This paper has been accepted at the JNPC 2001.

Quelques explications pour les patrons

Une application de la PPC avec explications pour l'identification de patrons de conception

Yann-Gaël Guéhéneuc*
Narendra Jussien
École des Mines de Nantes
4, rue Alfred Kastler – BP 20722
F-44307 Nantes Cedex 3
{Yann-Gael.Gueheneuc|Narendra.Jussien}@emn.fr

Résumé

Les patrons de conception décrivent des micro-architectures qui résolvent des problèmes architecturaux récurrents dans les langages orientés-objets. Il est important d'identifier ces micro-architectures lors de la maintenance des programmes orientés objets. Mais ces micro-architectures apparaissent souvent sous des formes dégradées dans le code source. Nous présentons une application de la programmation par contraintes avec explications pour l'identification de ces micro-architectures dégradées.

1 Introduction

La production de code source de qualité est un enjeu important pour l'industrie du logiciel. Un code source de qualité facilite l'évolution et la maintenance : ajout de nouvelles fonctionnalités, correction de bogues, adaptation à de nouvelles plates-formes d'exécution, intégration dans des bibliothèques de classes, ... En programmation par objets, un code source de qualité se distingue par deux aspects : des algorithmes efficaces et énoncés le plus clairement possible, respectant des conventions et des idiomes d'écriture, et une architecture de classes *élégante*. Si le premier aspect a été étudié dans de nombreux travaux ([DDN00] en présente une synthèse), le second aspect, l'architecture des classes, est plus difficile à définir et ne fait l'objet que de peu d'études (on peut citer [JZ97]).

Une micro-architecture décrit la structure d'un sous-ensemble des classes[1] d'un programme orienté objets. Les *patrons de conception* (*design patterns*) [GHJV94] sont des

*Ces travaux sont en partie financés par Object Technology International Inc. – 2670 Queensview Drive – Ottawa, Ontario, K2B 8K1 – Canada.

[1] Pour plus de clarté, dans le reste de cet article, nous utilisons le langage de programmation orienté objets à héritage simple JAVA et le terme *classe* représente indifféremment une classe, une classe abstraite ou une interface.

exemples de **bonnes** micro-architectures. Néanmoins, il n'est pas aisé d'écrire directement du code source qui respecte scrupuleusement des patrons de conception. C'est pourquoi les patrons de conception sont souvent présents dans le code source sous des formes dégradées (*i.e.*, des micro-architectures proches mais non identiques à celles proposées par les patrons de conception). Il n'existe pratiquement pas d'outils permettant de retrouver dans du code source déjà écrit ces versions dégradées de patrons de conception pour en proposer une amélioration à l'utilisateur – pour les rendre conformes aux bonnes micro-architectures formulées par les patrons de conception.

Dans cet article, nous présentons l'utilisation de la programmation par contraintes avec explications [JB00] pour construire un assistant logiciel permettant d'identifier et de corriger des architectures proches de patrons de conception. Après avoir introduit plus précisément la notion de patron de conception (section 2), rappelé les solutions proposées dans la littérature à l'identification et à la correction automatique du code source des programmes (section 3), et précisé les notions clés sur les explications dans le cadre de la programmation par contraintes (section 4), nous présentons la solution que nous avons développée à l'identification de micro-architectures proches de patrons de conception (section 5) avant de commenter les premiers résultats obtenus sur des bibliothèques industrielles (section 6).

2 Patrons de conception

L'architecture d'un logiciel est propre à celui-ci. Elle dépend du contexte dans lequel il est développé et de divers aspects : durée de vie attendue, coût de développement, évolutions prévues, expérience des concepteurs et développeurs, ... Pour un problème particulier, il n'existe pas **une** meilleure architecture mais une architecture adaptée à un contexte donné. Aussi, nous allons nous intéresser aux problèmes généraux d'architecture, bien définis et permettant de s'abstraire du contexte.

2.1 Définition

Pour des problèmes récurrents d'architecture, les patrons de conceptions (*design patterns*) [GHJV94] représentent des solutions indépendantes du contexte et du langage à objets utilisé. Ils fournissent un moyen de capturer l'expérience de développeurs expérimentés et de la restituer sous la forme d'un catalogue. Un patron de conception (*cf.* exemple 1 et section 6) contient quatre rubriques essentielles :

1. un nom identifiant le patron de manière unique ;
2. une description du problème architectural qu'il entend résoudre ;
3. une solution au problème, accompagnée de diagrammes de classes et d'instances représentant les classes participantes et leurs rôles (décrit à l'aide d'une notation proche de l'OMT – Object Modelling Technique [RBP+91]) ;
4. les conséquences de l'application de cette solution et les compromis possibles.

Par exemple, le patron de conception Composite ([GHJV94], pp. 163–173) comporte onze sections réparties sur dix pages (dix pages est la longueur moyenne d'un patron de conception dans cet ouvrage). Les trois premières sections, *Intent*, *Motivation* et *Applicability*, présentent le problème (rubrique 2) : composer des classes d'objets sous la

forme d'un arbre pour traiter uniformément un objet ou une composition d'objets. Les trois sections suivantes, *Structure*, *Participants* et *Collaborations*, proposent une solution (rubrique 3) à ce problème sous la forme de diagrammes de classes (*cf.* exemple 1) et d'instances, d'une liste des classes participantes – liste présentant leurs rôles –, et d'une liste des collaborations entre les classes – liste présentant leurs interactions. La section *Consequences* (rubrique 4) énonce les effets de l'application de ce patron de conception : simplification des clients et de l'implémentation. Enfin, les sections *Implementation*, *Sample Code*, *Known Uses* et *Related Patterns* fournissent des renseignements spécifiques à l'implémentation et à l'utilisation de la solution.

Les sections décrivant le problème résolu par un patron de conception sont toujours informelle. Elles présentent moins une description précise du problème qu'un ensemble de motivations pour lesquelles appliquer ce patron. C'est pourquoi la description du problème résolu ne peut être utilisée directement pour détecter des problèmes architecturaux. En revanche, l'architecture solution préconisée, présentée à l'aide de diagrammes de classes et d'exemples de code, peut être considérée comme un exemple de **bonne** micro-architecture.

Exemple 1 (Un survol du patron de conception Composite) :
Le patron de conception Composite construit des structures complexes d'objets en composant récursivement sous forme d'arbres des objets de même nature. Il permet de traiter uniformément les objets et les compositions d'objets. La structure générale du patron de conception Composite est présentée Figure 1.
La classe abstraite (ou interface) `Component` définit une `operation()` qui peut être appliquée indifféremment sur un objet de type `Leaf` ou sur une composition d'objets de type `Component`, `Composite`. Les instances de la classe `Composite` se chargent alors d'appliquer l'`operation()` sur l'ensemble de tous leurs fils.

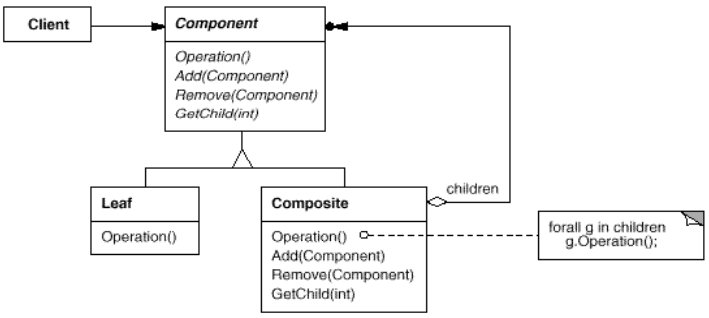

FIG. 1 – Le patron Composite (illustration tirée de [GHJV94])

2.2 Vers un code de meilleure qualité

Les évolutions et transformations auxquelles l'architecture d'un logiciel est soumise au cours de son cycle de vie freinent la mise en œuvre des patrons de conception ou en détériorent la qualité. Il est souvent difficile d'appliquer *a priori* ces règles de bonne architecture alors que le logiciel n'est pas finalisé. Cela nécessite une connaissance exhaustive des patrons de conception existants – culture que peu de développeurs ont –, et une vision précise de ce à quoi l'architecture de l'application devra ressembler. C'est pourquoi nous ne nous préoccupons pas ici d'aider à produire **directement** du code de bonne qualité mais cherchons plutôt à corriger le code produit pour suggérer de manière incrémentale la mise en œuvre de ces patrons.

Dans ce cadre, nous voulons identifier les emplacements dans l'architecture dont la qualité serait augmentée par l'application d'un patron de conception. Notre approche consiste à détecter les groupes de classes dont la structure est proche d'une micro-architecture solution préconisée par un patron de conception (*cf.* exemple 2). La structure des classes d'un tel groupe pourra alors être améliorée en appliquant la solution proposée par le patron de conception. Il s'agit donc de : **(a)** décrire les relations entre les classes introduites par les patrons de conception ; **(b)** détecter dans du code source les classes organisées selon une architecture proche d'un patron connu (déjà référencé) ; **(c)** appliquer sur le code source les transformations nécessaires.

Nous nous intéressons, dans cet article uniquement à la phase **(b)**, la phase d'identification.

> **Exemple 2 (Recherche du patron Composite) :**
> Considérons le développement d'une application de description de documents. Le noyau de l'application, présenté Figure 2 (à gauche) se compose d'une classe `Element`, définissant l'interface commune à tous les constituants d'un document : titres (classe `Title`), paragraphes (classe `Paragraph`), paragraphes indentés (classe `ParaIndent`), ... Une classe `Document` compose ces éléments pour représenter un document.
> Cette architecture est très proche du patron de conception Composite. D'après les spécifications de ce patron, la classe `Document` devrait être sous-classe de la classe `Element` afin d'unifier leurs interfaces respectives et de pouvoir composer des documents (*cf.* Figure 2, à droite).

3 État de l'art

Dans le domaine du génie logiciel et de la rétro-conception, il existe encore peu de travaux sur la détection et la correction automatique des défauts de conception. Il y a deux raisons à cela. D'une part, l'automatisation de processus et de techniques liés à une création *intellectuelle*, telle qu'un logiciel, est mal accueillie : la perte apparente de contrôle, dûe à l'automatisation d'un processus (la détection et correction des défauts de conception) intervenant sur des logiciels déjà difficilement maintenables, est souvent perçue comme trop importante par rapport aux bénéfices apportés par l'automatisation. D'autre part, quand des solutions automatiques ont été proposées, elles se réduisaient soit aux problèmes de détection ou correction (semi-)automatique des défauts de conception

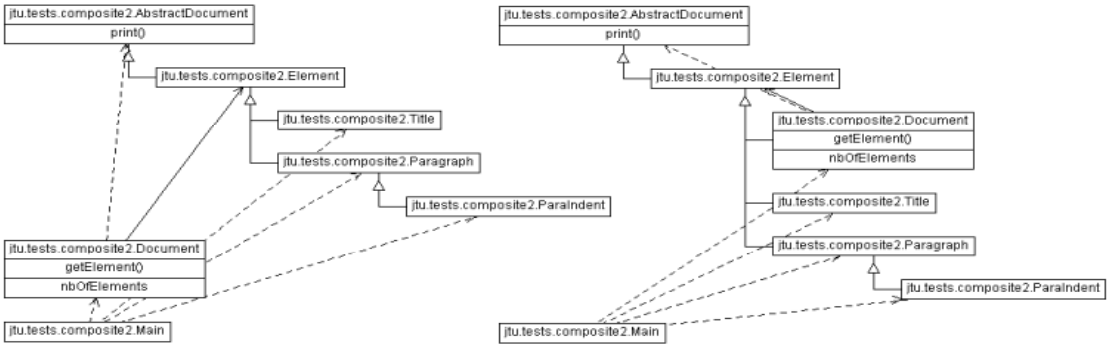

FIG. 2 – Noyau de l'application de descriptions de documents : **à gauche** la version originale et **à droite** la version corrigée

des classes elles-mêmes [JZ97, Fow99, DDN00] (par exemple, problèmes de corps de méthodes trop longues ou de manque de cohésion entre les méthodes d'une classe) ; soit aux problèmes de détection des patrons de conception existant dans le code pour aider à la rétro-documentation ou à la compréhension [Bro96, KP96, Wuy98, MMCG98, RD99].

La première catégorie de problèmes, la conception des classes elles-même, est intéressante car elle donne un aperçu des résultats à attendre d'approches automatiques. Les résultats présentés montrent qu'une approche complètement automatique est trop ambitieuse en raison de la complexité des logiciels.

La seconde catégorie de problèmes offre un aperçu des techniques utilisées pour la détection de patrons de conception. Parmi ces techniques, une qui semble offrir beaucoup d'intérêt est l'utilisation de la programmation logique [Wuy98] : un patron de conception est décrit sous forme de règles logiques qui sont unifiées avec une base de faits représentant le code source du logiciel. Mais cette technique est limitée car elle ne détecte que les classes dont les relations sont décrites par les règles logiques. Elle ne permet pas de détecter directement des architectures similaires mais non identiques à un patron de conception donné. Pour obtenir plus de solutions, il convient d'étendre la base de règles pour introduire les cas **dégradés** manquants. Ainsi, la base de règles devient rapidement inexploitable. La prise en compte d'un nouveau scénario devient problématique : il faut en effet tenir compte dès la conception du système de règles logiques de toutes les possibilités de dégradation existantes.

En dehors de la communauté *langage à objets*, on peut penser à la recherche de sous-graphes dans un graphe [Rég95]. Néanmoins, la recherche de sous-graphes *similaires* à un sous-graphe donné n'a pas été, à notre connaissance, encore abordée. On peut aussi penser à la phase d'*adaptation* du raisonnement à partir de cas. [FLMN00] a récemment présenté une technique bien adaptée aux domaines continus (avec une relation d'ordre sur les valeurs) mais inadaptée à un problème discret comme le nôtre.

En conclusion, une solution acceptable pour résoudre notre problème doit permettre de favoriser un dialogue avec le concepteur du code pour :

- **expliquer** concrètement pourquoi l'architecture d'un groupe de classes dans son programme est une version dégradée d'un patron de conception existant ;
- et diriger dynamiquement la recherche de telles architectures en proposant **interactivement** (en évitant ainsi d'avoir à déterminer *a priori* les possibilités d'évolution)

la prise en compte ou non de telle ou telle caractéristique du patron recherché.

Ces spécificités font de ce problème un candidat à l'utilisation de la PPC avec explications.

4 Un outil privilégié : PPC avec explications

Comme on vient de le voir, une possibilité pour aider à l'amélioration du code source de programmes orientés objets est de suggérer, en les expliquant, des modifications architecturales. L'utilisation de la programmation par contraintes avec explications a déjà montré son intérêt [JB00] dans plusieurs domaines. Nous rappelons ici de quoi il s'agit et quels peuvent être les intérêts de l'utilisation d'une telle technique y compris pour notre problème.

4.1 La notion d'explication

Dans la suite, nous considérons un CSP (V, D, C). Les décisions prises lors de l'énumération (affectations) sont considérées comme des contraintes ajoutées ou retirées (par exemple, lors d'un *backtrack*) au système courant de contraintes.

Une **explication de contradiction** (également appelée *nogood* [SV94]) est un sous-ensemble du système courant de contraintes qui, pris seul, conduit à une contradiction (aucune solution ne contient un *nogood*). Une explication de contradiction peut être divisée en deux parties : un sous-ensemble de l'ensemble de contraintes originel ($C' \subset C$ dans l'équation 1) et un sous-ensemble des décisions (rappelons-le, intégrées sous forme de contraintes au système) prises jusqu'à présent.

$$C \vdash \neg (C' \wedge v_1 = a_1 \wedge ... \wedge v_k = a_k) \tag{1}$$

Dans une explication de contradiction contenant au moins une contrainte de décision, une variable v_j peut être sélectionnée et la formule précédente réécrite ainsi[2] :

$$C \vdash C' \wedge \bigwedge_{i \in [1..k] \setminus j} (v_i = a_i) \rightarrow v_j \neq a_j$$

La partie gauche de l'implication constitue ce qu'on appelle une **explication de retrait** pour la valeur a_j du domaine de la variable v_j. Elle est notée $\texttt{expl}(v_j \neq a_j)$.

Les solveurs classiques sur les CSP procédant par réduction de domaine (retirant donc des valeurs), la conservation des explications de retrait suffit à la détermination des explications de contradiction. En effet, la contradiction est identifiée par la suppression de toutes les valeurs dans le domaine d'une variable v_j et un *nogood* peut être calculé simplement à partir des explications de retraits des valeurs supprimées :

$$C \vdash \neg \left(\bigwedge_{a \in d(v_j)} \texttt{expl}(v_j \neq a) \right)$$

[2]Une explication de contradiction sans une telle contrainte est caractéristique d'un problème surcontraint.

Il existe généralement plusieurs explications différentes pour un retrait donné. On pourrait vouloir les conserver toutes mais ceci conduirait bien évidemment à une complexité spatiale exponentielle. Une approche plus pragmatique consiste à **oublier** une partie des *nogoods* : ceux qui ne sont plus pertinents[3] dans l'état courant de la recherche. De cette façon, la complexité spatiale de l'enregistrement reste polynômiale. Ici, seule **une** explication de retrait est conservée à la fois.

4.2 Mise en œuvre

Les explications les plus intéressantes sont celles qui sont minimales pour l'inclusion, permettant ainsi d'obtenir des informations très précises sur les dépendances entre variables et contraintes apparues pendant la recherche. Malheureusement, un tel calcul prend énormément de temps [Jun01]. Un bon compromis entre précision et rapidité de calcul est d'essayer d'utiliser la connaissance intégrée dans le solveur lui-même pour fournir les explications. En effet, les solveurs de contraintes savent toujours pourquoi (hélas, rarement explicitement) ils retirent des valeurs dans les domaines des variables. En explicitant cette connaissance, des explications relativement précises et en tous cas intéressantes peuvent alors être fournies. Pour cela, il faut modifier le solveur pour qu'il fournisse ces explications. On pourra se reporter à [JB00] ou [Jus97] pour plus de détails sur une telle mise en œuvre.

4.3 Utilisations des explications

Les utilisations possibles des explications sont multiples [JDB00, JB00, JL00]. Les premières applications qui viennent à l'esprit ont trait au débogage : expliquer **clairement** les échecs, expliquer les différences entre le comportement observé et le comportement attendu (pourquoi la variable x ne peut-elle jamais valoir 4 ?).

Mais les explications peuvent aussi être utilisées pour identifier l'impact direct ou indirect d'une contrainte donnée sur les domaines des variables du problème et ainsi aider à son retrait dynamique. Comme par exemple dans le système de justifications utilisés par [Bes91] pour résoudre des CSP dynamiques qui est en fait un système partiel d'explications.

Plus encore, sachant expliquer les échecs et sachant retirer une contrainte dynamiquement, il devient aisé de proposer un système dynamique de résolution de problèmes sur-contraints [JB97].

Enfin, d'autres applications moins directes des explications peuvent être envisagées : en particulier, l'utilisation des explications pour guider la recherche. En effet, une recherche classique (basée sur le *backtrack*) de solutions pour un CSP ne progresse que face à des échecs (le plus classiquement par un retour au dernier point de choix). Les explications d'échecs peuvent justement servir à faire mieux que le *backtrack* standard et exploiter l'information ainsi obtenue pour améliorer la recherche : pour mettre en place un mécanisme intelligent [GJP00], pour remplacer le *backtrack* classique par une approche par saut à la *dynamic backtracking* [Gin93, JDB00], ou même pour développer une méthode de recherche locale sur une instantiation partielle des variables du problème [JL00].

[3] Un *nogood* est dit pertinent (*relevant*) si toutes les contraintes le constituant – en particulier les décisions prise lors de l'énumération – sont encore valides dans l'état courant du système [BM96].

Mais, ce qui nous intéresse plus particulièrement ici, c'est la possibilité, d'une part, d'expliquer pourquoi on ne peut trouver de solution à un problème donné et, d'autre part, de favoriser une recherche guidée par les données (*i.e.* par l'utilisateur via l'environnement de développement intégré).

5 Application au problème

La détection d'une micro-architecture similaires à un patron de conception à l'aide de la PPC avec explications se réalise en trois étapes :

1. le code source de l'utilisateur à analyser est modélisé afin de ne retenir que les informations nécessaires à l'application des contraintes : les noms des classes – formant le domaine des variables du CSP –, et les relations entre classes – permettant de vérifier la satisfaction des contraintes ;
2. modélisation d'un patron de conception du référentiel sous forme d'un CSP : l'architecture solution préconisée par un patron de conception est modélisée par un ensemble de contraintes. Une variable[4] est associée à chaque classe définie par le patron de conception. Les relations entre les classes (par exemple héritage, association ou encore connaissance) sont exprimées par des contraintes sur ces variables ;
3. résolution du CSP et recherche de solutions dégradées : une fois les solutions complètes du CSP trouvées, la recherche des solutions dégradées est guidée dynamiquement par l'utilisateur à l'aide des informations (explications de contradiction) fournies par le système.

5.1 Une bibliothèque de contraintes dédiées

À partir des relations entre classes définies dans [GHJV94], nous avons construit une première bibliothèque de contraintes. Les relations d'héritage, de composition, de connaissance, ... ont été exprimées par des contraintes dédiées. Ces contraintes sont binaires, quelconques, et définies sur des paires. Elles mettent en jeu des variables représentant à chaque fois une seule classe[5].

Les contraintes proposées dans notre implémentation permettent de couvrir un large spectre de patrons de conception. Cependant, certains patrons de conception sont difficiles à exprimer et nécessitent la définition de nouvelles relations ou la décomposition de relations en sous-relations.

Les contraintes génériques suivantes sont fournies (elles peuvent être combinées pour fournir des contraintes plus complexes) :

– **Héritage strict** : établit une relation d'héritage strict entre deux classes – entre deux variables – telle que définie dans l'exemple 3. Cette relation est exprimée par la contrainte `StrictInheritanceConstraint`. De cette notion d'héritage strict, une relation d'héritage (contrainte `InheritanceConstraint`) est dérivée telle que $\texttt{A} < \texttt{B}$

[4] Les variables dans notre modèle sont entières. Elles ont pour domaine l'ensemble des classes existantes dans le code source étudié. Chaque classe est unique et est identifiée par un entier.

[5] Comme les outils que nous utilisons ne gèrent pas (encore) les contraintes sur les ensembles, nous utilisons une astuce simple à laquelle les patrons de conception se prêtent bien : les variables représentant des ensembles de classes ne sont pas énumérées lors de la résolution. La cohérence des domaines des variables représentant un ensemble solution est alors assurée par les mécanismes de propagation.

ou `A` = `B` (les deux variables peuvent ici représenter la même classe contrairement à la contrainte précédente).

Exemple 3 (La contrainte d'héritage strict) :
Une relation d'héritage lie deux classes dans une relation de type parent–enfant ou super-classe–sous-classe. Dans le cas de l'héritage simple, la relation d'héritage crée un ordre partiel $<$ sur l'ensemble des classes E. Pour tout couple de classes distinctes `A` et `B`, si `B` hérite de `A` alors : `A` $<$ `B`
La contrainte associée avec la relation d'héritage strict est une contrainte binaire entre deux variables (classes) `A` et `B`. La sémantique opérationnelle de cette contrainte est (d_X représente le domaine de la variable `X`) :

$$\forall C_A \in d_A, \exists C_B \in d_B, C_A < C_B$$

$$\forall C_B \in d_B, \exists C_A \in d_A, C_A < C_B$$

- **Connaissance** : établit une relation de connaissance entre deux classes. Une classe `A` connaît une classe `B` si des méthodes définies dans `B` sont invoquées par `A`. On note cette relation `A` $\triangleright$ `B`. C'est une notion binaire, orientée et non-transitive. Cette relation est vérifiée par la contrainte `RelatedClassesConstraint`.
- **Non-connaissance** : assure la relation *opposée* de la relation précédente, la classe `B` ne doit pas avoir connaissance de la classe `A`. Cette relation est exprimée par la contrainte `UnRelatedClassesConstraint`.
- **Composition** : assure que deux classes sont en relation de composition. Une classe `A` est composée d'instances d'une classe `B` si la classe `A` définit un ou plusieurs champs de type `B`. On écrit `A` $\supset$ `B`. Il s'agit d'une relation binaire, orientée, et non-transitive. Cette relation est mise en place à l'aide de la contrainte `CompositionConstraint`.
- **Création** : établit qu'une classe `A` instancie (au moins une fois) une autre classe `B`. On note cette relation `A` $\overset{\star}{\triangleright}$ `B`. C'est une notion binaire, orientée et transitive. Cette relation est assurée par la contrainte `CreationConstraint`.
- **Appartenance à la liste d'héritage** : établit l'appartenance d'une classe à la liste des super-classes directes et indirectes d'une autre classe. Cette appartenance est vérifiée par la contrainte `InheritancePathConstraint`.
- **Type d'un champ**[6] : assure que le champ `f` de la classe `A` est de type `B` : `A.f` = `B`. Ce lien est établi à l'aide de la contrainte `PropertyTypeConstraint`.
- **Appartenance à un champ de type list**[6] : assure que la classe `B` appartient à la liste définie par le champ `f` de la classe `A` : `B` $\in$ `A.f`. Ce lien est établi à l'aide de la contrainte `ListPropertyTypeConstraint`.

5.2 Fonctionnement du solveur

Avoir une bibliothèque de contraintes adaptées à notre problème ne suffit pas. En effet, nous ne sommes pas réellement intéressés par les solutions complètes – solutions correspondants exactement à la modélisation du patron de conception recherché – mais surtout par les solutions dégradées. Pour cela, nous utilisons les spécificités de la programmation par contraintes avec explications. Une fois toutes les solutions complètes

[6]Cette contrainte est une contrainte générique utilisée pour définir de nouvelles contraintes plus simplement.

(respectant l'ensemble des contraintes) trouvées, le système est capable de fournir une explication à l'absence d'autres solutions complètes. Cette explication est donnée à l'utilisateur sous forme d'un choix de contraintes à relaxer ; l'utilisateur peut alors choisir une contrainte à relaxer – une contrainte dont il accepte la non-vérification.

Cette interaction permet de diriger la recherche vers les patrons intéressants. Pour aider l'utilisateur, il convient d'attribuer une importance relative à chacune des contraintes du problème, permettant ainsi de classer *a priori* les dégradations intéressantes. Les solutions dégradées fournies sont annotées par leur écart par rapport à la solution complète idéale (écart exprimé à l'aide d'une métrique – une valeur entière – définie lors de la modélisation).

L'ensemble maximal des solutions (complètes et dégradées) dépend uniquement de la modélisation choisie. Cependant, l'utilisateur peut choisir de relaxer les contraintes dans un ordre différent de celui proposé dans la modélisation. L'utilisateur peut ainsi faire apparaître plus rapidement les solutions dégradées qui lui semblent intéressantes et contourner la difficulté d'attribution *a priori* d'une importance relative aux contraintes reflétant correctement ses objectifs.

L'interaction entre l'utilisateur et le système de contraintes est très important pour notre application d'identification des patrons de conception. Il est important que l'utilisateur est un contrôle total sur sa recherche. C'est pourquoi nous avons préféré la PPC avec explications à d'autres techniques qui ne permettent pas un haut niveau d'interaction avec l'utilisateur, par exemple un CSP valué.

5.3 Application sur le patron Composite

Modéliser un patron de conception consiste donc à décrire un CSP (*cf.* exemple 4). Ensuite, le source code, présenté figure 2, est modélisé : les classes de l'application et leur relations sont encodées dans des tables (*cf.* exemple 5). Notre solveur de contraintes avec explications PALM [JB00] résout alors le CSP sur le model du code source pour identifier les sous-ensembles de classes dont la structure est similaire à la micro-architecture définie par le patron de conception.

> **Exemple 4 (La modélisation du patron de conception Composite) :** Le patron de conception Composite, tel que présenté précédemment (*cf.* exemple 1), est modélisé en associant une variable à chacune des classes définies (`component`, `composite` et `leaf`) et en contraignant les valeurs de ces variables par rapport aux relations entre les classes : `composite` $<$ `component`, `leaf` $<$ `component`, et `composite` $\supset$ `component`.

La recherche de solution offre alors toutes les solutions dégradées attendues pour corriger le noyau de notre application, exemple 2.

6 Résultats obtenus

Notre outil, PATTERNS TRACE IDENTIFICATION, DETECTION AND ENHANCEMENT FOR JAVA[7] (PTIDEJ), intègre les différentes étapes, présentées section 2.2, pour l'amélioration de la qualité du code du point de vue de l'architecture. Cet outil, écrit en JAVA, accepte

[7] Une démonstration est disponible à `www.yann-gael.gueheneuc.net/Work/PtidejDemo.html`

Exemple 5 (La modélisation du code source) :
Le code source de l'application, example 2, met en jeu sept classes : `AbstractDocument`, `Element`, `Title`, `Paragraph`, `ParaIndent`, `Document` et `Main`. Le domaine de chaque variable du CSP présenté dans l'example 4, `component`, `composite`, et `leaf`, est de taille 7 (une entrée pour chaque classe possible du code source). Nous avons défini un modèle générique pour encoder les classes du code source dans notre système. Ce modèle générique est une table :

```
PClass
    name:           string,
    superclasses:   list[PClass],
    components:     list[PClass],
    componentsType: PClass,
    relatesTo:      list[PClass],
    doNotRelateTo:  list[PClass]
```

Les relations entre les classes sont encodées dans ce modèle et sont utilisées pour vérifier les relations requises par le patron de conception :
- `name` représente le nom de la classe.
- `superclasses` est la liste des super-classes directes de la classe considérée.
- `components` est la liste de tous les composants agrégés par la classe considérée. `componentsType` est la super-classe commune à ces composants.
- `relatesTo` est la liste de toutes les classes connues par la classe considérée.
- `doNotRelateTo` est la liste de toutes les classes non-connues par la classe considérée.
- `create` est la liste de toutes les classes dont une instance est créée par la classe considérée.

Toutes ces informations peuvent être automatiquement déduites à partir du code source de l'application.

du code source JAVA. Il utilise des contraintes écrites en CLAIRE [CL96] v2.5.4 et le solveur de contraintes avec explications PALM [JB00] développé sur le système de PPC CHOCO [Lab00]. Il permet :

- de charger et visualiser (à la OMT) une application à partir de source JAVA ;
- de générer le modèle correspondant de l'application pour le système de contraintes ;
- d'appeler le système de contraintes PALM sur ce modèle pour détecter les différents patrons de conception chargés en mémoire (sous forme de CSP) ;
- de visualiser les solutions (complètes et surtout dégradées) trouvées ;
- d'effectuer les transformations nécessaires sur le code source de l'application pour en améliorer la qualité architecturale en le rapprochant d'un patron reconnu ;
- et enfin, de charger l'application modifiée et de l'afficher.

Pour l'instant, cinq patrons de conception sont intégrés à l'outil. Il s'agit de :

- Composite, vu section 2.1 ;
- Façade, un patron de conception modélisant les relations entre un ensemble de classes *clients* et un autre ensemble de classes formant un *sous-système* par l'intermédiaire unique d'une classe `Façade` et sans qu'il y ait connaissance mutuelle[8] ;
- Mediator, un patron de conception similaire à Façade, dans lequel les classes des *clients* et celles du *sous-système* peuvent avoir connaissance les unes des autres ;
- Chain of Responsibility, un patron de conception qui évite le couplage entre l'émetteur et le receveur d'un message, en donnant la possibilité à plus d'un objet de répondre au message[9] ;
- et, Factory Method, un patron de conception qui défini une interface pour créer un objet, mais qui laisse ses sous-classes choisir quelle classe instancier[10].

Potentiellement, n'importe quel patron de conception peut être modélisé et intégré à l'outil. Cependant, les patrons de conception dits *structurels* (comme Composite ou Façade) sont plus faciles à modéliser que les patrons de conception de *comportement* ou *créateurs*, ces derniers nécessitant parfois des informations non-décidables à partir du code source (tel que le type d'un objet particulier dans une collection générique). Les patrons de conception AbstractFactory[11], Observer[12] ou Singleton[13] sont actuellement en cours de modélisation.

Nous avons appliqué notre approche de façon automatique à différents systèmes, notamment à deux paquetages de la bibliothèque JAVA v1.3 : dans les paquetages `java.awt`

[8] Le patron Façade est constitué de trois classes `Clients`, `Façade` et `SubsystemClasses` telles que : `clients` $\triangleright$ `facade` $\triangleright$ `subsystemClasses`, `subsystemClasses` $\not\triangleright$ `facade` $\not\triangleright$ `clients` et `subsystemClasses` $\not\triangleright$ `clients`.

[9] Le patron de conception de comportement Chain of Responsibility comporte trois classes `Clients`, `Handler` et `ConcreteHandlers` telles que : `clients` $\triangleright$ `handler`, `handler` $\triangleright$ `handler`, `clients` $\not\triangleright$ `concreteHandlers` et `handler` $<$ `concreteHandlers`.

[10] Le patron Factory Method est un patron de conception créateur. Il est constitué de quatre classes `Creator`, `ConcreteCreator`, `Product` et `ConcreteProduct` telles que : `creator` $\triangleright$ `product`, `concretecreator` $\overset{\star}{\triangleright}$ `concreteProduct`, `creator` $<$ `concreteCreator` et `product` $<$ `concreteProduct`.

[11] Le patron de conception créateur AbstractFactory fournit une interface pour créer des familles d'objets dépendants les uns des autres sans spécifier leurs classes concrètes.

[12] Observer est un patron de conception de comportement qui défini une dépendance entre objets de telle sorte que lorsqu'un objet change d'état, tous les objets dépendants sont avertis et mis à jour automatiquement.

[13] Le patron de conception Singleton est un patron créateur. Il assure qu'une classe a une unique instance et il fournit un point d'entrée global sur celle-ci.

et `java.net`, les occurrences connues des patrons de conception Composite et Façade[14] ont été identifiées, ainsi que d'autres occurrences dégradées moins connues. Ces résultats sont prometteurs mais les paquetages doivent maintenant être analysés manuellement pour trouver d'éventuelles solutions dégradées que les modèles n'ont pas permis de détecter : nous devons vérifier la *modélisation* des patrons de conception, pas la *méthode*, qui a été prouvée complète.

7 Conclusion

Dans cet article, nous avons présenté une application originale de la programmation par contraintes avec explications pour faire un premier pas dans un problème difficile et pour lequel cette méthodologie semble particulièrement adaptée : l'identification de patrons de conception dans du code source provenant d'un langages à objets.

Nous avons développé une bibliothèque de contraintes dédiées à ce problème que nous utilisons dans l'outil PTIDEJ. Les résultats obtenus par notre approche sont satisfaisants parce qu'ils permettent, pour la première fois, de proposer un outil pour résoudre ce problème.

Nos travaux actuels concernent la définition de plus de relations entre classes, l'extension de la bibliothèque de contraintes et surtout son application à d'autres bibliothèques JAVA, comme JHOTDRAW [Gam98].

Références

[Bes91] Christian Bessière. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings AAAI'91*, 1991.

[BM96] Roberto J. Bayardo Jr. and Daniel P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI'96*, 1996.

[Bro96] Kyle Brown. Design reverse-engineering and automated design pattern detection in smalltalk. Technical Report TR-96-07, University of Illinois at Urbana-Champaign, 1996.

[CL96] Yves Caseau and François Laburthe. CLAIRE : Combining objects and rules for problem solving. *Proceedings of JICSLP, workshop on multi-paradigm logic programming*, 1996.

[DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Object-oriented reengineering OOPSLA'00 tutorial. *OOPSLA Tutorial Notes*, 2000.

[FLMN00] B. Fuchs, J. Lieber, A. Mille, and A. Napoli. An Algorithm for Adaptation in Case-Based Reasoning. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000), Berlin, Germany*, pages 45–49, 2000.

[Fow99] Martin Fowler. *Refactoring - Improving the Design of Existing Code.* Addison-Wesley, 1999.

[Gam98] Erich Gamma. JHotDraw, 1998. Available at http ://members.pingnet.ch/gamma/JHD-5.1.zip.

[14] Par exemple, le groupe de classes `Component`, `Container` et `Button` est un exemple du patron de conception Composite dans `java.awt`.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Gin93] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1 :25–46, 1993.

[GJP00] Christelle Guret, Narendra Jussien, and Christian Prins. Using intelligent backtracking to improve branch and bound methods : an application to open-shop problems. *European Journal of Operational Research*, 127(2) :344–354, 2000.

[JB97] Narendra Jussien and Patrice Boizumault. Best-first search for property maintenance in reactive constraints systems. In *International Logic Programming Symposium*, pages 339–353, Port Jefferson, N.Y., USA, October 1997. MIT Press.

[JB00] Narendra Jussien and Vincent Barichard. The palm system : explanation-based constraint programming. In *Proceedings of TRICS : Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, September 2000.

[JDB00] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in Lecture Notes in Computer Science, pages 249–261, Singapore, September 2000. Springer-Verlag.

[JL00] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. In *Seventh National Conference on Artificial Intelligence – AAAI'2000*, pages 169–174, Austin, TX, USA, August 2000.

[Jun01] Ulrich Junker. QUICKXPLAIN : Conflict detection for arbitrary constraint propagation algorithms. Technical report, Ilog SA, 2001.

[Jus97] Narendra Jussien. *Relaxation de Contraintes pour les problèmes dynamiques*. 1. thèse, Université de Rennes I, 24 October 1997.

[JZ97] Jens Jahnke and Albert Zündorf. Rewriting poor design patterns by good design patterns. *Proceedings the Workshop on Object-Oriented Reengineering at ESEC/FSE*, September 1997.

[KP96] Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. *Proceedings of the Working Conference on Reverse Engineering*, pages 208–215, 1996.

[Lab00] François Laburthe. CHOCO's API. Technical Report Version 0.13, OCRE Committee, 2000.

[MMCG98] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch : A clustering tool for the recovery and maintenance of software system structures. *Proceedings of ICSM*, 1998.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. *Proceedings of ICSM*, 1999.

[Rég95] Jean-Charles Régin. *Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique.* Thèse de doctorat, Université de Montpellier II, 21 December 1995. In French.

[SV94] Thomas Schiex and Gérard Verfaillie. Nogood Recording fot Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2) :187–207, 1994.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. *Proceedings of TOOLS USA*, pages 112–124, 1998.