partie de la vue, d'appliquer des filtres, d'avoir plus de détails, de voir les relations entre les entités et de revoir l'historique.

Holten et al. [14] ont proposé un outil pour la visualisation du code source du logiciel. Ils représentent le code sur un rectangle 2D. Ils décomposent ce dernier en plusieurs petits rectangles selon la hiérarchie du logiciel. Chaque rectangle correspond à une méthode. La texture et la couleur des rectangles nous permettent de savoir le nombre d'appels entrants et sortants de cette méthode.

Wang et al. [37] ont proposé une nouvelle méthode pour visualiser les arbres (hiérarchies d'entités logicielles). Ils représentent ces derniers avec des cylindres et sphères imbriqués. Les cylindres permettent de mettre en évidence la hiérarchie et la structure des entités à visualiser et les sphères permettent de visualiser les feuilles. La couleur et la taille des cylindres et des sphères représentent des métriques de ces entités. L'utilisateur de l'outil proposé peut faire un zoom ou avoir plus de détails sur une entité.

Langelier et al. [18] ont développé un outil de visualisation nommé Verso. Ce dernier utilise des représentations $2^{1/2}$ D, c.-à-d. des parallélépipèdes rectangles 3D placés sur un plan (Figure 3). Verso permet de visualiser les paquetages. Il permet aussi de représenter les métriques des entités par la couleur, l'orientation, la hauteur, la position et la texture des parallélépipèdes. Verso offre à ses utilisateurs des actions de visualisation comme le zoom, les filtres, la demande de détails et la sélection des entités. Il permet aussi de visualiser l'évolution des logiciels.

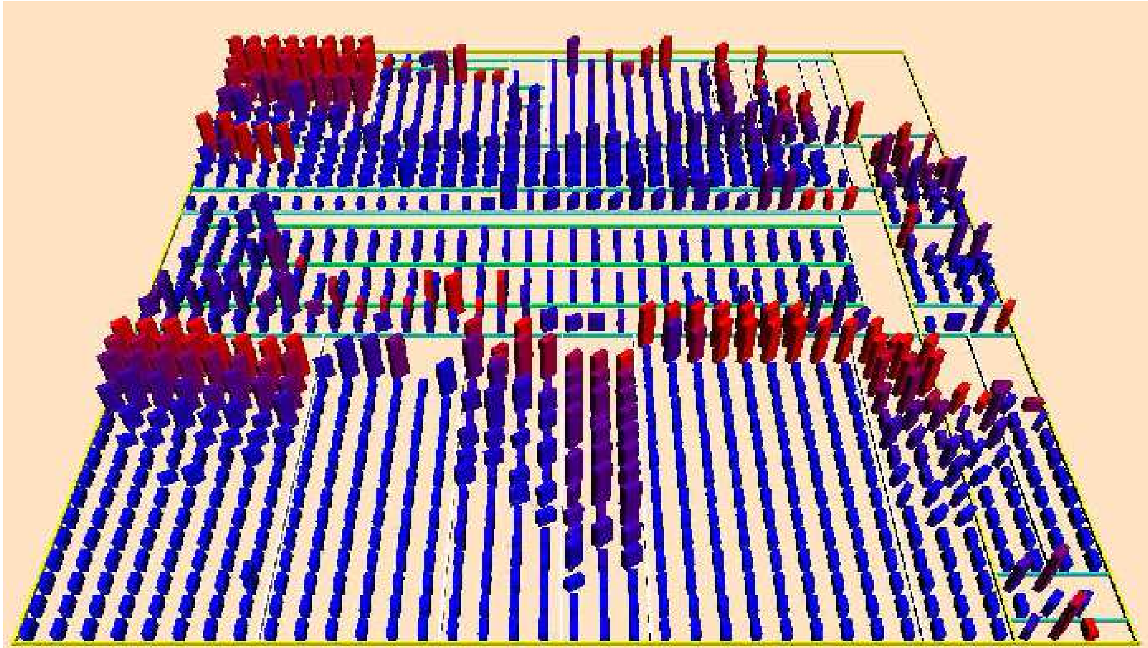

Figure 3 : Une vue de l'outil proposé par Langelier et al. [18].

2.3 Formalisation de la description des tâches de maintenance et des actions de visualisation

Nous trouvons dans la littérature, différentes méthodes pour exprimer les tâches d'analyse et les actions de visualisation appliquées sur un outil de visualisation.

Moha et al. [26] proposent une méthode nommée “DECOR” qui permet la spécification et la détection d'anomalie de conception. Chaque anomalie peut être exprimée avec une liste de règles. Ces règles sont des ensembles de propriétés des métriques, de la structure et de la sémantique. Une fois l'anomalie décrite en ce langage, “DECOR” génère automatiquement l'algorithme de détection de cette anomalie sous forme d'un code JAVA. La

description des anomalies pour “DECOR” est complexe et nécessite une expérience dans ce domaine.

Alikacem et Sahraoui [1] proposent une approche basée sur les règles qui permet la spécification et la détection des anomalies de conception. Ils ont défini un langage qui permet de spécifier les anomalies comme un ensemble de règles. Ces derniers seront reformulés pour obtenir des règles sous forme des règles de JESS. Les règles obtenues seront introduites dans le moteur d'inférence de JESS.

Marinescu [24] présente une stratégie de détection d'anomalie de conception basée sur les règles. Pour faire la détection d'une anomalie, il faut définir un ensemble de règles sur les métriques pour filtrer l'ensemble du système et retrouver le sous-ensemble qui répond aux critères. Les règles sur les métriques peuvent être absolues (par exemple (*LOC(c), HigherThan(100))*, c.-à-d., si le nombre de lignes de code de la classe *c* est supérieur à 100) ou relatives (par exemple *(WMC(c), TopValues(25%))*, c.-à-d., si la complexité de la classe *c* est parmi les valeurs du quartile supérieur).

D'autres travaux se sont intéressés à définir une taxonomie des tâches d'analyse.

Whrend et Lewis ont proposé une taxonomie d'opérateurs qui permettent de définir une tâche d'analyse [38]. Ils ont essayé de citer toutes les requêtes d'information qu'un utilisateur pourrait se poser en manipulant les données.

Amar et al. [2] ont défini des primitives d'analyse. Ces primitives permettent à l'utilisateur de formuler les demandes d'informations sur des données. Ces opérateurs sont : *Retrieve Value, Filter, Compute Derived Value, Find Extremum,* Sort, *Determine Range, Characterize Distribution, Find Anomalies, Cluster, Correlate.*

Shneiderman [31] propose une taxonomie des interactions dans une scène de visualisation. Il cite des actions qu'un utilisateur peut faire pendant un scénario de visualisation. Ces actions sont : *vue globale, zoom, détails sur demande, filtre, historique et extraction.*

Hassaine et al. [10] ont proposé une approche de génération de scénarios de détection d'anomalies, basée sur la méta-modélisation. Pour ce faire, ils ont défini un méta-modèle de tâche d'analyse et un méta-modèle de scénario de visualisation. Ils ont défini un ensemble d'opérateurs basés sur les deux travaux précédents [2] [31]. Ils ont caractérisé chaque opérateur par des paramètres qui donnent plus d'informations sur les données à manipuler. Ils ont décomposé une tache d'analyse en des buts et des sous-buts et ils ont organisé les opérateurs par des structures de contrôle.

2.4 La visualisation assistée

Les travaux que nous allons citer dans ce qui suit sont des approches proposées pour assister la visualisation en automatisant la génération des vues.

Healey et al. [12] ont proposé une approche semi-automatique pour assister la visualisation (ViA). Ce système est capable d'aider les utilisateurs à trouver les mappings optimaux de point de vue perceptuel. L'algorithme de recherche permet de trouver le meilleur mapping entre tous les mappings possibles en considérant le poids attribué par le moteur d'évaluation. L'évaluation des mappings entre les attributs graphiques et les métriques des données à visualiser se base sur des directives liées à la perception. Les principales directives sont les interférences entre les mappings de chaque métrique, la possibilité de l'application d'une tâche donnée sur un attribut graphique

Dans ViA seules les données sont mappées, mais pas la tâche d'analyse. Cet outil explore l'espace des mappings possibles entre les attributs de données et caractéristiques visuelles afin de sélectionner une solution optimale. L'exploration est guidée par la nature des données, la tâche à effectuer sur ces données, et les propriétés d'ensemble de données telles que la distribution des valeurs.
ViA utilise l'algorithme de recherche heuristique LRTA * [16], combinée avec les règles de perception, pour produire mappings visuels. L’assistance est limitée par les métaphores de visualisation supportées.
L’utilisateur de ViA doit trier les métriques des données selon leurs importances puis préciser dans quelles tâches ces métriques seront utilisées et leurs fréquences d’apparition.
Cette approche a été appliquée pour la visualisation des agents des enchères dans des applications cybercommerce [11] et pour la visualisation dans la météo (Figure 4).

Figure 4: Historiques des conditions météorologiques pour le mois de janvier dans tous les états unis d’Amérique

Mackinlay et al. [22] ont développé un système d’analyse visuel nommé Tableau. Ce système permet de générer facilement plusieurs vues. Il sélectionne automatiquement les types graphiques (barres alignées, histogramme, etc.). Tableau prend en entrée des métriques et

selon ces dernières, il construit la vue la plus appropriée. Quand un utilisateur ajoute une métrique dans une vue, cette vue sera remplacée automatiquement par une autre qui permet de visualiser les nouvelles et les anciennes métriques. Les développeurs de cet outil considèrent que l'utilisateur connaît bien les données qu'il va analyser et la tâche d'analyse qu'il va faire. L'utilisateur doit spécifier le type, le rôle et l'interpolation de ces données. La génération des vues se base sur des règles de choix des types graphiques (barres alignées, histogrammes, etc.) selon les types des données en entrées. L'avantage de cet outil est qu'il est interactif : les vues s'adaptent automatiquement à l'ajout ou à la suppression d'une donnée de la vue.

Dans le paragraphe 2.3, nous avons cité les travaux de Hassaine et al. [10]. Ils ont défini un méta-modèle de tâche d'analyse et un méta-modèle de scénario de visualisation. Cette définition avait pour but de transformer les tâches d'analyse décrite par le premier méta-modèle en des scénarios de visualisation. Nous allons reprendre des idées de ce travail pour les améliorer et aboutir à nos objectifs.

Le travail le plus proche de notre contribution est certainement d'Agrawala et al. [3]. Ils utilisent une méthode méta-heuristique et les contraintes de perception pour générer des vues pour des instructions visuelles. Ils explorent l'espace des solutions possibles et sanctionnent ceux qui violent les règles de perception. Cette approche a été appliquée pour générer un itinéraire semblable à un itinéraire dessiné à la main à partir des cartes générées automatiquement. Même si nous nous inspirons de cette approche, nous ciblons un problème très différent. En effet, les résultats générés par Agrawala et al. sont statiques et sont destinés à être imprimés. Dans notre cas, nous générons des vues de visualisation et les scénarios d'interaction pour effectuer des tâches d'analyse.

2.5 Conclusion

Tous ces travaux nous permettront de retrouver les bases sur lesquelles nous allons bâtir nos travaux. Nous utiliserons les taxonomies proposées pour décrire les tâches d'analyses et les scénarios de visualisations. Nous aurons aussi besoin de toutes les métaphores et les actions de visualisations dans les outils que nous avons vues, et nous allons nous inspirer des approches d'assistance à la visualisation.

CHAPITRE 3 : APPROCHE

3.1 Introduction

Notre but est de générer un scénario d'interaction pour un outil de visualisation afin d'effectuer une tâche d'analyse de code. Par conséquent, notre assistant de visualisation prend en entrée une description de la tâche d'analyse à effectuer et une spécification de l'outil de visualisation à utiliser. Il produit en sortie la description de l'ensemble des vues dans l'outil utilisé et le scénario d'interactions à appliquer sur ces vues. Ce scénario d'interaction est une séquence d'interactions que l'utilisateur de l'outil doit effectuer pour réaliser sa tâche d'analyse du code. La Figure 5 donne un aperçu de notre approche en termes d'entrées / sorties.

Pour mettre en œuvre notre générateur de scénario de visualisation, nous avons suivi le principe de l'ingénierie dirigée par les modèles. Sur la base de ce principe, les tâches d'analyse de code et des scénarios d'interaction sont décrites au moyen de méta-modèles indépendants, présentés dans les paragraphes qui suivent. Le méta-modèle des tâches est une abstraction des données utilisées dans l'analyse de code et les opérations qui permettent de les explorer. De même, le méta-modèle d'interaction définit les opérations habituelles que l'on peut effectuer à l'aide d'un environnement de visualisation telles que le zoom et la sélection. Nous utilisons un troisième méta-modèle pour caractériser l'outil de visualisation ciblée. Cette caractérisation se fait en décrivant les fonctionnalités qui sont fournies par l'outil. Enfin, un mécanisme de transformation dont nous avons défini un méta-modèle, permet de générer un scénario d'interactions à partir d'une tâche d'analyse.

La transformation qui mappe les tâches à des scénarios d'interaction est considérée comme un problème de satisfaction de contrainte. En effet, en utilisant un outil de visualisation, une tâche d'analyse de code peut être réalisée de plusieurs manières différentes, chacune exigeant un certain effort de l'analyste. Notre objectif est de déterminer le scénario

qui nécessite le moins d'effort. Cet effort est défini comme le coût si les différents types de contraintes sont violés. Ces violations de contraintes comprennent l'absence de fonctionnalités nécessaires et l'utilisation des opérations qui nécessitent un grand effort perceptuel ou cognitif pour l'analyste faite.

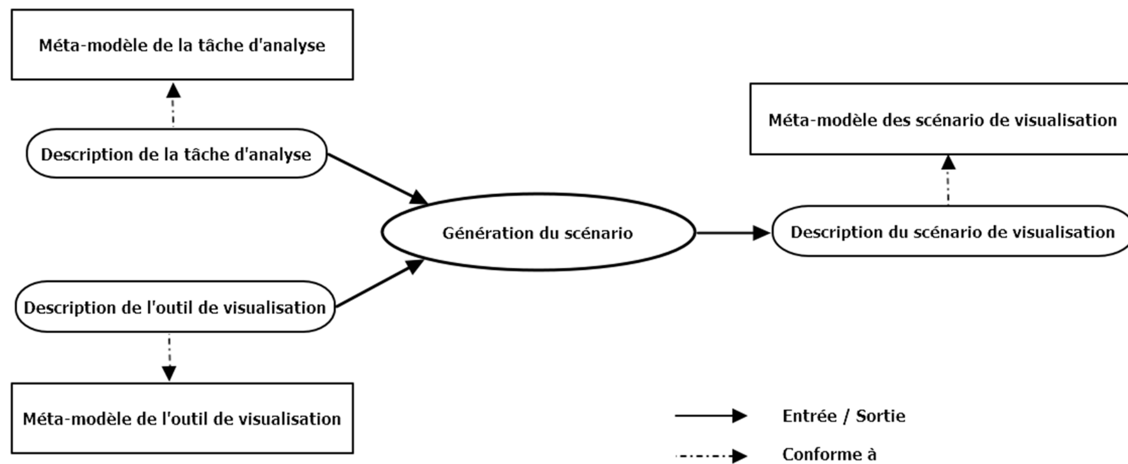

Figure 5 : Aperçu de notre approche

3.2 Description des tâches d'analyse

La tâche d'analyse permet de décrire rigoureusement le déroulement d'une tâche d'analyse sur un code en considérant sa réalisation comme une analyse des données. Une tâche d'analyse de code pourrait être définie comme le processus d'exploration d'un ensemble de données extraites du code source. Ces données sont reliées à des entités du code d'un programme tel que les classes, interfaces, méthodes et déclarations. Ils décrivent globalement les propriétés des entités (nom, type, couplage, la cohésion, la taille, etc.), leurs relations (association, appel, héritage, etc.) et leur structure (divers points de vue architecturaux).

L'exploration de données obéit généralement à un ou plusieurs objectifs et est réalisée par des séquences d'opérations de base.

3.2.1 Méta-modèle

À partir de ces observations, nous proposons un méta-modèle qui définit une syntaxe abstraite pour la description des tâches d'analyse. Le méta-modèle est représenté sur la Figure 6. Il est composé de deux parties: la description des données à analyser et la description de la tâche d'analyse.

3.2.1.1 Description des données de la tâche d'analyse

Les données sont décrites en termes d'entités typées qui contiennent éventuellement d'autres entités (par exemple, package - package, package - classe, classe-méthodes, méthode-déclarations). Chaque entité est décrite par des propriétés. Ces propriétés sont quantitatives (indicateurs de taille, le polymorphisme, couplage, etc.). Les entités peuvent être liées par différents types de relations (appel, l'héritage, etc.).

Par exemple, pour décrire un programme orienté objet, nous pouvons décrire notre environnement comme un ensemble d'éléments que nous pouvons les appeler classe. Ces classes sont composées d'autres éléments que nous appelons méthodes.

3.2.1.2 Description de la tâche d'analyse

L'analyse est décrite indépendamment de toute méthode d'exploration de données (automatique ou manuelle). La description commence par un but qui contient des sous-buts, des primitives et des opérations de contrôle.

3.2.1.2.1 Les buts

La description commence par un but qui pourrait être affiné de manière récursive en une liste de sous-buts. Pour atteindre chaque but, l'analyse doit effectuer un ensemble d'opérations. Les opérations sont soit des actions primitives (appelons les primitives) ou des opérations de contrôle.

3.2.1.2.2 Les opérations de contrôle

Nous limitons les opérations de contrôle à des instructions d'itération (for, for each) et des instructions de sélection (if). Les utilisateurs de ce langage, les développeurs, sont familiarisés avec ce type de structuration empruntée des langages de programmation.

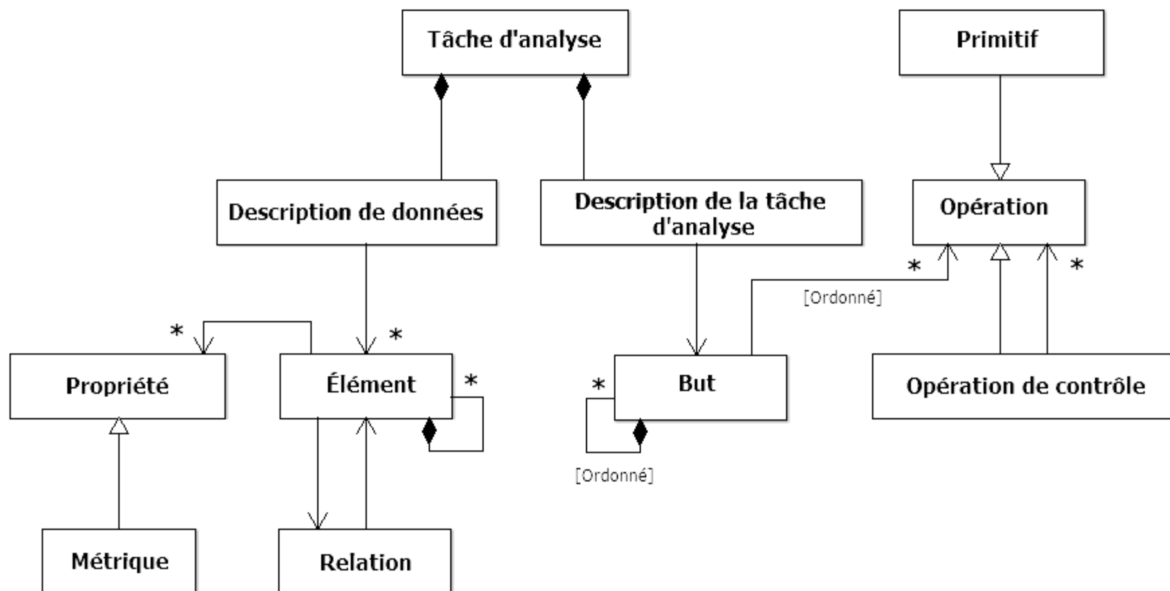

Figure 6 : Vue simplifié du méta-modèle des tâches d'analyses

3.2.1.2.3 Les primitives

Les primitives utilisées sont celles proposées par Amar et al. [2]. Dans ce travail, les auteurs proposent une liste de 10 opérations d'analyse de bas niveau pour explorer, caractériser et modifier un ensemble de données. Nous avons retenu huit d'entre elles que nous vous présentons dans le tableau 1. Nous n'avons pas inclus les primitives « Trouver l'extremum » (Find extremum) et « Trouver l'anomalie » (Find Anomalies), car ils peuvent être obtenus en combinant certaines des huit autres primitives. Nous avons ajouté une primitive générique « Operation » qui permet d'appliquer des opérateurs arithmétiques et affecter les résultats à d'autres primitives, par exemple, de fusionner les résultats de deux filtres utilisant l'opérateur d'union ou en ajoutant les valeurs de deux métriques d'une entité sélectionnée. Le tableau 1 montre les définitions de ces primitives ainsi que la syntaxe concrète que nous utilisons pour exprimer une tâche d'analyse. Nous avons ajouté des paramètres à ces primitives. Ces paramètres permettent de préciser l'ensemble sur lequel la primitive sera appliquée, les conditions à vérifier ou les objets retournés.

Par exemple, nous allons considérer un logiciel composé de plusieurs éléments qui sont les classes et que ces classes sont composées de sous éléments qui sont les méthodes. Nous allons aussi considérer que les éléments « méthode » ont une métrique LOC (nombre de lignes).
Pour connaître le nombre de lignes d'une méthode *m*, nous utilisons la primitive Retrieve_value(*m*, LOC, loc_m).
Pour trouver, parmi toutes les méthodes d'un programme (Methods), celles qui ont plus que 10 lignes de code, nous utilisons la primitive Filter(Methods, LOC<10 , more_10_set).

Tableau 1: Description des primitives

Primitive	**Description**	**Syntaxe**
Retrieve_value	Cette primitive permet de retourner la valeur d'une métrique d'un élément ou une relation donnée.	Retrieve_value(<input>,<metric>,<output_val>)

Filter	Retourne les éléments ou les relations qui satisfont les conditions données.	Filter(<input_set>,<condition>, <output_set>)
Compute_derived_value	Retourne la valeur de l'application d'une fonction à un ensemble donné.	Compute_derived_value(<input_set>, <function> ,<output>)
Sort	Retourne les éléments ou les relations données en entrée dans l'ordre croissant ou décroissant selon une métrique donnée.	Sort(<input_set>,<metric>, increasing/decreasing)
Determine_range	Permets de déterminer la distribution des valeurs d'un attribut	Determine_range(<input_set>, <metric>,<output_set>)
Caracterise_distribution	Permets de caractériser la distribution des valeurs d'une métrique quantitatives sur un ensemble d'éléments ou de relations.	Caracterise_distribution (<input_set>,<metric>)
Cluster	Retourne les clusters d'éléments ou de relations qui satisfont les mêmes conditions.	Cluster(<input_set>,<metric>, <output_set>)
Correlate	Retourne la relation, si elle existe, entre deux ou plusieurs métriques ou relations.	Correlate(<input_set>,<metric1>, <metric2>)
Operation	Retourne le résultat de l'application d'une opération arithmétique donnée sur deux opérandes	Operation (<operation>, <output_value>, <input_operand1>, <input_operand2>)
Achieve	Réaliser un but	Achieve(<Goal_name>,<input_set>, <output_set>)

3.2.2 Utilisation du langage proposé

Pour utiliser notre approche, l'utilisateur doit exprimer sa tâche manuellement en respectant le méta-modèle des tâches d'analyse (Figure 6) et en utilisant les primitives présentés dans le tableau 1. Chaque donnée et chaque primitive vont avoir un code correspondant pour faciliter le traitement et l'analyse (voir Figure 10 à droite : Le vecteur à droite représente les codes consécutifs de chaque donnée, but ou primitive de la tâche d'analyse de détection de Blob).

3.2.3 Exemple de description d'une tâche d'analyses

Nous allons décrire la tâche d'analyse de détection de Blob. Un Blob est un anti-patron connu dans la communauté du logiciel [4]. Un Blob est une classe qui monopolise l'exécution d'un programme alors que les autres classes encapsulent les données (appelées classes de données). C'est une classe "contrôleur", complexe et non cohésive, associée à des classes de données simples.

Nous avons exprimé la méthode de détection d'un Blob avec notre langage (tableau 2).
Cette tâche d'analyse a pour objectif principal la détection de Blob (Détection de Blob) qui s'applique sur tout le système analysé (Système) et dont l'exécution retourne l'ensemble de Blob détecté (EnsembleBlob). Cet objectif est raffiné en deux sous-buts : (1) détecter les classes de type "contrôleur" et (2) vérifier que ce sont des classes de données.
Le premier sous-but (Détecter les class contrôleuses) permettra la détection des classes candidates pour être des Blob (CC) trouvés dans le système. Ce but sera obtenu en sélectionnant les entités du système qui ont une complexité élevée (WMC) (Weighted Methods per Class [5]), une cohésion LCOM5 de basse à moyenne (Lack of Cohesion in Methods [13]), et une faible profondeur de l'héritage DIT (Depth in Inheritance Tree [5]).
Le second sous-but (vérifier que ce sont des classes de données) permettra de vérifier que les candidats Blob sont liés à des classes de données. Pour cela, pour chaque candidat Blob c,

nous sélectionnons les classes qui sont appelées par c. Ensuite, nous identifions dans l'ensemble retourné, les entités qui ont les caractéristiques de la classe de données : WMC et DIT basses. Si le nombre de ces classes est élevé, alors le candidat est ajouté à la liste de Blob. La description statique comporte la description des données que nous allons analyser. Dans notre cas, nous avons des classes qui ont comme métriques WMC, LCOM5 et DIT. Ces trois métriques sont de type quantitatif.

Tableau 2: Description de la tâche de détection de Blob

Description de la tâche d'analyse	Description des données

Goal (Blob_detection, Input_system
output_Blobset, Système)
{
Achieve (Detect_controller_class, Input_system, CC)
Achieve (Is_data_class,CC , output_Blobset)
}
	Element class
	Metric WMC class quantitative
	Metric LCOM5 class quantitative
	Metric DIT class quantitative

Goal (Detect_controller_class, Input_system, CC)
{
Filter(Input_system ,
isHigh WMC AND
Between LOW MEDIUM LCOM5 AND
isLow DIT,
CC)
}	

Goal (Is_data_class, CC, output_Blobset)
{Foreach(c, CC) {
Filter(Input_system, isCalled(c), Callers)
Filter(Callers, isLow WMC and isLow DIT,
Data_class_callers)
Compute_derived_value(Data_class_callers,
count, Num)
IF (isHigh Num) {
operation (+,output_Blobset, output_Blobset, c)
// output_Blobset=output_Blobset+c } } }	

3.3 Description des outils de visualisation

Il existe de nombreux environnements de visualisation pour réaliser diverses tâches d'exploration et d'analyse de code. Bien qu'ils aient été conçus dans des contextes différents et pour réaliser diverses tâches, il est possible de définir un méta-modèle qui permet de décrire la majorité d'entre eux. La Figure 7 montre le méta-modèle des outils de visualisation. Un outil de visualisation offre une ou plusieurs vues. Chaque vue affiche un ou plusieurs types de représentation graphique (différents types de nœuds, liens, textes, etc.).

Les représentations graphiques pourraient inclure d'autres représentations. Dans certains outils de visualisation, par exemple, une représentation du package contient des représentations de classe. Ils pourraient également être liés à d'autres représentations. Par exemple, lors de la représentation des diagrammes de classes, les nœuds de classe sont liés à différents nœuds relationnels (généralisation, association, etc.). En outre, les représentations graphiques ont des attributs graphiques sur lesquels les données sont mappées.

Les outils de visualisation offrent généralement des fonctionnalités pour interagir / explorer des vues et des représentations graphiques. Par exemple, on peut effectuer un zoom avant,

zoom arrière, ou naviguer dans une vue. Pour les représentations graphiques, des exemples d'interaction sont la *sélection (selection), le marquage (tagging), les détails-sur–-demande(details_on_demand)*, etc. Lors de la description d'un outil, si la fonction est présente, une estimation de l'effort de son utilisation est spécifiée. Par exemple, si la fonctionnalité *détails-sur-demande (details_on_demand)* est implantée de manière à ouvrir un fichier dans une fenêtre séparée, ceci nécessite plus d'effort de l'analyste (changement de contexte, etc.) que si les détails sont présentés dans un pop-up.

À chaque fois, que nous allons utiliser un outil de visualisation, nous devons préciser les représentations graphiques, les attributs graphiques, les actions possibles, les actions automatiques qu'il offre. Il faut aussi spécifier les caractéristiques de toutes ces descriptions. Par exemple, il faut préciser le type de l'attribut graphique : la couleur, l'orientation, etc. Il faut aussi spécifier pour chaque attribut graphique s'il est distinguable dans une vue globale ou locale. Nous devons aussi préciser l'effort que l'utilisateur doit fournir pour réaliser chaque action possible dans l'outil.

Pour chaque outil nous devons définir l'effort des actions disponibles comme *navigate, zoom, tag, details on demand, change mapping, access sub-element* .

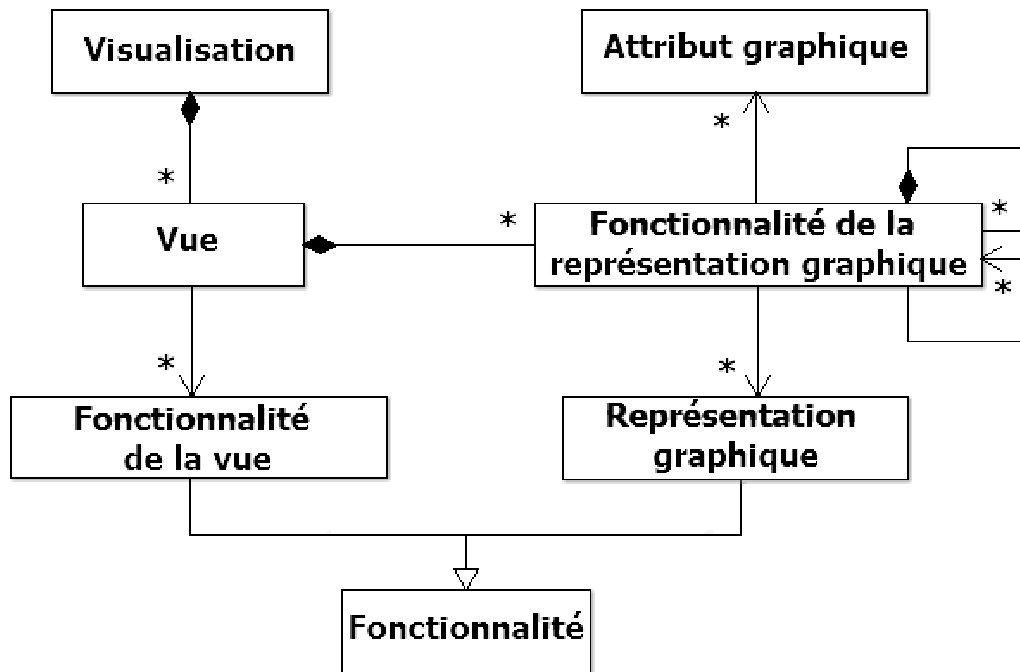

Figure 7 : Vue simplifié du méta-modèle des outils de visualisation

3.3.1 Exemple de description d'un outil de visualisation

L'outil de visualisation ciblée est Verso [18] que nous avons présenté dans l'état de l'art.

La description de l'outil de visualisation est l'une des deux entrées du processus de génération de scénarios de visualisation telle que représentée à la Figure 5. Cette description doit être faite suivant le méta-modèle que nous avons défini dans la Figure 7. Verso permet de définir un ensemble de points de vue, tous du même type. Une vue affiche les classes d'un système orienté-objet. Les classes peuvent être mappées à des boîtes 3-D ou à des cylindres. Trois attributs graphiques sont utilisés pour représenter visuellement les métriques des classes: hauteur, orientation, et couleur des boîtes (Figure 8-gauche). Dans le cas de cylindres,

l'orientation ne peut pas être utilisée. Lorsque plus de trois (respectivement deux) métriques doivent être mappées à des attributs graphiques, il est possible de créer une ou plusieurs vues où les attributs graphiques sont associés à d'autres métriques. Toutes les classes du système sont placées selon un algorithme de mise en place qui respecte l'architecture des packages du système (Figure 8- droite).

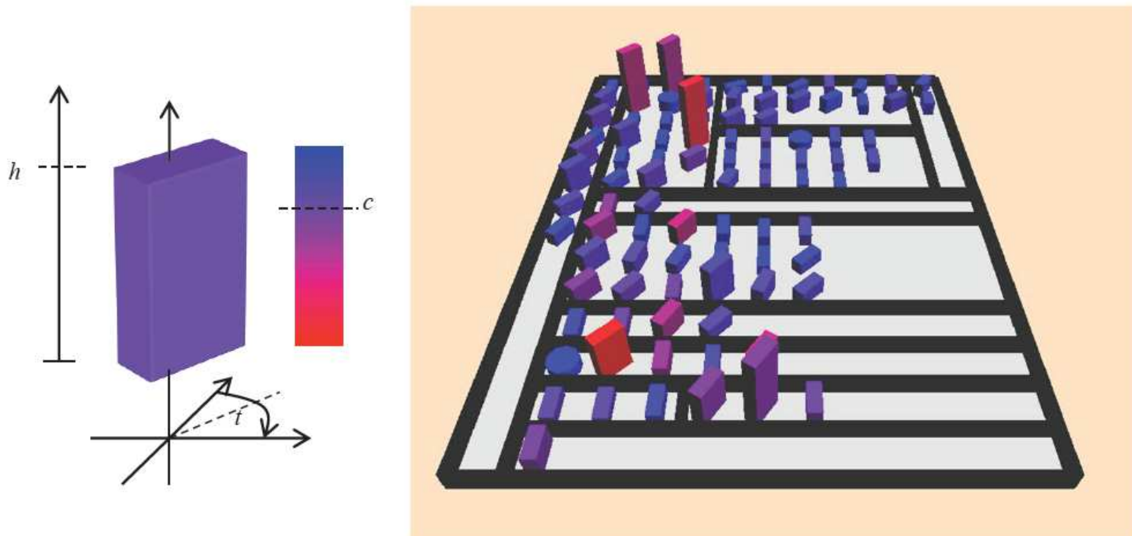

Figure 8 : Vue et représentation graphique de Verso

Verso offre l'interaction *détails-sur-demande (Details_on_demand.)* . Il permet également de *sélectionner (Select)* et *tagger (Tag)* des entités. Il est possible d'utiliser deux types de filtres automatisés, une basée sur des seuils métriques et l'autre pour afficher les entités liées à une entité donnée.

L'élément *Naviguer (Navigate)* est également assuré par la possibilité de déplacer librement la caméra dans une demi-sphère au-dessus du plan d'affichage. Ce mécanisme peut être utilisé pour s'approcher *(zoom-in)* ou s'éloigner *(zoom-out)* d'un élément. La majorité des interactions offertes par Verso sont simples à utiliser et ne nécessitent pas un effort significatif. La seule exception est *Détails-sur-demande (Details_on_demand)*. Cette interaction nécessite un effort supérieur (plus d'interaction et gestion de nombreuses valeurs associées à de nombreuses entités). Une autre interaction dans Verso qui nécessite plus

d'effort est le changement de vue lorsque la limite de métriques mappées est dépassée sur le même attribut graphique. Dans ce cas, les métriques vont être mappées dans 2 ou plusieurs vues. Chaque vue va contenir des métriques différentes. Pour visualiser toutes les métriques, il faut alterner d'une vue à une autre. Ce changement de vue est considéré comme un grand effort en raison de l'effort cognitif associé au changement de contexte pour construire la vue demandée.

3.4 Modélisation des scénarios de visualisation et des vues

Le scénario de visualisation permet de décrire le déroulement des actions de visualisation qui seront appliquées dans les vues. La Figure 9 montre le méta-modèle proposé pour les scénarios de visualisations et pour les vues. Les descriptions des scénarios de visualisation sont définies comme étant un ensemble d'interactions appliquées dans une ou plusieurs vues. Dans les vues, les représentations graphiques sont mappées aux entités définies dans la tâche d'analyse, et leurs caractéristiques aux propriétés de ces entités. Il y a deux possibilités: (1) le mapping est codé en dur, c'est à dire, les représentations et leurs caractéristiques sont mappées à des types et des propriétés des entités, et (2) le mapping pourraient être défini par les analystes avant de commencer l'analyse des données. Nous nous intéressons seulement à des outils offrant la deuxième possibilité. Notre approche peut encore être appliquée à des outils avec des mappings codés en dur tant que la génération est limitée seulement à l'analyse et ne comprend pas des vues.

Les scénarios d'interaction seront exécutés par un analyste. Ce dernier effectue une série d'interaction. Ces interactions pourraient être d'autres scénarios (à des fins de modularité et la réutilisation), des éléments simples d'interaction tels que le *zoom-in*, et des structures d'itération ou de sélection d'un ensemble d'interaction. Dans tous les cas précédents, les étapes sont séquentielles.

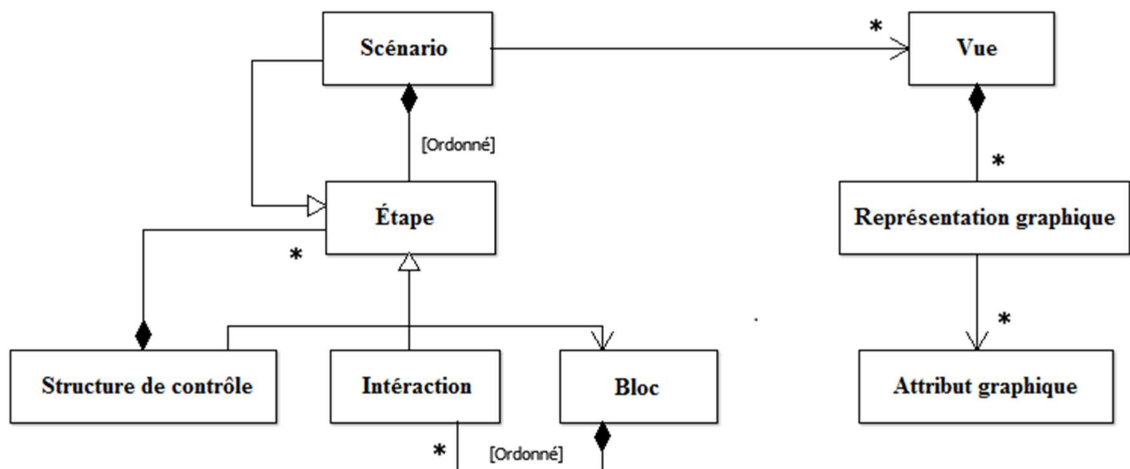

Figure 9 : Vue simplifié du méta-modèle des vues et des scénarios de visualisation

Dans quelques cas, une étape est constituée d'un ensemble (bloc) d'interaction qui pourrait être effectué dans un ordre défini par l'utilisateur. Par exemple, l'analyste pourrait utiliser la *navigation*, *zoom*, et la *sélection* pour effectuer un filtrage lorsque la fonction de filtre n'est pas fournie par un outil.

Pour déterminer quels sont les éléments d'interaction de base qui sont nécessaires pour exprimer des scénarios d'interaction, nous nous sommes inspirés de la taxonomie proposée par Shneiderman [31]. Dans ce travail, sept éléments d'interaction, appelés tâches par l'auteur, sont proposés, discutés et appliqués à diverses représentations graphiques des données (1-, 2- et 3-D, arbre, réseau, temporelle, etc.). Nous avons ajouté à la taxonomie proposée par Shneiderman [31] d'autres actions de bas niveau que nous jugeons nécessaires, par exemple, les actions *tag, select, change_mapping.* La liste obtenue est montrée dans le tableau 3. Les interactions s'appliquent aux représentations graphiques, à leurs caractéristiques et aux vues.

Pour mieux comprendre ce langage, nous allons montrer quelques exemples :

Overview(class) nous indique qu'il faut faire un grand plan sur toutes les classes.

Apply_automatic_filter(C, find_related_method) nous indique qu'il faut appliquer le filtre *find_related_methode* sur l'ensemble C.

Navigate nous indique qu'il faut naviguer entre les entités. Cette action est souvent associée à d'autres actions pour localiser une entité.

Tableau 3 : Interaction de visualisation

Interaction de visualisation	Description
Details_on_demand	Donner une valeur d'un attribut ou une entité affichée
Tag	Marquer une ou plusieurs entités
Select	Sélectionner une entité
Filter	Appliquer un filtre sur un ensemble d'entités
Access_to_sub_element	Accéder à une sous-entité si elle existe.
Apply_automatic_filter	Appliquer un filtre automatique
Navigate	Naviguer entre les entités
Zoom	Se rapprocher ou s'éloigner d'une entité
Check_if	L'utilisateur vérifie si les entités satisfont une condition
Change_mapping	Changer les mappings entre les métriques et les attributs graphiques
Overview	Afficher la scène avec toutes les entités (ou un sous-groupe)
Mapping	Initialiser le mapping entre les métriques et les attributs graphiques
History	Explorer l'historique du scénario de visualisation
Do_function	L'utilisateur applique une fonction spécifiée

3.5 Génération des scénarios de visualisation

Un scénario d'interaction est une séquence d'interactions que l'utilisateur de l'outil doit effectuer pour réaliser sa tâche d'analyse de code.

3.5.1 Génération des Scénarios comme un problème d'optimisation

Pour générer un scénario d'interaction pour une tâche donnée, notre générateur doit mapper trois types d'information: (1) les données dans des vues, (2) les buts en scénarios, et (3) les opérations d'analyse en interactions de visualisation. Bien que le deuxième mapping

soit simple, de nombreuses possibilités doivent être explorées pour le premier et le troisième. Pour le mapping des données, les représentations graphiques doivent être associées avec des entités de données. Si les représentations sélectionnées appartiennent à plus d'une vue, il sera nécessaire de basculer entre les vues lors de la réalisation du scénario de visualisation.

Une fois que le mapping entre une représentation graphique et une entité de données est fait, l'ensemble des attributs disponibles dans la représentation graphique qui pourraient être mappés aux propriétés d'entité des données sera obtenu. Si la représentation n'a pas assez d'attributs à mapper aux propriétés des données, d'autres mappings seront générés ainsi qu'une commutation entre eux est nécessaire lors de l'analyse. Selon l'outil utilisé, le nombre de mappings possibles pourrait être très grand.

Le mapping des primitives avec les interactions de visualisation est également difficile. La description d'une tâche d'analyse contient de nombreuses primitives. Chaque primitive pourrait être effectuée par un des nombreux groupes d'interaction de visualisation (relation un-à-plusieurs). Par exemple, comme le montre le tableau 4, la primitive de filtrage (*Filter*) peut être mappée à plusieurs groupes d'interaction de visualisation en fonction l'outil choisi. Ces transformations représentent différentes façons de réaliser un filtre dans l'outil de visualisation cible (dans ce cas Verso).

Tableau 4 : Exemples de transformations possibles de la primitive filtre (Filter)

Transformations (ensemble d'action = Bloc d'action)	Code du Bloc
Overview Apply automatic filter Tag	B1
Overview Block {Navigate, Zoom, Select, Check if, Tag }	B2
Overview Block {Select, Check if, Tag}	B3
Overview Block {Select, Details on demand, Check if, Tag }	B4

Change mapping Overview Block {Select, Check if, Tag}	B5

Les mappings de données et des primitives ne sont pas indépendants, ce qui rend la situation encore plus compliquée. La difficulté de réaliser des interactions de visualisation varie en fonction des attributs graphiques choisis. En revanche, le choix de représentations et des attributs graphique est influencé par les primitives qui doivent être réalisées en les utilisant. La compatibilité entre les groupes d'interactions de visualisation choisis est aussi une contrainte.

Compte tenu de toutes les possibilités et l'interdépendance entre les mappings, nous allons considérer la génération des scénarios de visualisations comme un problème d'optimisation. En effet, l'objectif est de trouver le scénario qui nécessite le moins d'effort d'analyste, qui enfreint le moins les nombreuses contraintes de mapping d'élément. Pour pouvoir trouver le meilleur scénario de visualisation entre le grand nombre de possibilités, nous avons implémenté la génération de ce scénario en utilisant un algorithme génétique (AG) [9].

L'idée de base d'un algorithme génétique est d'appliquer les mécanismes d'évolution sur un ensemble de solutions initiales pour trouver la solution optimale. Cet algorithme est inspiré de l'évolution biologique, pour dériver de nouvelles et peut-être de meilleures solutions (appelées chromosomes) à partir de la population initiale.

La dérivation est une séquence d'ensembles de solutions appelés populations, obtenues chacune soit par "mutation" soit par croisement de la population précédente en commençant par la population initiale. L'aptitude de chaque solution est mesurée par une fonction objective. À chaque génération, l'algorithme sélectionne des paires de solutions en utilisant une méthode de sélection qui donne la priorité aux solutions les plus convenables. Sur chaque paire, il applique selon une probabilité deux opérateurs: le croisement et la mutation, en utilisant une probabilité associée à chaque opérateur. Chaque paire de solutions sélectionnée produit deux

nouvelles solutions qui sont incluses dans la prochaine génération. L'algorithme s'arrête si un critère de convergence est satisfait ou si un nombre fixe de générations est atteint.

Tous les algorithmes génétiques suivent le modèle générique décrit ci-dessus. Selon la nature du problème, le codage des solutions, la définition des opérateurs et la définition de la fonction de calcul de coût sont différents. Dans le reste de ce chapitre, nous présentons chacun de ces aspects pour la génération du scénario de visualisation.

3.6 Codage du scénario de visualisation

Nous allons expliquer le codage du scénario de visualisation en nous basant sur la tâche d'analyse de détection de Blob, décrite dans le paragraphe 3.2.3 et dans le tableau 2.

Chaque but ainsi que ces primitives vont être listés dans un vecteur par des codes. Par exemple, le premier but (Détection de Blob) va avoir le code G1. Le deuxième but va avoir G2. Les deux premières primitives dans le deuxième but (Filter) vont avoir le code P1.

Les données utilisées dans cette tâche vont aussi être listées dans le vecteur. Les classes sont des éléments codifiés par E1. Les métriques de ces classes WMC, LCOM5 et DIT vont être codifié consécutivement M1, M2 et M3.

À la fin, toute la tâche d'analyse sera listée dans ce vecteur qui est représenté dans la Figure 10 à gauche.

Dans le processus de production, chaque chromosome (qui peut être une solution potentielle) est représenté par un vecteur de gènes (voir la Figure 10 avec les vecteurs de la solution 1, solution 2,… solution n). Ces solutions sont une combinaison des possibilités de mappings du vecteur initial qui représente la tâche d'analyse (Figure 10 à gauche). Par exemple, les deux premières primitives dans le deuxième but (Filter) peuvent avoir plusieurs mappings possibles cités dans le tableau 4.

Toutes ces possibilités doivent être présentes dans une des solutions initiales. Filter a comme code P1 et a cinq possibilités de mappings (tableau 4). Chacune de ces possibilités, qui est un

bloc d'interaction, est codifiée de B1 à B5 et doit être présente dans au moins une des solutions (Figure 10 de la solution 1 à solution n).

Comme mentionné précédemment, le mapping des buts est simple.

Les mappings de chaque élément et de chaque métrique utilisés dans cette tâche vont aussi être présents dans les solutions. Dans notre cas, l'outil utilisé est Verso. Verso offre des parallélépipèdes rectangles 3D pour représenter les éléments (dans ce cas les classes). Ces parallélépipèdes permettent de représenter les métriques des éléments par la couleur, la hauteur et l'orientation. Chaque métrique peut être mappée à un de ces derniers attributs graphiques. Par exemple, M1, qui est la métrique WMC, peut être mappée à la couleur (codé AG1), la hauteur (codé AG2) et l'orientation (codé AG3).

La population initiale des solutions est générée aléatoirement.

3.7 Obtention des nouveaux scénarios

À chaque génération, de nouvelles solutions sont dérivées de celles qui existent déjà. Premièrement, des paires de scénarios possibles sont choisies au hasard en utilisant la stratégie de sélection par roulette. La sélection par roulette permettra de choisir les solutions proportionnellement à leur coût calculé (fitness). Après, deux nouveaux scénarios sont dérivés à partir de croisement et/ou de mutation des deux scénarios choisis.

Une fois que les parents sont sélectionnés, le croisement est appliqué avec une certaine probabilité. Nous utilisons deux points de croisement (Figure 11). Deux points sont choisis de façon aléatoire, ce qui permet de diviser les vecteurs parents en trois parties. Le premier enfant prend les parties 1 et 3 du premier parent et la partie 2 du second. La deuxième enfant fait le contraire.

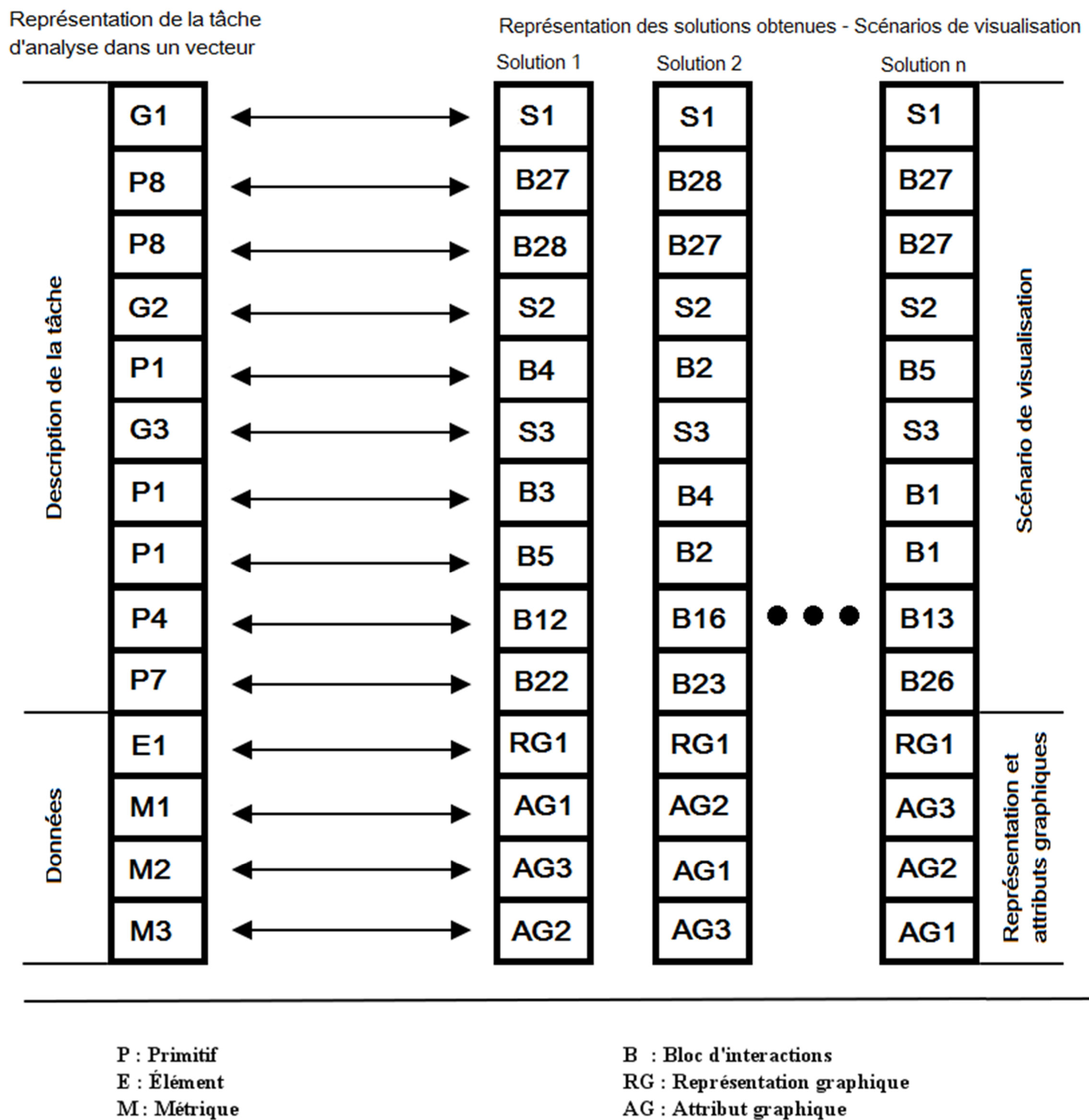

Figure 10 : Codage du scénario de visualisation – Population initial (solutions 1 à n)

Les descendants, ou les parents si le croisement n'est pas appliqué, sont mutés avec une certaine probabilité (en général beaucoup plus faible que celle du croisement). La mutation consiste à choisir un ou plusieurs gènes de manière aléatoire et à changer leur mapping. Si le gène est une primitive d'analyse, le bloc d'interaction de visualisation correspondant sera changé avec un autre bloc de manière aléatoire. Pour les éléments et les métriques, les mappings sont également modifiés de façon aléatoire.

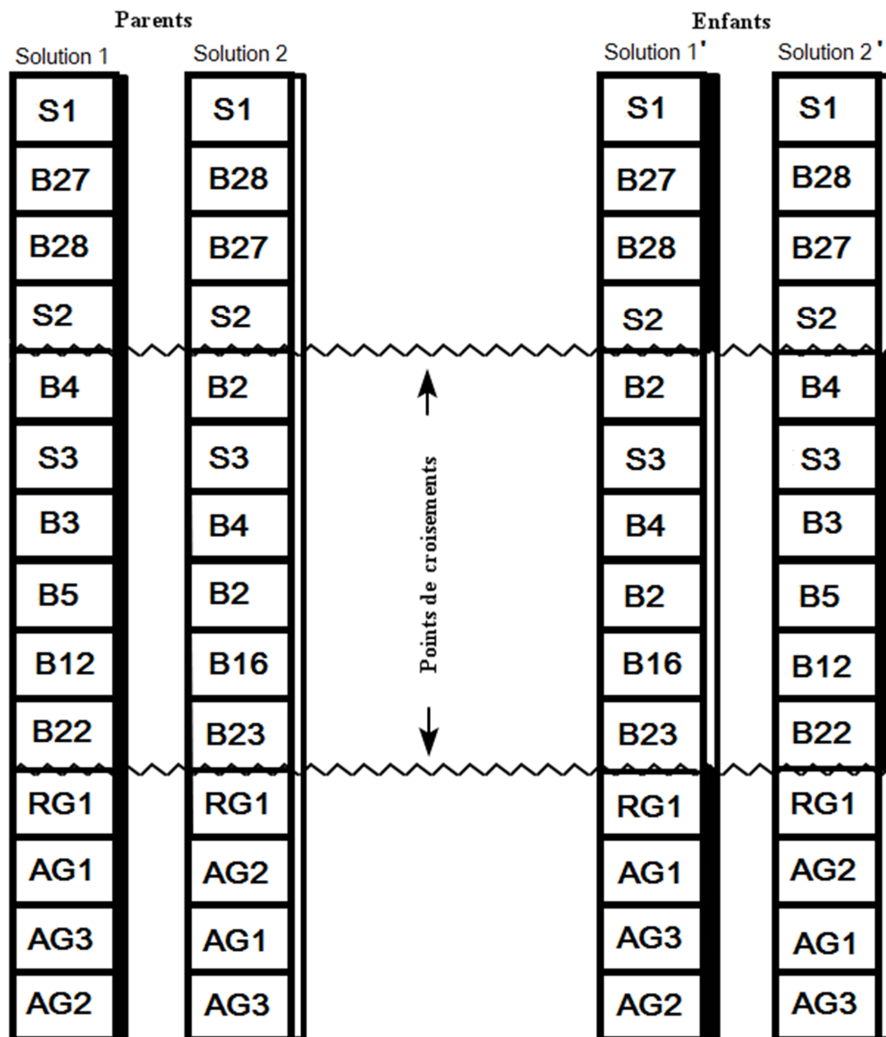

Figure 11: Opération de croisement

3.8 Évaluation des scénarios de visualisation

Notre objectif est de générer le scénario d'interaction de visualisation qui nécessite le moins d'effort lors de la réalisation de la tâche d'analyse avec un outil de visualisation. Cet effort d'analyse est difficile à mesurer. Cependant, du point de vue de la perception, il existe certains choix de visualisation qui peuvent rendre l'interaction avec les données plus difficile.

Dans ce contexte, nous avons rassemblé un ensemble de contraintes de perception et leurs coûts. Ces contraintes concernent le mapping des éléments et des métriques aux objets graphiques ainsi que les primitives aux blocs d'interaction de visualisation. Ils sont classés en trois catégories en fonction de leur impact sur l'effort d'analyse.
La première catégorie comprend les contraintes dont les violations rendent le scénario d'interaction très difficile à réaliser. Ces contraintes ont une valeur de pénalité élevée. La deuxième catégorie concerne les contraintes dont les violations augmentent modérément l'effort d'analyse. Ces contraintes sont liées aux mappings des primitives. Leurs valeurs de pénalités sont faibles à moyennes.
La troisième catégorie concerne les contraintes du mapping de données qui ont des valeurs de pénalités faibles à moyenne.
Pour chaque catégorie, les valeurs de pénalité des contraintes sont précisées, en tant que paramètres, en utilisant des intervalles disjoints de coûts (respectivement [100, 150] pour la première catégorie et [0, 50] pour la deuxième et la troisième catégorie).
Cela garantit que, quels que soient les choix, les violations des contraintes de la première catégorie sont plus pénalisées que ceux des deux autres catégories.

Avec ces contraintes, notre fonction de calcul du coût évalue un scénario de visualisation en calculant le score global correspondant aux violations de contraintes. Dans le reste de ce chapitre, nous allons décrire les contraintes que nous avons définies. Nous pouvons ajouter à cet ensemble de contraintes d'autres contraintes sans changer le processus de génération.

3.8.1 Les contraintes des scénarios irréalisables (C1)

Toute solution obtenue dans toutes les générations est une proposition pour la transformation des primitives et les données de la tâche d'analyse en des interactions de visualisations et des vues. Certaines de ces propositions représentent des scénarios qui ne peuvent être exécutés par l'analyste avec l'outil cible. Nous avons identifié quatre de ces situations.

3.8.1.1 L'élément d'interaction n'est pas offert par l'outil de visualisation (C1.1)

Chaque primitive dans la description de la tâche d'analyse est transformée en un bloc d'interaction de visualisation. Parfois, l'une des interactions incluses dans un scénario n'est pas proposée par l'outil de visualisation cible, ce qui peut rendre le scénario très difficile à réaliser et même impossible. Nous considérons ces situations comme des violations de contraintes et nous pénalisons le scénario correspondant en conséquence.

On pourrait supposer qu'une étape de filtrage des mappings entre les primitives et les blocs d'interactions, pour un outil donné, peut empêcher ces cas. Cependant, pour de nombreux outils existants, l'élimination de ces mappings pourrait diminuer les possibilités d'obtenir de nouvelle solution. En outre, l'analyste compense généralement, avec un effort important l'absence d'un élément.

Par exemple, l'opération filtre peut être transformée en un bloc comprenant l'interaction de marquage (Tag). Si l'outil n'offre pas cette interaction, lors de l'analyse, l'analyste peut utiliser un papier et un crayon pour noter les représentations qu'il veut marquer, mais l'effort résultant sera élevé.

3.8.1.2 Incompatibilité entre les interactions de visualisation et les attributs graphiques (C1.2)

Les attributs graphiques ont différentes propriétés de perceptions. Certains sont visibles dans une vue globale, d'autres dans une vues local ou étroite. Par exemple, si des classes Java sont mappées sur des rectangles et leur taille à la texture de ces rectangles (par exemple, une jauge dessinée sur un rectangle), il est difficile de distinguer les différences de textures (et donc de la taille) lors de l'observation d'une vue avec des centaines d'entités d'un système (vue globale).

Cependant, quand on regarde de près les rectangles, nous pouvons comparer les différences de texture et détecter facilement la différence de taille. Nous avons une incompatibilité entre une interaction et un attribut graphique lorsque l'interaction nécessite une analyse sur une vue globale pour une métrique qui est mappée à un attribut visible uniquement dans une vue locale (violation globale-locale). L'inverse (locale-globale) est également une violation.

3.8.1.3 Fausse hypothèse sur une métrique non mappée (C1.3)

Lorsque la tâche d'analyse nécessite de mapper plus de métriques que les attributs graphiques disponibles, deux stratégies sont possibles. La première consiste à mapper plus d'une métrique à un même attribut graphique. Cela nécessite de basculer entre les mappings lors de l'analyse avec un certain effort. La seconde possibilité est de ne pas mapper la métrique et utiliser l'interaction *Détail-sur-demande (Detail_on_demand*) pour afficher les valeurs si nécessaire lors de l'analyse. La violation de contrainte ici est quand une interaction considère la métrique mappée alors qu'elle ne l'est pas.

3.8.1.4 Fausse hypothèse sur une métrique mappée (C1.4)

Cette contrainte est similaire à C1.3. Cette contrainte sera violée quand le scénario utilise Détail-sur-demande pour accéder aux valeurs d'une métrique qui est déjà mappée à un attribut graphique, donc visible dans la vue.

3.8.2 Les contraintes d'utilisation des interactions de visualisation (C2)

Quand un analyste exécute un scénario, il effectue chaque étape du scénario avec un certain effort. Les différentes quantités d'effort sont considérées comme des violations de cette catégorie de contraintes.

3.8.2.1 L’effort des interactions (C2.1)

Chaque interaction dans un scénario de visualisation nécessite un certain effort de l'analyste pour l’effectuer. Cet effort n’est pas universel et dépend de l'outil dans lequel la tâche sera réalisée. Par exemple, l'action *zoom avant* pourrait être déclenchée dans un outil en appuyant sur la flèche en haut du clavier.

Sur un autre outil, il faut aller dans le menu, cliquer sur zoom avant (généralement dans le menu de second niveau), spécifier une valeur (par exemple, 150%), regarder si le *zoom-avant* nous convient, et sinon, répéter les étapes précédentes. Ces deux choix de mise en œuvre du zoom avant nécessitent clairement des efforts différents. Par conséquent, le coût de chaque interaction doit être spécifié dans la description de l'outil.

3.8.2.2 Avantager l’utilisation des interactions automatisées (C2.2)

En plus d'associer un coût à chaque interaction, un coût supplémentaire est additionné à l’effort si le mapping des primitives avec un bloc d’interaction nécessite une intervention humaine, tandis que l’outil offre ces interactions de manière automatisées. Cette contrainte permet de promouvoir le mapping avec des interactions déjà automatisées.

Par exemple, si nous avons besoin de sélectionner les classes qui ont plus de 50 méthodes, un scénario avec un filtre automatique est préféré à un autre scénario qui contient des interactions qui nécessitent que l’analyste vérifie les représentations graphiques, une par une, puis marque celles qui ont plus de 50 méthodes.

3.8.3 Les contraintes de mappings des métriques à des attributs graphiques (C3)

Dans cette catégorie, nous avons un seul type de contraintes. Ces contraintes concernent l'effort de percevoir les valeurs des métriques en fonction de l'attribut graphique avec lequel elles sont mappées. D'une part, les métriques sont définies en fonction d’une échelle de mesure (quantitative, ordinales et nominales). D'autre part, il existe un nombre fixe d’attributs graphiques pour visualiser les métriques, comme indiqué dans le tableau 5. L'association entre l'échelle d'une métrique et le type d’un attribut graphique détermine l'effort nécessaire pour percevoir la valeur de cette métrique au moyen de l'attribut graphique qui lui est associé.

Dans [21], Mackinlay et al. ont proposé le classement d'attributs graphiques, présentés dans le tableau 5, selon les échelles de mesure. Dans ces classements, la position est toujours considérée comme le meilleur choix. Toutefois, la longueur, qui est considérée comme le deuxième choix pour des mesures quantitatives, n’est que dans la huitième et la neuvième position respectivement pour les données ordinales et nominales.

Certains attributs, indiqués par "N", ne peuvent être utilisés pour certaines échelles, tels que la forme pour les données quantitatives et ordinales. Nous avons décidé d'utiliser ces classements pour définir les coûts des mappings des métriques aux attributs graphiques. Les mappings interdits, selon le tableau, seront sanctionnés avec un coût très élevé pour pénaliser les solutions qui les contiennent.

Tableau 5: Classification des indices visuels selon l'échelle d'expressivité [12].

	Quantitative	Ordonnée	Nominal
Position	1	1	1
Longueur	2	8	9
Angle	3	9	10
Pente	4	10	11
Surface	5	11	12
Volume	6	12	13
Densité	7	2	6
Saturation des couleurs	8	3	7
Teinte	9	4	2
Texture	N	5	3
Connexion	N	6	4
Inclusion	N	7	5
Forme	N	N	8

3.9 Fonctionnement de l'Algorithme

L'algorithme d'optimisation va avoir en entrée la tâche d'analyse listée dans un vecteur (Figure 10 à gauche). Ce vecteur va contenir la tâche d'analyse sous forme de code pour chaque but, primitive, élément et métrique. Par exemple, la tâche détection de Blob (Tableau 2) est décrite avec des codes dans la Figure 10 - gauche. Le vecteur de gauche contient les codes qui représentent la tâche). Chacun de ces codes, qui représente soit un but, une primitive, un élément ou une métrique va être mappé respectivement à un scénario, un bloc d'actions, une représentation graphique ou un attribut graphique. Les éléments mappés sont aussi codifiés pour faciliter le traitement et la compréhension.

Chaque solution (représenté par un vecteur) de la population initiale va avoir un mapping diffèrent de ces cellules (exemple Figure 10). L'algorithme va créer de nouvelle population en croisant les solutions. À chaque génération, l'algorithme va calculer l'effort en fonction des contraintes liées aux choix de mapping et déterminer ainsi la qualité de chaque solution. L'algorithme s'arrête quand la qualité des solutions ne s'améliore pas après plusieurs générations. La solution ayant la meilleure qualité (nécessitant le moins d'effort) sera proposée à l'analyste.

Cette solution se présente sous la forme d'un vecteur qui contient les codes du scénario, des blocs d'actions, des représentations graphiques et des attributs graphiques. Ces codes permettent de retrouver le nom du scénario, le bloc d'actions qu'il faut utiliser (par exemple le bloc d'actions à utiliser pour réaliser le filtre voulu, Tableau 4), le numéro de la représentation et de l'attribut graphique qu'on retrouve dans la description de l'outil de visualisation.

CHAPITRE 4 : ÉTUDE DE CAS

Cette étude de cas a pour but d'illustrer notre approche. Elle ne se veut pas une validation rigoureuse de cette dernière.

Nous allons reprendre la tâche d'analyse (Détection de Blob) détaillée dans le chapitre précédent, dans la description des tâches d'analyse. Nous avons défini le Blob et nous avons décrit cette tâche avec notre langage de description des tâches d'analyse. Notre but dans ce cas d'étude est de réaliser cette tâche d'analyse en utilisant l'outil de visualisation Verso que nous avons déjà présenté et modélisé dans le chapitre précédent, dans la description des outils de visualisation.

Dans ce qui suit, nous allons présenter le scénario de visualisation et les vues générées ainsi que le fonctionnement de notre algorithme de génération.

4.1 Génération des vues et des scénarios de visualisation

Nous avons exécuté notre algorithme avec en entrée la tâche du tableau 2 et avec un grand nombre de générations, ayant chacune une petite population. Nous présentons dans le tableau 8 le coût du meilleur scénario de visualisation dans les générations où des améliorations ont été détectées. Dans la première génération, le meilleur scénario de visualisation a un coût de 842 unités d'effort. La majorité des unités d'effort sont dues à des violations de contraintes C1.2, C1.4 et C2.2. En effet, certaines interactions considèrent que les attributs graphiques mappés sur les métriques sont perçus uniquement dans les vues locales alors qu'ils sont visibles dans la vue globale. En outre, certaines métriques sont considérées comme non mappées et *Détail–sur-demande (Detail_on_demand)* est utilisé pour accéder à leurs valeurs. Enfin, dans de nombreux cas, les interactions manuelles sont utilisées au lieu de celles automatisées.

Par la suite, nous avons obtenu à la génération 652, une solution avec 278 unités d'effort. Dans cette solution, le nombre de violations a été réduit de manière significative, mais certaines violations de la contrainte C1 ont été encore observées. En particulier, le scénario propose de modifier les mappings des métriques, tandis qu'il y a suffisamment d'attributs graphiques pour mapper toutes les métriques sans aucune modification.

Enfin, la meilleure solution a été trouvée dans la génération 1125. Le mapping des métriques ainsi que le scénario de visualisation sont représentées respectivement dans les tableaux 6 et 7. Cette solution a un coût de 18 unités d'effort. Dans cette solution, les seules contraintes qui influent sur le coût sont celles de type C2.1 et C3. Par exemple, le fait que la métrique WMC est mappée à la couleur génère un coût plus élevé que les mappings de DIT à la hauteur et LCOM5 à l'orientation.

Tableau 6 : Associations (mappings) des données pour le meilleur scénario de détection de Blob

Représentation graphique		
Boîte 3D	↔	Classe

Attributs graphiques		
Orientation	↔	LCOM5
Hauteur	↔	DIT
Couleur	↔	WMC

Tableau 7 : Le meilleur scénario obtenu pour la détection de Blob

```
Scenario(Blob_detection)
{
Run scenario(Detect_controller_class)
Run scenario(Is_data_class)
}
```

```
Scenario(Detect_controller_class)
{
Overview(Classes)
Block(Classes)
        {
        Check if(Color: Red
                and Twist: 0 to 45
                and Height: Medium to High)
        Select(Result)
        Tag(CC, Result)
        }
}
```

```
Scenario(Is_data_class)
{
For each(c in CC)
        {
        Overview(Classes)
        Apply automatic filter(Classes, isCalled(c) )
        Tag (Callers,Result)
        Overview(Classes)
        Bloc(REL)
                {
                Check if(Color: Bleu
```

```
                    and Height: Low)
                Select(Result)
                Tag(Data_class_callers, Result)
                }
        Overview(Data_class_callers)
        Do function(count, Data_class_callers, Num)
        Block
                {
                Check if(Num, High)
                Tag(Blob, c)
                }
        }
}
```

Le Tableau 7 décrit le scénario de visualisation (suite d'interactions que l'utilisateur doit effectuer dans l'outil de visualisation) qui permet de réaliser la tâche d'analyse voulue.

L'utilisateur doit d'abord créer les vues dans Verso en le configurant selon le mapping donné dans le Tableau 6. Une fois que l'on obtient la vue, on doit appliquer le scénario décrit dans le Tableau 7. Par exemple, pour réaliser le deuxième scénario on doit faire un Overview dans la vue pour voir toutes les classes qui sont représentées par des parallélépipèdes. Puis on va détecter les parallélépipèdes qui sont rouges, ont un angle de 0 à 45 degrés et une hauteur de moyenne à haute. Une fois les éléments détectés, il faut les sélectionner et les marquer (Tag).

Tableau 8 : Coût des meilleurs scénarios dans chaque génération

Génération	Coût
Initiale	842
2	687
9	683
11	576
33	428
92	326
652	278
865	277
994	175
1125	18

CHAPITRE 5 : CONCLUSION

Ce mémoire décrit une solution générique qui permet la génération d'un assistant pour un scénario de visualisation interactive permettant d'effectuer une tâche d'analyse de code. Notre solution prend en entrée une description de tâche et une spécification de l'outil de visualisation et produit un ensemble de vue et des scénarios d'interaction pour effectuer la tâche en entrée. Le processus de génération est implémenté comme un algorithme d'optimisation dont l'objectif est de trouver le scénario d'interaction qui nécessite le moins d'effort de l'analyste. Nous avons choisi d'utiliser un algorithme génétique avec une fonction objective qui pénalise les violations des contraintes de perception. Nous avons testé avec succès notre approche sur des tâches telles que la détection de défaut de conception. Nous avons aussi illustré notre approche avec le cas particulier de la détection de Blob.

Nous avons montré à travers du l'exemple du chapitre 4 que notre mécanisme de génération permet de converger, après un certain nombre d'itérations, vers un scénario d'analyse efficace. Notre étude a montré des résultats encourageants pour aider les analystes dans les tâches de la maintenance ou d'analyse de code. Cependant, certaines limites doivent être prises en considération à l'avenir. Tout d'abord, l'ensemble des contraintes doit être amélioré pour tenir compte d'autres aspects de visualisation tels que le choix d'emplacement des objets graphiques dans la vue utilisée. Ceci facilitera la perception et l'analyse des objets. En outre, l'estimation de l'effort pour chaque violation de contrainte devrait être améliorée. Dans notre solution, nous avons défini des intervalles de coûts. Cependant, nous pouvons mesurer l'effort de chaque interaction dans un outil spécifique à l'aide d'une étude empirique en utilisant des mécanismes tels que les systèmes de suivi du regard (Eye Tracking systems) (voir, par exemple, le travail de Jeanmart et al. [15], dans lequel ils ont étudié l'effort de comprendre un diagramme UML en présence d'un modèle de conception).

Pour démontrer l'efficacité de notre approche et montrer que les solutions proposées facilitent et améliorent le travail des analystes lors de l'utilisation des outils de visualisations,

nous nous proposons de faire une étude expérimentale. Cette dernière consiste à proposer à deux groupes d'analystes d'effectuer les mêmes tâches d'analyses de code en utilisant des outils de visualisation. Notre approche sera utilisée seulement par le premier groupe. La comparaison des résultats obtenus par les analystes, leurs performances et leur avis permettront d'évaluer l'efficacité de notre approche.

Comme nous l'indiquons dans notre article [30], en plus de la production de scénarios de visualisation, notre approche pourrait également être utilisée pour sélectionner des outils de visualisation. Dans ce cas, pour la même tâche d'analyse, nous ferons la génération de scénario de visualisation deux fois, chacune avec une spécification d'un outil. Puis, nous comparons les scénarios générés des outils respectifs et sélectionnons celui qui s'effectue avec le moins d'effort. La généralisation de notre solution pourrait être poussée plus loin en considérant la possibilité de construire des outils de visualisation qui sont les mieux adaptés pour une famille de tâches. Ceci sera possible si nous avons des bibliothèques de modules de visualisation qui peuvent être assemblés selon l'effort nécessaire pour effectuer les tâches ciblées.

Bibliographie

[1] E. Alikacem et Houari A. Sahraoui.*Rule-Based System for Flaw Specification and Detection in Object-Oriented Programs.* Dans Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2010)

[2] R. Amar, J. Eagan, and J. Stasko. *Low-level components of analytic activity in information visualization.* Information Visualization, IEEE Symposium on, 0:15, 2005.

[3] M. Agrawala and C. Stolte. *Rendering effective route maps: improving usability through generalization.* In Proceedings of the 28th annual conference on Computer graphics and interactive techniques, SIGGRAPH ’01, pages 241–249, New York, NY, USA, 2001. ACM.

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

[5] S. Chidamber and C. Kemerer. *A metrics suite for object oriented design.* Software Engineering, IEEE Transactions on, 20(6):476 –493, June 1994.

[6] M. D’Ambros et M. Lanza. *Reverse engineering with logical coupling.* Dans WCRE ’06: Proceedings of the 13th Working Conference on Reverse Engineering, pages 189–198, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2719-1.

[7] S. Demeyer, S. Ducasse et O. Nierstrasz. Object Oriented Reengineering Patterns. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1558606394. Foreword By-Ralph E. Johnson.

[8] S. Diehl, *Software Visualization:Visualizing the Structure, Behaviour, and Evolution of Software.*2007, XII

[9] D. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning.* Addison-Wesley, 1989.

[10] S. Hassaine, K. Dhambri, H. A. Sahraoui, and P. Poulin. *Generating visualization-based analysis scenarios from maintenance task descriptions.* In VISSOFT 2009, pages 41–44, Sept. 2009.

[11] C. Healey, G.Amant and J. Chang, 2001. *Assisted visualization of e-commerce auction agents.* In Graphics Interface, 201–208.

[12] C. Healey, S. Kocherlakota, V. Rao, R. Mehta, R. Amant,. *Visual Perception and Mixed-Initiative Interaction for Assisted Visualization Design.* Dans IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS. 2008, VOL 14; NUMB 2, pages 396-411.

[13] B. Henderson-Sellers. *Object-oriented metrics: measures of complexity.* Prentice-Hall, 1996.

[14] D. Holten, R. Vliegen et J. Van Wijk. *Visual realism for the visualization of software metrics.* Dans Stéphane Ducasse, Michele Lanza, Andrian Marcus, Jonathan I. Maletic et Margaret-Anne D, Storey, éditeur, VISSOFT, pages 27-32. IEEE Computer Society, 2005. ISBN 0-7803-9540-9.

[15] S. Jeanmart, Y.-G. Gueheneuc, H. Sahraoui, and N. Habra. *Impact of the visitor pattern on program comprehension and maintenance.* In Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on, pages 69 –78, 2009.

[16] R. Koschke. *Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey.* Journal of Software Maintenance, 15:87–109, March 2003.

[17] G. Langelier. *Visualisation de la qualité des logiciels de grandes tailles.* Mémoire de maîtrise, Département d'informatique et recherche opérationnelle, Université de Montréal, 2006.

[18] G. Langelier, H. Sahraoui, and P. Poulin. *Visualization-based analysis of quality for large-scale software systems.* In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ASE '05, pages 214–223, New York, NY, USA, 2005. ACM.

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

[20] G. Lommerse, F. Nossin, L. Voinea et A. Telea. *The visual code navigator: An interactive toolset for source code investigation.* Dans INFOVIS'05: Proceedings IEEE

Symposium on Information Visualization 2005, pages 25–32, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7803-9464-x.

[21] J. Mackinlay. *Automating the design of graphical presentations of relational information.* ACM Trans. Graph., 5:110–141, April 1986.

[22] J. Mackinlay, P. Hanrahan, and C. Stolte. *Show me: Automatic presentation for visual analysis.* IEEE Transactions on Visualization and Computer Graphics, 13:1137–1144, 2007.

[23] A. Marcus, L. Feng, and J.Maletic. *3D representations for software visualization system.* Dans SOFTVIS,2003.

[24] R. Marinescu. *Detection strategies: Metrics-based rules for detecting design flaws.* Dans Proceedings 20th IEEE International Conference on software Maintenance, pages 350-359, 2004.

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

[26] M. Moha, Y. Guéhéneuc, L. Duchien, A. Le Meur : *DECOR: A Method for the Specification and Detection of Code and Design Smells.* IEEE Trans. Software Eng. 36 (1) : 20-36 (2010)

[27] T. Munzner et P. Burchard.*Visualizing the structure of the world wide web in 3D hyperbolic Space.*Proceedings of the first symposium on Virtual reality modeling language, 1995.ISBN:0-89791-818-5

[28] S. Pfleeger, J. Atlee.*Chapitre 1: Software engineering: theory and practice (Fourth edition)* –Prentice Hall, 2009

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

[30] A. Sfayhi, H. Sahraoui (2011), "*What You See is What You Asked for: An Effort-Based Transformation of Code Analysis Tasks into Interactive Visualization Scenarios*", 11th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2011, 25-26 September 2011, Williamsburg, Virginia, USA, pp.195-203.

[31] B. Shneiderman. *The eyes have it: A task by data type taxonomy for information visualizations.* In IEEE Visual Languages, number UMCPCSD CS-TR-3665, pages 336 343, College Park, Maryland 20742, U.S.A., 1996.

[32] G. Stephan, E. Joseph, L.Steffen et E. Eric Summer Jr. *SeeSoft- a tool for visualizing line oriented software statistics.* IEEE Transactions on Software Engineering, 18(11):957-968, 1992. ISSN 0098-5589

[33] M. Termeer, Christian F. J. Lange, Alexandru Telea et Michel R. V. Chaudron.*Visual exploration of combined architectural and metric information.* Dans VISSOFT, pages 21–26, 2005.

[34] P. Tonella, M. Torchiano, B. Du Bois. *Empirical studies in reverse engineering: state of the art and future trends.* To appear in Journal on Empirical Software Engineering, 2007.

[35] O. Turetken and R. Sharda 2007. *Visualization of Web Spaces: State of the Art and Future Directions.* SIGMIS Database. 38, 3, 51-81.

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

[37] W. Wang, H. Wang, G. Dai, H. Wang. *Visualization of large hierarchical data by circle packing.* Dans Conference on Human Factors in Computing Systems, 2006. ISBN:1-59593-372-7

[38] S. Wehrend et C. Lewis. *A problem-oriented classification of visualization techniques.* Dans VIS'90: Proceedings of the 1st conference on Visualization'90, pages 139-143, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press. ISBN 0-8186-2083-8

[39] R. Wettel et M. Lanza. *Visualizing Software for Understanding and Analysis,* 2007. VISSOFT 2007. 4th IEEE International Workshop on, pages 92-99, 2007.