Détection visuelle d'anomalies de conception

Karim Dhambri, Salima Hassaine, Houari Sahraoui, Pierre Poulin

Dép. I.R.O., Université de Montréal
{dhambrik | hassaisa | sahraouh | poulin}@iro.umontreal.ca
http://www.iro.umontreal.ca

Résumé. De nos jours, les logiciels doivent être flexibles pour pouvoir accommoder d'éventuels changements. Les anomalies de conception introduites durant l'évolution du logiciel causent souvent des difficultés de maintenance. Cependant, la détection d'anomalies de conception n'est pas triviale. La détection manuelle est coûteuse en temps et en ressources, alors que la détection automatique génère trop de faux positifs. Dans cet article, nous proposons une approche semi-automatique pour la détection visuelle d'anomalies de conception. Pour aider la détection visuelle, nous utilisons un générateur qui transforme une tâche d'analyse décrite dans un langage de haut niveau en un scénario d'utilisation pour un environnement de visualisation. Finalement, nous illustrons notre approche avec un problème particulier de détection d'anomalies en utilisant la visualisation et les métriques de code.

1 Introduction

Les logiciels orientés objet (OO) sont en constante évolution. Dans plusieurs domaines, les programmes requièrent des changements fréquents, et ce, à tout niveau de considération. Un logiciel ne peut plus simplement répondre aux spécifications fonctionnelles ; il doit être flexible aux changements futurs. Les anomalies de conception introduites durant les processus de développement et de maintenance peuvent compromettre la facilité d'évolution d'un logiciel. Selon Fenton et Pfleeger [8], les anomalies de conception sont rarement la cause directe d'une défaillance, mais elles le sont souvent indirectement. La détection et la correction de ces anomalies est donc une contribution concrète à l'amélioration de la qualité du logiciel.

Cependant, la détection d'anomalies de conception n'est pas triviale [17]. La détection manuelle est coûteuse en temps et en ressources, alors que la détection automatique génère trop de faux positifs. Ces faux positifs sont dus à la nature des connaissances impliquées dans la détection [19]. De plus, il existe des difficultés additionnelles inhérentes à la détection d'anomalies, telles que la dépendance au contexte, la taille de l'espace de recherche, l'ambiguïté des définitions, ainsi que le problème bien connu de définition de valeurs seuil quand la détection utilise des aspects quantitatifs (métriques) [17].

Dans cet article, nous proposons une approche semi-automatique de détection d'anomalies de conception en utilisant la visualisation de logiciels. En utilisant notre outil de visualisation VERSO [14], quatre catégories d'information sont présentées à l'analyste : quantitative, architecturale, relationnelle et sémantique. Nous modélisons les anomalies de conception sous

forme de scénarios, dans lesquels les classes jouent des rôles primaires et secondaires. Ce modèle contribue à réduire l'espace de recherche pour la détection visuelle. De plus, nous présentons un générateur qui transforme une tâche de détection décrite dans un langage de haut niveau en un scénario d'utilisation pour un environnement de visualisation.

Notre approche est complémentaire à la détection automatique. En effet, nous nous concentrons précisément sur les anomalies qui sont difficiles à détecter de façon automatique. Nous utilisons des stratégies de détection qui combinent des actions automatisées, avec des actions exécutées par l'usager. Le jugement de l'utilisateur est sollicité lorsque les décisions automatiques sont difficiles à prendre. De telles décisions sont liées au contexte de l'application, au choix de l'architecture de bas niveau, à la variabilité des occurrences d'anomalies, et aux valeurs seuil des métriques.

Le reste de l'article est organisé comme suit. Un ensemble de travaux connexes est présenté dans la Section 2. La Section 3 introduit la détection visuelle ainsi que la façon dont nous représentons les différents types d'information avec notre outil de visualisation. Notre principe général de détection et deux exemples de stratégies de détection sont présentés dans la Section 4. La Section 5 décrit une méthode qui transforme une description de tâche de détection dans un langage de haut niveau en un scénario de détection qu'un analyste peut utiliser avec notre outil de visualisation VERSO. La Section 6 présente un exemple de transformation appliquée à l'anomalie de conception *blob*. Finalement, nous concluons en Section 7.

2 Travaux connexes

Bien qu'elles ne soient pas des défauts, *i.e.*, des violations de spécifications fonctionnelles, les anomalies de conception peuvent avoir un effet considérablement négatif sur plusieurs caractéristiques de qualité durant l'évolution du logiciel [3]. La détection d'anomalies de conception dans des logiciels à objet a été un sujet de grand intérêt au cours de la dernière décennie. Les *antipatterns* décrits par Brown *et al.* [3] ainsi que les *code smells* introduits par Fowler *et al.* [9] sont des exemples de recueils de description d'anomalies. D'autres exemples de violations d'heuristiques de conception OO ont été définis pas Riel [21] et Martin [18]. Une taxonomie de défauts de conception, proposée par Moha *et al.* [19], est utilisée pour générer automatiquement des algorithmes de détection. À propos du processus de détection, Ciupke [5] propose une technique d'analyse de code source, en spécifiant les problèmes de conception sous forme de requêtes, et de localisation d'occurrences de ces problèmes dans un modèle dérivé du code source. La majorité des anomalies détectées sont relativement simples, *i.e.*, de simples conditions avec valeurs seuil fixes telles que "la profondeur de l'arbre d'héritage ne doit pas excéder 6 niveaux". Les problèmes complexes, comme la détection d'*antipatterns*, ne sont pas adressés dans ce travail. Marinescu [17] propose une approche complète pour la détection d'*antipatterns* en utilisant des règles basées sur les métriques. Ratiu *et al.* [20] proposent d'utiliser l'historique de l'évolution d'occurrences suspectes du programme afin d'accroître l'efficacité de la détection automatique. Bien que plusieurs problèmes de détection soient traités dans ce travail, les taux de précision de la détection varient beaucoup d'une anomalie à l'autre, selon la nature de la connaissance requise.

D'autres approches utilisent la visualisation pour analyser des systèmes logiciels. Lanza et Ducasse [15] proposent une technique de visualisation de logiciels légère pour guider l'ingénieur logiciel durant les premières phases de rétro-ingénierie de logiciels de grande taille.

Dans [7], Ducasse et Lanza étendent le travail amorcé dans [15] en ajoutant la visualisation de la structure interne des classes. Plus récemment, Wettel et Lanza [22] ont présenté un système de visualisation 3D qui utilise la métaphore de la ville pour la compréhension de logiciels.

3 Détection visuelle

Notre approche de détection d'anomalies est basée sur des fonctionnalités fournies par notre système de visualisation de logiciel VERSO [14]. VERSO génère des représentations 3D efficaces de logiciels de grande taille. Les entités du logiciel, représentées par des éléments graphiques 3D, sont distribuées sur un plan 2D selon l'architecture de bas niveau. Les représentations utilisent des données quantitatives (métriques) et structurelles (relations) obtenues par des outils maison de rétro-ingénierie et d'extraction de métriques, PADL [10] et POM [11].

Nos stratégies de détection utilisent de l'information classifiée en quatre catégories : quantitative (*e.g.*, une classe très complexe, fortement couplée), relationnelle (*e.g.*, une classe utilise, invoque), architecturale (*e.g.*, une classe utilise des classes dans d'autres packages) et sémantique (*e.g.*, une classe joue le rôle d'un contrôleur, implémente une fonctionnalité unique). Le reste de cette section présente comment ces catégories d'information sont rendues disponibles à l'analyste en utilisant notre outil de visualisation. Nous verrons dans la Section 4 comment notre approche exploite ces quatre catégories d'information.

3.1 Information quantitative

L'information quantitative est capturée par les métriques extraites à partir du code. Pour une classe donnée, ses valeurs de métriques sont associées à des attributs graphiques qui seront visualisés par l'analyste. Les classes sont représentées par des boîtes 3D, et les métriques sont associées avec trois attributs de la boîte 3D : la hauteur, la couleur et l'orientation. Considérons par exemple une classe Java de 150 lignes de code (LOC = 150), avec deux ancêtres (DIT = 3), et couplée avec 5 classes (CBO = 5). Une telle classe peut être représentée par une boîte 3D de hauteur h correspondant au LOC de 150, une couleur c (dans un spectre qui varie de bleu à rouge) correspondant au CBO de 5, et une orientation t (variant de 0 à 90 degrés, où 0 degré correspond à l'axe nord-sud) correspondant au DIT de 3. Ces attributs de la boîte 3D ont été choisis car ils sont faciles à distinguer, et présentent peu d'interférence entre eux, même avec plusieurs milliers de classes affichées simultanément [14]. Les interfaces sont représentées par des cylindres pour les distinguer des classes.

Une fois que les métriques ont été calculées et associées aux attributs graphiques, un analyste peut évaluer visuellement si une condition de détection s'applique à une classe spécifique (par exemple si la classe est grande). Cette évaluation peut être faite de deux façons. Premièrement, pendant que l'analyste regarde toutes les boîtes (*i.e.*, les classes du programme analysé), il peut décider, en considérant le contexte global, si une classe a une grande valeur pour une métrique donnée, comparativement aux autres.

L'autre alternative est d'employer le filtre de distribution qui utilise la technique du *box plot* [8]. Ce filtre colore les classes selon leur position dans la distribution d'une métrique donnée. Par exemple, les classes qui ont une valeur de métrique anormalement grande, *i.e.*, très éloignée du quartile supérieur, prennent la couleur rouge, alors que les classes ayant des

valeurs situées entre les deux quartiles sont colorées en vert. La Figure 1 montre l'application du filtre de distribution pour la métrique de couplage CBO sur les classes de la librairie *Xerces*[1].

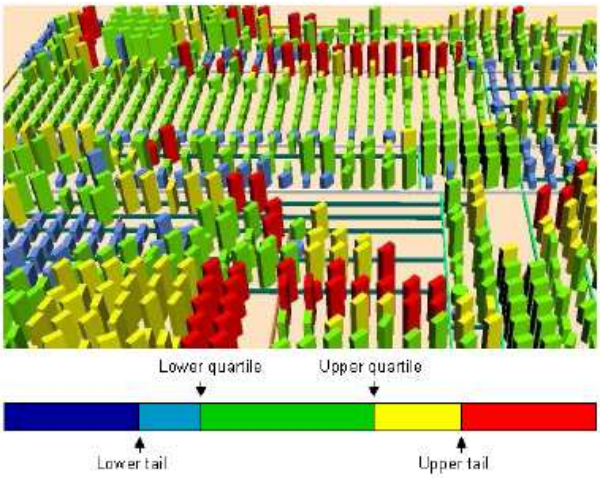

FIG. 1 – *Une vue de* Xerces *avec le filtre de distribution appliqué pour la métrique CBO. Note : les images de cet article doivent idéalement être vues en couleur. Vous pouvez les trouver à l'adresse www.iro.umontreal.ca/∼sahraouh/papers/LMO08_detection*

3.2 Information d'architecture de bas niveau

Ce type d'information fait référence à la structure du programme sous forme de modules/packages. Comme nous nous intéressons aux programmes Java ayant une structure de package en arbre, nous organisons visuellement les classes en utilisant l'algorithme de disposition *Treemap* [14]. L'algorithme du *treemap* représente graphiquement une arborescence à l'intérieur d'un rectangle. Les noeuds de l'arbre sont représentés par des subdivisions récursives du rectangle ; la direction des divisions alterne entre horizontal et vertical. La partie gauche de la Figure 2 montre un exemple de visualisation de programme.

L'algorithme de disposition considère le rectangle d'affichage (plan) comme la représentation du programme. Le rectangle est ensuite découpé en un nombre de tranches verticales, une pour chaque package principal. La taille de chaque tranche est calculée en fonction de la taille du package. À l'étape suivante, les nouveaux rectangles (tranches) sont découpés horizontalement pour diviser la surface d'affichage entre les sous-packages. Le processus de découpage continue de la même façon, en alternant les séparations verticales et horizontales, jusqu'à ce que l'on atteigne les packages feuilles. Les boîtes 3D représentant les classes qui sont contenues dans un package donné sont placées sur le rectangle correspondant.

Lorsque l'on observe la représentation d'un programme, il est facile de déterminer à quel package appartient une classe. Cette représentation peut informer l'analyste sur des problèmes de conception d'architecture, comme la prolifération de petits packages.

[1] http ://xerces.apache.org/

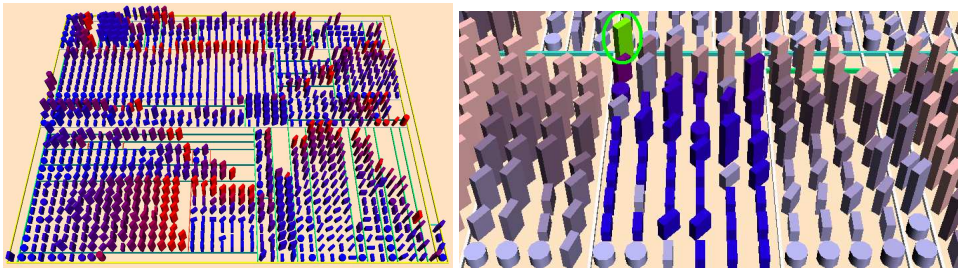

FIG. 2 – *(Gauche) Représentation de* Xalan *(1194 classes). (Droite) Un exemple de filtre de relation. Les classes ayant conservé leur couleur originale sont reliées à la classe verte par rapport à la relation sélectionnée.*

3.3 Information relationnelle

Les anomalies de conception peuvent être vues comme des scénarios joués par des classes inter-reliées. Plusieurs types de relations sont utilisées dans ces scénarios, en particulier les relations d'association et d'invocation.

Dans notre outil de visualisation, nous utilisons des relations obtenues par rétro-ingénierie (association, agrégation, généralisation, implémentation, invocation, etc.). Étant donné que nous travaillons avec des programmes de grande taille (des milliers de classes inter-reliées), un affichage explicite et statique des relations pourrait rapidement surcharger la perception de l'analyste. Pour contrer ce problème, nous avons développé un ensemble de filtres de structure, un pour chaque type de relation. Lorsqu'un filtre est appliqué pour une classe donnée, toutes les classes reliées conservent leur couleur originale, alors que la saturation de la couleur est réduite pour toutes les autres classes non-reliées, tel qu'illustré sur la partie droite de la Figure 2. La réduction de la saturation conserve l'information représentée par la couleur, mais diminue l'emphase sur les classes non-reliées. Ceci permet à l'analyste de concentrer son attention sur les classes reliées, tout en conservant une vue du contexte global du programme. Les filtres de structure peuvent aussi être combinés, permettant ainsi de se concentrer sur un sous-ensemble de classes vérifiant un certain nombre de relations.

3.4 Information sémantique

L'information sémantique fait référence à la connaissance du domaine qui n'est pas explicitement représentée dans le code source. Par exemple, dans le cas de l'anomalie décomposition fonctionnelle (Section 4), nous parlons de classes implémentant une fonctionnalité unique.

Lors de la détection avec notre outil de visualisation, si à un moment l'analyste doit déterminer si une classe joue un rôle donné, il peut déduire cette information de deux façons. Premièrement, il peut regarder le nom de la classe. Dans plusieurs cas, le nom est suffisamment significatif pour indiquer le rôle de la classe. Si ce n'est pas suffisant, l'analyste peut parcourir le code source de la classe pour une analyse plus approfondie. Bien que ce processus soit coûteux en temps, il est généralement fait après un premier filtrage, réduisant ainsi sa fréquence d'utilisation. En effet, lorsque nécessaire, l'information sémantique est souvent utilisée à la fin du processus de détection, pour accepter ou rejeter une classe candidate.

4 Stratégies de détection

Cette section décrit notre approche de détection visuelle semi-automatique. D'abord, nous expliquons notre principe général de détection, à partir duquel nous dérivons des stratégies de détection pour des anomalies de conception spécifiques. Ensuite, nous présentons des stratégies de détection pour deux anomalies, avec quelques exemples concrets de détection dans des logiciels libres.

4.1 Principe de détection

Les anomalies de conception que nous étudions dans ce travail prennent la forme soit d'une classe unique ayant certaines propriétés non-désirées, soit d'une micro-architecture reliant un groupe de classes. Dans ce dernier cas, une des classes joue le rôle principal, tandis que les autres jouent des rôles de support secondaires. Par exemple, pour l'anomalie du *blob* (Section 6) la classe contrôleur joue le rôle principal, et les *data classes* jouent des rôles secondaires. Pour toutes les anomalies de conception impliquant plusieurs classes, nous avons trouvé qu'il était plus efficace de rechercher les classes principales en premier, et de se concentrer sur les classes secondaires ensuite. Une explication pourrait être que les classes principales ont souvent des valeurs de métriques anormales, les rendant plus faciles à détecter avec notre outil de visualisation. De plus, le fait de voir les anomalies de micro-architecture comme des scénarios comprenant des rôles principaux et secondaires réduit considérablement l'espace de recherche pour l'analyste humain. Détecter des anomalies de conception comprenant m classes dans un programme de n classes peut mener à un grand nombre de combinaisons de classes à vérifier, dans le pire cas C_m^n. Cependant, lorsque l'on cherche des classes principales comme points d'entrée pour la recherche, le nombre de combinaisons de classes à vérifier est réduit à $n(m-1)$. En se basant sur cette observation, nous avons conçu un principe de détection général, à partir duquel nous dérivons des stratégies de détection pour des anomalies de conception spécifiques.

La première étape du principe de détection est de régler les associations entre les valeurs de métriques et les attributs graphiques, en utilisant des métriques pertinentes pour caractériser les différents rôles de l'anomalie que l'on veut détecter. Ensuite, l'analyste localise toutes les classes candidates pour le rôle principal de l'anomalie. Ceci peut être fait en regardant la représentation du programme et en cherchant des classes ayant une certaine apparence (selon le *mapping* de métriques). Le filtre de distribution peut aussi être utilisé pour cette deuxième étape pour visualiser la distribution de métriques et localiser les classes ayant des valeurs de métriques anormales. Toutes les classes candidates pour le rôle primaire sont marquées par l'outil de détection. Puis pour chaque classe marquée, l'analyste peut inspecter le nom et/ou le code source pour confirmer que la classe est réellement une occurrence du rôle principal. Si le rôle principal doit être supporté par des rôles secondaires (selon l'anomalie à détecter), une étape supplémentaire consiste à appliquer des filtres de relations. L'analyste considère l'apparence des classes reliées et décide s'il s'agit d'une occurrence de l'anomalie recherchée.

4.2 Exemples de détection d'anomalies

Le principe général défini plus haut peut être appliqué à un grand éventail d'anomalies. Dans le reste de cette section, nous présentons son application à deux anomalies : classe

mal placée et décomposition fonctionnelle. Chaque anomalie est accompagnée d'un exemple concret. Les stratégies de détection utilisent les métriques présentées dans le Tableau 1. Les définitions détaillées des quatre premières peuvent être trouvées dans [2, 4, 13]. Nous définissons RCU dans la sous-section suivante.

Nom	Définition
DIT	Niveau (profondeur) de la classe dans l'arbre d'héritage
LCOM5	Manque de cohésion dans les méthodes d'une classe
NPDM	Nombre de méthodes publiques déclarées d'une classe
WMC	Somme pondérée des complexités des méthodes d'une classe
RCU	Utilité relative d'une classe dans son package

TAB. 1 – *Métriques de détection.*

4.2.1 Classe mal placée

Stratégie de détection La classe mal placée va à l'encontre du principe de conception modulaire. Il s'agit d'une classe qui utilise et qui est utilisée par davantage de classes provenant d'autres packages que de son propre package. Si ces classes associées se trouvent principalement dans un même package, nous pourrions transférer la classe mal placée vers ce package [18]. RCU est la proportion de classes qui utilisent ou qui sont utilisées par la classe considérée, se trouvant dans des packages autres que le sien. RCU$= 0$ signifie que la classe interagit seulement avec des classes de son propre package, alors que RCU$= 1$ indique que la classe interagit uniquement avec des classes d'autres packages.

La détection s'effectue en trois étapes : (1) effectuer le *mapping*, (2) identifier les classes ayant une valeur de RCU proche de 1, une grande complexité et une faible cohésion, et (3) inspecter l'emplacement des classes reliées pour vérifier si elles se trouvent majoritairement dans un même package. La stratégie de détection pour la classe mal placée est résumée dans l'Algorithme 1.

Algorithm 1 Stratégie de détection de la classe mal placée

1: Régler le *mapping* métriques–attributs graphiques à :
   RCU-couleur, WMC-hauteur, LCOM5-orientation
2: Localiser les boîtes rouges (RCU≈ 1), grandes (complexes), et tournées (non cohésives)
3: **for** chaque classe candidate *c* **do**
4: Appliquer le filtre d'association "entrant/sortant" sur *c*
5: Inspecter les classes associées avec *c*
   (elles doivent être pour la plupart localisées dans un même package)
6: **if** toutes les étapes précédentes sont positives **then**
7: Sauvegarder l'occurrence de classe mal placée
8: **end if**
9: **end for**

Exemple Un exemple de classe mal placée est présenté dans la Figure 3. Dans la partie gauche, la classe verte encerclée a été sélectionnée comme candidate car elle est rouge vif (RCU près de 1), grande (grande valeur de WMC), et tournée (grande valeur de LCOM5). À droite, on peut voir le résultat du filtre d'association "entrant/sortant". La majorité des classes reliées (celles dont la couleur est restée saturée) sont concentrées dans un autre package, à droite de l'image.

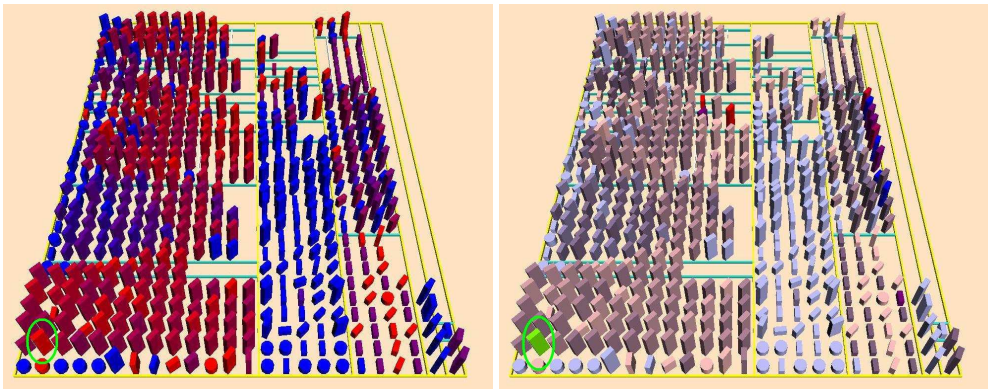

FIG. 3 – *Un exemple de classe mal placée découverte dans* Art of Illusion. *(Gauche)* Mapping *initial pour détecter la classe de rôle principal. (Droite) Filtre d'association sur la classe encerclée.*

4.2.2 Décomposition fonctionnelle

Stratégie de détection L'anti-patron décomposition fonctionnelle apparaît lorsqu'une classe est conçue avec l'intention d'accomplir une fonction unique dans le système. La classe possède donc une grande méthode qui est responsable de l'implémentation de cette fonction. La plupart des attributs de la classe sont privés, et utilisés principalement à l'intérieur de la classe. De telles classes portent souvent des noms qui dénotent une fonction (*e.g.*, *CalculerInterets* ou *AfficherTable*) [3]. Les conséquences d'une telle conception sont que les classes n'utilisent pas le mécanisme d'héritage (DIT très faible), et n'ont pratiquement qu'une seule méthode publique déclarée (NPDM$\approx$ 1). Cette méthode est très grande et complexe, ce qui influence la complexité de la classe (WMC élevé). La détection de ces symptômes est résumée dans l'Algorithme 2.

Exemple Des exemples de décomposition fonctionnelle détectés dans *ArgoUML* sont présentés dans la Figure 4. Les classes encerclées en vert ont été identifiées comme classes candidates. Elles sont droites (faible valeur de DIT), bleues (faible valeur de NPDM), et grandes (valeur de WMC élevée). L'inspection du code source a révélé que ces classes implémentent les fonctions d'initialisation (*checklist.Init* et *critics.Init*) ainsi qu'une fonction de génération (*critics.ChildGenUml*).

Algorithm 2 Stratégie de détection de la décomposition fonctionnelle

1: Régler le *mapping* métriques–attributs graphiques à :
NPDM-couleur, WMC-hauteur, DIT-orientation
2: Localiser les boîtes qui sont grandes (complexes), bleues (NPDM très faible), et presque droites (DIT très faible)
3: **for** chaque classe candidate c **do**
4: Inspecter le nom et le code source de c
(nom de classe indiquant une fonction, plusieurs attributs privés, etc.)
5: **if** toutes les étapes précédentes sont positives **then**
6: Sauvegarder l'occurrence de décomposition fonctionnelle
7: **end if**
8: **end for**

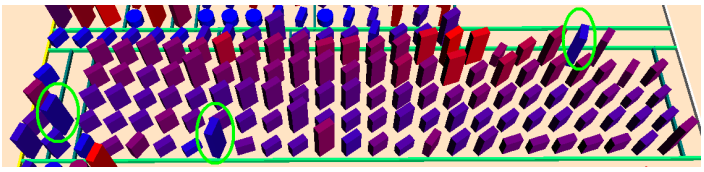

FIG. 4 – *Quelques exemples de décomposition fonctionnelle trouvés dans* ArgoUML.

5 Génération automatique de stratégies de détection

Lorsque l'utilisateur se trouve engagé dans un processus de traitement d'une grande quantité d'information, une représentation visuelle peut être un bon moyen de l'aider à réaliser cette tâche. Cependant, l'interprétation visuelle de l'information met en jeu des aspects cognitifs que l'utilisateur doit prendre en compte lors de l'utilisation de l'outil de visualisation. Malheureusement, les spécialistes en génie logiciel n'ont pas nécessairement une connaissance suffisante des règles de perception visuelle, et doivent donc souvent avoir recours à l'aide d'experts en visualisation pour les aider à utiliser les outils de manière plus efficace.

Face à la croissance continue de la quantité et des dimensions des données à manipuler, et à l'évolution des systèmes de visualisation, il est donc nécessaire d'assister les utilisateurs dans leur tâche d'analyse. Dans ce contexte, plusieurs approches sont envisageables : améliorer la qualité de la perception et de l'interactivité de l'espace d'information, permettre l'expression précise de tâches d'analyse en langage formel et déléguer certaines tâches d'analyse à un système d'aide. De toute évidence, toutes ces pistes sont valides et fonctionnent de manière complémentaire. Cette section présente le modèle de tâches d'analyse qui offre à l'utilisateur un langage lui permettant d'exprimer précisément une tâche d'analyse, pour les déléguer à un système d'aide à la visualisation qui génère la visualisation la plus adéquate à ses besoins.

5.1 Le modèle de tâches d'analyse

Le modèle de tâches d'analyse est utilisé pour décrire une tâche particulière, en termes d'objectifs et de procédures. Les procédures se décomposent récursivement en tâches et en sous-tâches, jusqu'à ce qu'on atteigne les opérateurs définis par une taxonomie. Ce modèle

permet d'identifier les types de connaissances que l'utilisateur doit avoir pour accomplir sa tâche. Les opérateurs sont des tâches élémentaires réalisées sur des ensembles de données, par exemple trouver des valeurs extrêmes.

Amar *et al.* [1] ont élaboré une taxonomie de ces tâches élémentaires. En se basant sur cette taxonomie, nous avons dérivé un ensemble d'opérateurs qui vont nous servir pour construire le langage de description de tâches ; chaque opérateur est caractérisé par les paramètres nécessaires à son exécution et la portée de son application. Un opérateur est donc décrit comme un tuple <opération, paramètres> où l'opération est une action à exécuter sur les données utilisant les paramètres spécifiés. Les paramètres incluent l'entité de code ou l'ensemble d'entités, attributs, conditions, étiquettes des textes, rapports, et propriétés telles que le nom de classe, la catégorie, la position de classe en ce qui concerne l'architecture de packages, etc. Chaque opérateur a une portée : global ou local. Un opérateur global est appliqué à un grand ensemble de classes et d'interfaces d'entités de code, alors qu'un opérateur local est appliqué sur un sous-ensemble réduit d'entités. Par exemple, en recherchant les classes qui ont les valeurs les plus élevées de couplage dans un programme P, l'opérateur global à employer serait *Find Extremum*⟨C, CBO⟩ avec les paramètres C, l'ensemble de classes dans P, et CBO (le couplage entre les objets), une métrique de couplage.

5.2 Génération de tâches

En plus de proposer des modes d'interaction pour aider l'utilisateur à manipuler les données, notre système d'aide à la visualisation doit pouvoir l'aider à trouver des modes d'expressions ou des attributs graphiques efficaces pour visualiser ses données. Par ailleurs, il est nécessaire de connaître les règles qui gouvernent la perception et la compréhension de cet attribut graphique. Il faut cependant noter que l'efficacité d'une représentation graphique est difficile à évaluer de façon absolue. Nous proposons de l'évaluer par rapport à un contexte particulier, en tenant compte du type de tâche et des caractéristiques des données. En effet, la modélisation de ce problème comme un problème de satisfaction de contraintes est naturelle, et sa résolution en utilisant la programmation par contraintes s'avère souvent efficace. Le cadre des problèmes de satisfaction de contraintes offre la possibilité de modéliser des problèmes dont les variables doivent prendre différentes valeurs de leur domaine, sous certaines contraintes.

Ainsi, pour le problème de recherche des attributs graphiques les plus adéquats à la visualisation des données, nous retrouvons les notions d'expressivité et d'efficacité d'un attribut graphique [16], ainsi que le phénomène d'interférence entre les attributs graphiques [12]. L'ensemble des variables est composé des données à visualiser, les domaines regroupent les attributs graphiques pouvant être associés aux données, et les contraintes portent sur les règles de perception qui permettent d'estimer l'efficacité d'un attribut graphique pour une certaine donnée.

6 Étude de cas

Dans cette section, nous présentons un exemple d'une tâche d'analyse de détection du *blob*, une anomalie bien connue dans la communauté du logiciel [3].

Le *blob* consiste en une classe qui monopolise une bonne partie du comportement d'un programme et qui interagit avec des classes qui servent principalement à encapsuler des don-

nées. Il est caractérisé par un diagramme de classes composé d'une classe contrôleur unique, complexe et non cohésive, associée à des classes de données simples. Dans la Section 6.1, nous voyons d'abord comment la tâche de détection du *blob* est spécifiée selon le modèle de tâches d'analyse. Ensuite, nous décrivons la transformation vers le scénario de détection du *blob* spécifique à VERSO dans la Section 6.2. Finalement, nous présentons comment ce scénario a été appliqué à un système réel pour détecter une occurrence de *blob*.

6.1 Description de la tâche d'analyse

Dans un premier temps, la détection du *blob* est décrite par une séquence d'actions à accomplir selon le modèle de tâches d'analyse (voir la Section 5.1). La spécification[2] de la tâche de détection du *blob* est présentée dans le Tableau 2. Cette description utilise des métriques logicielles et des techniques générales d'analyse de données.

Method to accomplish Goal : Blob Detection
1 : Find Extremum(whole system, WMC)
2 : FOR EACH (c, selected classes)
3 : IF NOT(Verify Value(c, $LCOM5$, $HIGH$)) CONTINUE
4 : IF NOT(Verify Value(c, DIT, LOW)) CONTINUE
5 : Inspect(c, class name, method properties)
6 : Decide(the inspection confirmed that c is a controller class, Save(c, controller class))
7 : END FOR EACH

Method to accomplish Goal : Data Class Verification
1 : FOR EACH (m, controller classes)
2 : Relationship Filter(m, association out)
3 : Locate(associated classes, $WMC = LOW$, $LCOM5 = LOW$, $DIT = LOW$)
4 : Decide(located classes are too numerous, Save(m, Blob))
5 : END FOR EACH

TAB. 2 – *Détection du* blob *exprimée par le modèle de tâches d'analyse.*

6.2 Génération des tâches interactives

Une fois que la détection du *blob* a été spécifiée en utilisant le modèle de tâches d'analyse, un scénario spécifique de VERSO est généré automatiquement. Puisque chaque attribut impliqué dans la tâche d'analyse est employé par au moins un opérateur global, les attributs visuels disponibles sont la couleur, la hauteur et l'orientation.

Le scénario généré est présenté dans le Tableau 3. Bien que le scénario de VERSO contienne plus d'instructions que la description de départ (voir le Tableau 2), nous avons gardé la même

[2]L'anglais est utilisé dans le tableau 2 pour être fidèle à la spécification du langage de tâches d'analyse

numérotation pour montrer de quelle façon les opérateurs de la description ont été transformés en séquences d'actions dans le scénario.

Mapping d'attributs
DIT ↔ couleur
WMC ↔ hauteur
LCOM5 ↔ orientation

Méthode pour accomplir le but : détection du blob
1 : Appliquer le `zoom-out` si nécessaire
Appliquer le `filtre statistique` sur WMC pour tout le système
En utilisant l'`itérateur`, `sélectionner` les classes tant que WMC est jugé assez grand
Enlever le `filtre statistique`
2 : POUR CHAQUE c parmi les classes sélectionnées
3 : SI(vérifier que l'orientation n'est pas GRANDE pour c) CONTINUER
4 : SI(vérifier que la couleur n'est pas BLEU pour c) CONTINUER
5 : Appliquer le `zoom-in` si nécessaire
`Afficher le code source` de c
Inspecter le nom de la classe
Inspecter les signatures de méthodes
6 : Si l'inspection confirme que c est une classe contrôleur,
`Marquer` c comme classe contrôleur.
7 : FIN POUR CHAQUE

Méthode pour accomplir le but : vérification de classe de données
1 : POUR CHAQUE m parmi les classes contrôleur
2 : Appliquer le `filtre associations-sortantes` sur m
3 : Parmi les classes associées, localiser les classes pour lesquelles hauteur est BASSE, orientation est BASSE, couleur est BLEU
4 : Si les classes localisées sont nombreuses, `marquer` m comme blob
5 : FIN POUR CHAQUE

TAB. 3 – *Scénario de détection du* blob *spécifique à VERSO.*

6.3 Application du scénario de détection du *blob*

Cette section présente comment le scénario de détection du *blob* généré dans la Section 6.2 (voir le Tableau 3) peut être employé par un analyste en utilisant VERSO.

La première instruction à l'étape 1 est d'appliquer *Zoom-out* au besoin. Il s'agit de s'assurer que l'utilisateur a un point de vue global du système avant d'appliquer le filtre statistique. Une vue d'ensemble de *PCGEN* est illustrée sur la partie gauche de la Figure 5. Le filtre statistique est ensuite appliqué sur le système entier pour la métrique WMC. Les classes qui nous intéressent, *i.e.*, ayant des valeurs extrêmement élevées pour cette métrique, sont colorées en rouge. Un itérateur est employé pour visiter et sélectionner les classes contrôleurs candidates en colorant en noir la prochaine classe à explorer. Une étape de ce processus est illustrée sur la partie droite de la Figure 5, où la classe encerclée en noir a été choisie comme candidate de classe contrôleur.

Une fois que toutes les classes candidates pour le rôle de contrôleur ont été choisies, le filtre statistique est enlevé, et les classes reprennent leur couleur originale (variant de bleu à rouge selon la valeur de DIT). L'analyste vérifie ensuite l'orientation et la couleur de chaque classe sélectionnée. Si l'orientation n'est pas élevée ou la couleur n'est pas bleue, l'analyste passe à la classe suivante (la profondeur et la cohésion sont trop élevées pour que cette classe soit considérée comme une classe contrôleur). Si une classe sélectionnée est bleue et tournée, l'analyste peut inspecter son code source pour vérifier le nom de la classe et les signatures de méthodes. Si l'inspection confirme que la classe sélectionnée est une classe contrôleur, elle doit être marquée en conséquence.

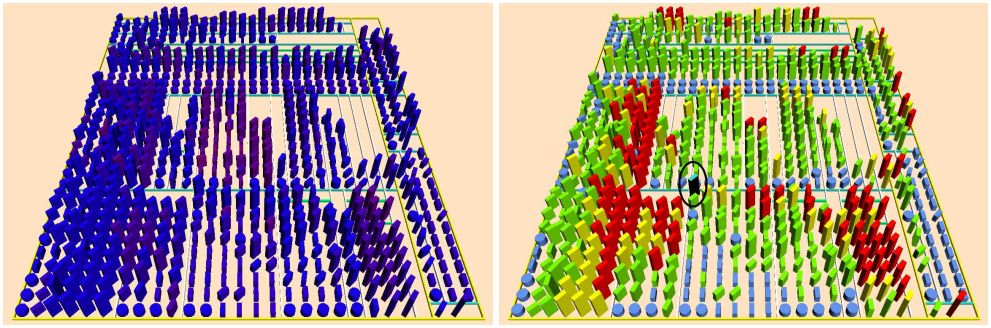

FIG. 5 – *Un exemple de* blob *découvert dans* PCGEN. *(Gauche) Opération Zoom-out appliquée. (Droite) Le filtre statistique est appliqué pour la métrique WMC. La classe encerclée en noir a été sélectionnée comme classe contrôleur.*

Pour chaque classe contrôleur, l'analyste doit maintenant considérer les classes associées. Le filtre d'associations sortantes est ainsi appliqué pour chaque classe contrôleur. Cette opération est illustrée sur la partie gauche de la Figure 6. Les classes associées à la classe contrôleur gardent leur couleur originale, alors que les autres classes subissent une baisse de saturation. Une fois que le filtre est appliqué, l'analyste doit localiser, parmi les classes associées, les classes qui sont petites, droites et bleues. En conclusion, si ces classes sont nombreuses, la classe contrôleur est marquée comme étant une occurrence de *blob*. La partie droite de la Figure 6 montre qu'un bon nombre de classes associées sont petites, droites et bleues. Ainsi, la classe contrôleur a été marquée comme *blob*.

7 Conclusion

La détection et la correction d'anomalies sont une façon concrète et efficace d'améliorer la qualité du logiciel. Nous proposons une approche semi-automatique de détection qui combine le pré-traitement automatique et la représentation visuelle des données de manière à exploiter les capacités du système visuel humain. Notre approche est complémentaire aux approches automatiques pour les anomalies dont la détection exige une connaissance qui ne peut être extraite directement à partir du code source. Plus précisément, nous détectons des occurrences d'anomalies en les modélisant en tant qu'ensembles de classes jouant des rôles dans des scénarios pré-définis. Des rôles primaires sont identifiés en premier lieu, puis, à partir de ces rôles,

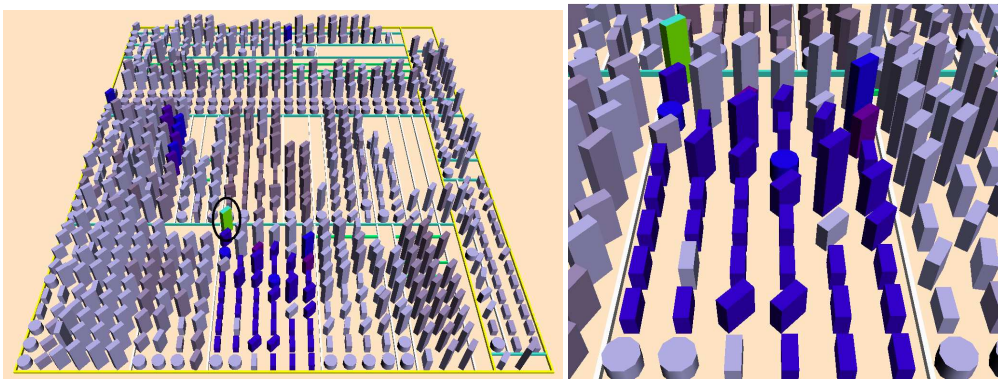

FIG. 6 – *Un exemple de* blob *découvert dans* PCGEN.

des rôles secondaires sont localisés. Cette stratégie réduit l'espace de recherche, ce qui permet d'intégrer l'intervention humaine. Nous avons appliqué avec succès notre approche sur des systèmes comprenant plus de 1000 classes.

Comme travaux futurs, nous prévoyons combiner notre approche avec d'autres approches automatiques. Trois combinaisons sont possibles. D'abord, nous pouvons utiliser notre outil pour explorer les résultats de la détection automatique et accepter/rejeter les occurrences détectées. Nous pouvons également employer les deux approches en parallèle selon les connaissances requises pour détecter les anomalies. Finalement, nous pouvons employer la technique de *focus stylisé* [6] pour attirer l'attention de l'analyste vers les parties du programme où se trouvent les occurrences candidates détectées automatiquement.

Références

[1] Robert A. Amar, James Eagan, and John T. Stasko. Low-level components of analytic activity in information visualization. In *INFOVIS*, page 15, 2005.

[2] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Engineering*, 25(1) :91–121, January/February 1999.

[3] William J. Brown, Raphael C. Malveau, III Hays W. McCormick, and Thomas J. Mowbray. *AntiPatterns : Refactoring Software, Architectures, and Projects in Crisis*. John Wiley Press, 1998.

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

[5] Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proc. Technology of Object-Oriented Languages and Systems*, pages 18–32, 1999.

[6] Forrester Cole, Doug DeCarlo, Adam Finkelstein, Kenrick Kin, Keith Morley, and Anthony Santella. Directing gaze in 3D models with stylized focus. *Eurographics Symposium on Rendering*, pages 377–387, June 2006.

[7] Stéphane Ducasse and Michele Lanza. The class blueprint : Visually supporting the understanding of classes. *IEEE Trans. Software Engineering*, pages 75–90, 2005.

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

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

[10] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships : Putting icing on the UML cake. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–314, 2004.

[11] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In *Proc. Working Conference on Reverse Engineering*, pages 172–181, November 2004.

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

[13] Brian Henderson-Sellers. *Object-Oriented Metrics, Measures of Complexity, The Object Oriented Series*. Prentice Hall, 1996.

[14] Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proc. IEEE/ACM International Conference on Automated Software Engineering*, pages 214–223, 2005.

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

[16] Jock Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5(2) :110–141, April 1986.

[17] Radu Marinescu. Detection strategies : Metrics-based rules for detecting design flaws. In *Proc. IEEE International Conference on Software Maintenance*, pages 350–359, 2004.

[18] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.

[19] Naouel Moha, Yann-Gaël Guéhéneuc, and Pierre Leduc. Automatic generation of detection algorithms for design defects. In *Proc. IEEE/ACM International Conference on Automated Software Engineering*, pages 297–300, September 2006.

[20] Daniel Ratiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *Proc. Conference on Software Maintenance and Reengineering*, 2004.

[21] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[22] Richard Wettel and Michele Lanza. Program comprehension through software habitability. In *15th International Conference on Program Comprehension*, 2007.

Détection visuelle d'anomalies de conception

Summary

Nowadays, software must be flexible to accommodate future changes. Design anomalies, introduced during software evolution, are frequent causes of low maintainability and low flexibility to future changes. However, detecting design anomalies is far from being trivial. Manual detection is time and resource consuming, while automatic detection yields too many false positives. In this paper we propose a visualization-based approach to semi-automatic detection of design anomalies. To assist visual detection, we use a generator that transforms analysis tasks described in a high-level language into a scenario for our visualization framework. Finally, we illustrate our approach on the particular problem of anomaly detection using visualization and code metrics.