
Un méta-modèle pour coupler application et détection des design patterns

Hervé Albin-Amiot* — Pierre Cointe — Yann-Gaël Guéhéneuc**

École des Mines de Nantes
4, rue Alfred Kastler – BP 20 722 – 44 307 Nantes cedex 3 – France
{albin, cointe, guehene}@emn.fr

* *Soft-Maint*
4, rue du Château de l'Éraudière – BP 72 438 – 44 324 Nantes cedex 3 – France

** *Object Technology International Inc.*
2670 Queensview Drive – Ottawa, Ontario, K2B 8K1 – Canada

RÉSUMÉ *Les design patterns (ou patrons de conception) sont reconnus comme une des bonnes techniques du génie logiciel à objets. Cette technique améliore le cycle de vie du logiciel en facilitant la conception, la documentation, la maintenance et la rétro-conception. Peu d'environnements de développement intégrés (EDIs) sont à la fois outillés pour inciter l'utilisateur à appliquer un catalogue de patterns et pour l'aider à améliorer ses programmes en y extrayant automatiquement des micro-architectures correspondant à celles de patterns. Ce papier présente une partie de nos travaux visant à outiller l'EDI Visual Age pour Java en lui adjoignant un catalogue recensant l'ensemble des patterns du GoF et deux assistants dédiés à l'application et à la détection de patterns. Nous proposons un méta-modèle permettant de décrire ces différents patterns, de les manipuler, de les synthétiser et de les reconnaître dans des programmes existants. Nous discutons les limites de ce méta-modèle apparues après expérimentation et suggérons comment l'améliorer pour prendre en charge l'aspect intentionnel des patterns et supporter la reconnaissance de micro-architectures voisines de celles de patterns déjà répertoriés.*

ABSTRACT. *Design patterns are a good technique of OO software engineering: They improve the life cycle of software systems by easing the conception, the documentation, and the maintenance. Few integrated development environments (IDEs) propose tools instigating the user to apply patterns and to improve existing programs by extracting patterns. This paper presents a part of our research to add two assistants to the Visual Age for Java IDE: An assistant to apply patterns; An assistant to detect patterns. Both assistants work with a common meta-model describing the patterns, to manipulate them, to generate source code, and to detect them. We discuss the limitations of this meta-model, and we suggest enriching it to include pattern intent and to detect proximal pattern micro-architectures.*

MOTS-CLÉS. *Patrons de conception, méta-modélisation, génération de code, détection, EDI.*

KEYWORDS. *Design patterns, meta-modelling, code generation, detection, IDE.*

1. Introduction

La présentation des *design patterns*¹ en général, et ceux du GoF [GAM 95] en particulier, a suscité depuis 1995 beaucoup d'intérêt dans la communauté du génie logiciel à objets. Néanmoins, force est de constater qu'au-delà d'un nombre important de papiers de recherche² proposant d'utiliser ces patterns dans la définition et la documentation de bibliothèques de classes ou dans l'élaboration de *frameworks*, peu d'outils logiciels permettent de les intégrer dans le cycle de vie du logiciel [CHA 00]. Ainsi, quant il s'agit d'exposer leurs développeurs à des problèmes de conception récurrents et de les sensibiliser aux solutions correspondantes, beaucoup d'entreprises en sont réduites à leur proposer la lecture du GoF, présenté comme le « bon livre de recettes ».

Notre objectif est d'outiller un environnement de développement intégré (en l'occurrence Visual Age pour Java) pour lui ajouter les assistants logiciels permettant : i) de représenter un catalogue de patterns ; ii) d'instancier les patterns de ce catalogue pendant le développement ; iii) de reconnaître ces patterns dans du code source existant. Ces assistants logiciels doivent nous permettre de mener deux expérimentations duales :

- 1) la programmation visuelle pour élaborer le choix d'un pattern répertorié dans un catalogue, l'adapter par instanciation pour une application donnée et générer le squelette du programme correspondant ;
- 2) la détection et la correction semi-automatique des défauts de conception apparaissant à l'analyse de code source existant.

Il se trouve que la réalisation de tels assistants pose d'abord le problème de la représentation des différents aspects associés à ces patterns (intention, motivation, structure, participants...). Cette représentation doit garantir la traçabilité [SOU 95] des patterns de la phase de conception à celle d'implémentation pour permettre l'aller-retour (*roundtrip*) entre les phases d'application et de détection [ALB 01b].

La traçabilité nous semble essentielle dans la phase d'apprentissage et de mise en œuvre des patterns, l'utilisateur devant simultanément « jongler » entre la connaissance d'une recette répertoriée dans un catalogue et la détection pratique de celle-ci dans une application donnée.

¹ A défaut d'une traduction française consensuelle, nous utiliserons le terme original.

² Une présentation des travaux les plus représentatifs est proposée dans [BOR 99].

Il existe deux approches pour représenter les patterns et en assurer la traçabilité³ :

- 1) les réifier directement au niveau du langage de programmation, comme dans [BAU 96, BOS 97, DUC 97, TAT 98]. Cette solution conduit à définir son propre langage ou à étendre un langage donné, donc à s'écarter des standards, ce qui est en contradiction avec notre objectif d'outiller un environnement de développement dédié à Java ;
- 2) les réifier au niveau de la conception par l'intermédiaire d'un méta-modèle, comme dans [EDE 00, FLO 97, PAG 96, RAP 00, SUN 99, SUN 00]. Dans ce cas, la traçabilité n'est pas assurée *de facto* et doit être prise en charge par des mécanismes dédiés.

Dans un souci d'uniformité, nos outils chargés de réaliser les trois aspects liés à la représentation, l'application et la détection d'un pattern, reposent sur un même méta-modèle dont la justification, la description et l'étude constituent le cœur de ce papier. Le détail de son intégration au sein des assistants dédiés à l'application et à la détection est proposé dans [ALB 01a].

Dans un premier temps, nous présentons et justifions ce méta-modèle en insistant sur la description des patterns du GoF sous forme de rôles (entités et éléments). Nous décrivons une première mise en œuvre permettant de réaliser l'application de patterns en Java (cf. l'outil PatternsBox) et proposons un premier algorithme de détection. Après une validation expérimentale, nous présentons les limites de notre approche et proposons des solutions pour y remédier.

2. Méta-modélisation et design patterns

L'un des problèmes récurrents dans l'utilisation des patterns concerne leur représentation dans les différentes étapes du cycle de vie du logiciel. En particulier, il est important de pouvoir en donner une représentation manipulable à la fois dans les phases de conception et d'implémentation. Comme beaucoup d'autres auteurs (par exemple [EDE 00, SUN 99]), nous avons choisi d'en proposer une description au niveau de la conception. Cette description doit permettre la synthèse de code au niveau d'un langage de programmation, tout en garantissant la traçabilité des patterns, c'est-à-dire la possibilité d'identifier un modèle de pattern à partir d'un code source donné.

Les techniques de méta-modélisation consistent en la définition d'un ensemble de méta-entités élémentaires, qui permettent d'obtenir par composition la description d'un modèle de pattern. Ces techniques peuvent s'appliquer au domaine des patterns pour rendre compte de leurs principaux aspects, comme l'intention, la structure,

³ On ne parle pas ici des approches par transformation de modèles telles que [DES 99].

l'implémentation, l'applicabilité... Néanmoins, il est important de noter que du fait de l'aspect informel des patterns, un méta-modèle ne peut prétendre capturer ce qu'est un pattern *en général* mais seulement la description opérationnelle d'un ou de plusieurs de ses aspects. Un méta-modèle rend compte de la « palette d'utilisation » d'un pattern, c'est-à-dire des variations et approximations possibles par rapport aux cas d'utilisation considérés.

Ainsi, si plusieurs méta-modèles ont été proposés dans la littérature, chacun d'eux s'intéresse à des aspects bien précis. Dans [RAP 00] et [PAG96] la modélisation concerne l'instanciation et la validation de patterns mais la génération de code n'est pas abordée. Dans l'outil PatternGen [SUN 99], le méta-modèle n'assure ni la production de code (elle est prise en charge par un autre module) ni la détection. Dans [FLO 97], le système de fragments ne permet que la représentation de patterns. Dans [EDE 00], le méta-modèle autorise la validation et l'application de patterns, la génération étant prise en compte par un module externe. Enfin, les méta-patterns de [PRE 95] servent essentiellement à la documentation de patterns.

Jusqu'à présent, aucun méta-modèle n'est réellement destiné à produire automatiquement des programmes tout en supportant un mécanisme de détection, couplage qui nous intéresse en premier lieu. Aussi, avons-nous été amenés à définir notre propre méta-modèle, supportant génération de code et détection, et assurant ainsi la traçabilité.

3. Présentation du méta-modèle

Afin de rendre cette présentation la plus didactique possible, la description du méta-modèle est réalisée en trois temps. Les principes de base sont rapidement posés pour esquisser ensuite les grandes lignes de son implémentation et conclure par la modélisation du pattern *Composite*.

3.1. Principe

Notre principal objectif dans la définition d'un méta-modèle de patterns est d'obtenir un langage de description aussi concis que possible dont chacun des constituants puisse être aisément traduit dans un langage de programmation et, de manière duale, être détecté dans une implémentation existante. Ainsi, notre méta-modèle sert à la fois dans les processus de génération et de détection.

Le méta-modèle définit un ensemble de constituants (entités et éléments) et les règles régissant leurs interactions. Ces constituants permettent de décrire la structure et le comportement des patterns du GoF. Il propose en outre une hiérarchie de

classes associée au système de génération (`PatternBuilder`), une classe en charge de la détection (`PatternIntrospector`⁴), ainsi qu'un entrepôt de patterns (`PatternsRepository`). La figure 1, ci-dessous, propose une version simplifiée du méta-modèle ne présentant que les constituants utiles à la compréhension de ce papier :

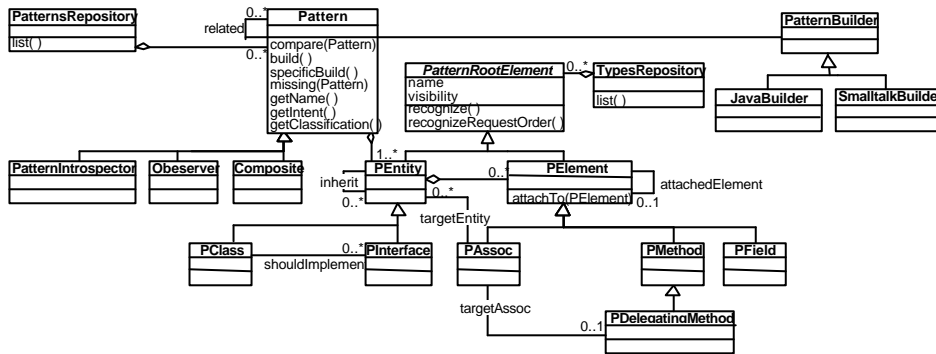


Figure 1. Schéma UML (simplifié) du méta-modèle

Un modèle de pattern (par exemple *Composite* ou *Observer*) prend la forme d'une instance de la classe `Pattern` ou de l'une de ses sous-classes. Il est constitué d'une collection d'entités (`PEntity`), représentant la notion de participants au sens du GoF. Chaque entité est composée d'une collection d'éléments (`PElement`) traduisant ses attributions. Ces entités et éléments décrivent la structure et le comportement du modèle et sont ajoutés au fur et à mesure de la modélisation de nouveaux patterns, par spécialisation des classes `PEntity` et `PElement`. L'héritage est donc utilisé pour produire une taxinomie des éléments constitutifs des patterns.

En suivant la sémantique définie par ce méta-modèle, un modèle de pattern est composé d'une ou plusieurs classes ou interfaces, instances respectivement de `PClass` et `PInterface`, et sous-classes de `PEntity`. Une instance de `PEntity` peut contenir des méthodes (`PMethod`), des champs (`PField`) ou, d'une manière plus générale, n'importe quelle instance de `PElement`, qu'il s'agisse d'une association ou d'une délégation :

- les relations d'association binaires et unidirectionnelles (seules à être utilisées dans le GoF) sont représentées par la classe `PAssoc`. Une association liant une classe A à une classe B est représentée par une instance de `PAssoc` agrégée par A et référençant B ;

⁴ La classe `PatternIntrospector` est sous-classe de la classe `Pattern` car ses instances représentent les descriptions des patterns détectés dynamiquement à l'analyse du code source.

6 L'objet – 8/2002. LMO'2002

- la délégation est exprimée d'une manière similaire par la classe `PDelegatingMethod`. Lors de la délégation du comportement d'une méthode `a()` de la classe `A` à une méthode `b()` de la classe `B`, une instance d'une `PDelegatingMethod a()` agrégée par `A` et référençant `b()` est utilisée. Elle référence en outre l'association entre les deux classes afin de déduire de sa cardinalité le traitement à réaliser : envoi simple ou *multicast*.

Ce méta-modèle supporte un certain nombre de services visant à faciliter la modélisation, la génération de code et la détection de patterns. Nous en présentons cinq, utiles à la compréhension de la suite de ce papier :

- `assumeAllInterfaces()` : définie au niveau de `PClass`, cette méthode permet de générer automatiquement tous les éléments (méthodes) manquants à une classe pour qu'elle satisfasse l'ensemble des contrats associés aux interfaces qu'elle implémente.
- `attachTo()` : définie au niveau de `PElement`, cette méthode crée un lien entre deux éléments. On l'utilise, par exemple, pour spécifier un lien d'implémentation entre un élément abstrait et sa version concrète et maintenir ainsi une compatibilité entre les deux. Dans le cas de `PMethod`, ce mécanisme est utilisé pour expliciter la surcharge par masquage.
- `getActor()` : permet d'obtenir, à partir d'une instance de la classe `Pattern`, l'entité (instance de `PEntity`) jouant le rôle spécifié. Les rôles accessibles pour chaque pattern sont ceux décrits dans la section *Participants* du GoF. Le même message à destination d'une instance de `PEntity` retourne une instance de `PElement` (le nom d'acteur est alors celui utilisé dans la section *Structure* du GoF).
- `recognize()` : chaque élément ou entité propose cette méthode qui figure le point d'entrée du système de détection qui lui est associé. Le méta-modèle pouvant être enrichi par ajout d'entités ou d'éléments, chacun doit être en mesure de reconnaître parmi un ensemble de constructions du langage lesquelles peuvent lui être associées, par exemple les classes dans le cas de `PClass`.
- `toString()` : chaque élément ou entité du méta-modèle fournit son « équivalent code » par l'intermédiaire de la méthode `toString()`. La génération consiste alors pour le `PatternBuilder` (cf. Fig. 1) à appeler cette méthode sur les classes appropriées et à réaliser l'agencement de l'ensemble. L'intérêt de cette approche est qu'elle ne nécessite pas de modification du système de génération lors de l'ajout d'un nouvel élément ou entité. Pour une génération multi-langages une solution inspirée du pattern *Visitor* du GoF a été mise en place.

3.2. *Inspiration des JavaBeans*

La définition d'un méta-modèle pose la question de la représentation des méta-entités et de leurs interactions. Pour ce qui nous concerne, et compte tenu de notre désir d'expérimenter dans un environnement de développement dédié à Java, nous avons choisi :

- de définir les méta-entités permettant de décrire les éléments constitutifs du modèle de classes Java (i.e., les classes, les interfaces, les méthodes, les champs...);
- de nous inspirer du modèle des JavaBeans [JB 97] pour décrire les relations et interactions entre ces méta-entités et réaliser un système de notification ;
- de nous inspirer également de ce modèle pour proposer un système d'introspection basé sur des conventions de nommage.

Chaque classe du méta-modèle supporte un mécanisme de notification (*listeners* dans la terminologie JavaBeans) pour chacune de ses propriétés. Ce mécanisme peut être utilisé pour restreindre la nature des modifications possibles (*veto-listeners*), on parle alors de propriétés contraintes. Le système de notification est mis à profit pour établir les différentes contraintes de définition des modèles de pattern. A titre d'exemple, si une `PClass` concrète, à l'écoute de tous les éléments qui la constituent, est notifiée qu'un de ceux-ci vient d'être déclaré abstrait, alors elle est automatiquement redéfinie abstraite. Si elle ne peut l'être, elle fait alors valoir son droit de veto et l'élément n'est pas modifié.

De plus, une déclinaison des idiomes associés aux JavaBeans est utilisée pour la définition et la détection des services propres à chaque pattern (sous-classe de `Pattern`) : un jeu de méthodes préfixées par `get` et `set` dénote une propriété simple en lecture-écriture et les préfixes `add` et `remove` désignent les propriétés de type « liste »⁵. Ainsi, l'introspection et le formalisme des JavaBeans permettent de découvrir et d'utiliser facilement une méthode telle que `addLeaf()` (qui ajoute dynamiquement de nouvelles feuilles pour le pattern *Composite*) ou `getApplicability()` (cf. [GAM 95]).

3.3. *Exemple du pattern Composite*

Nous développons une mise en œuvre du méta-modèle au travers de l'exemple du pattern *Composite* (cf. page 163 du GoF). Ce pattern est relativement simple et sert souvent d'exemple [SUN 00] pour l'évaluation à la fois des systèmes de génération de code [BUD 96] et des systèmes de détection [BRO 95, WUY 98]. Il

⁵ A distinguer des `add` et `remove` utilisées dans les JavaBeans pour la gestion des *listeners* [JB 97].

permet d'organiser des objets en une structure arborescente dans laquelle objets simples et compositions d'objets sont traités uniformément. La figure 2 en rappelle la structure. Nous utilisons la déclinaison *Safety* telle qu'elle est présentée dans la rubrique *Implementation* du GoF, sous l'intitulé : *Declaring the child management operations*. Il s'agit de l'utilisation la plus fréquemment rencontrée, par exemple : les classes *Component* et *Container* de l'AWT Java.

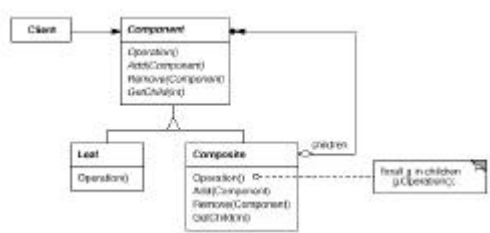


Figure 2. Structure du pattern Composite

Le schéma général d'utilisation du méta-modèle est résumé par la figure 3, dont les cinq points servent de fil conducteur tout au long des sous-sections suivantes, selon les trois aspects que sont la déclaration, l'application et la détection du pattern *Composite*.

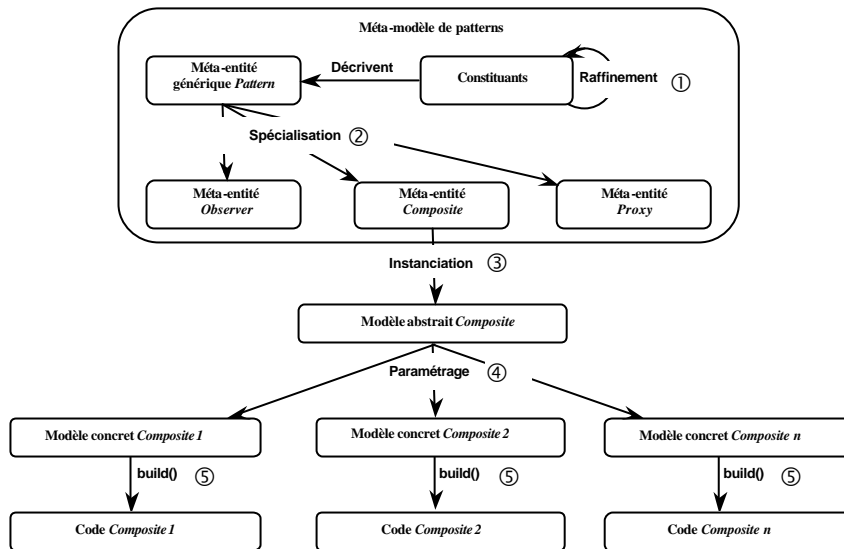


Figure 3. Schéma général d'utilisation du méta-modèle

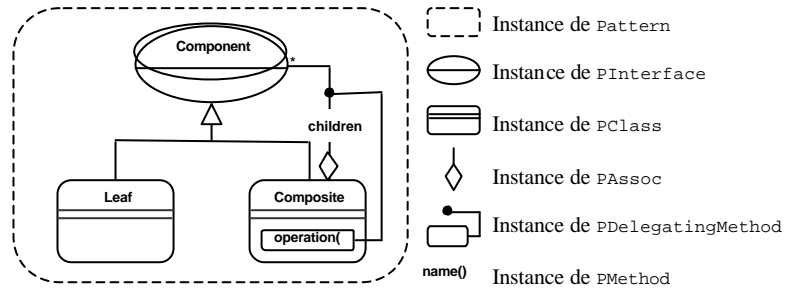


Figure 4. Description du leitmotiv du pattern Composite

3.3.1. Description du pattern Composite

La première étape consiste en la définition de la méta-entité *Composite* dont est issu le modèle de même nom. Cette définition est réalisée en deux temps.

Tout d’abord, le méta-modèle est raffiné (point ①, Fig. 3), pour rajouter à celui-ci tous les constituants structurels et comportementaux nécessaires à la modélisation du pattern. Ces constituants sont par la suite disponibles pour la modélisation d’autres patterns. Dans le cas de *Composite*, le méta-modèle présenté figure 1 est suffisant. Si la modélisation d’un pattern requérait l’introduction d’une nouvelle propriété de classe, par exemple la notion de classe immuable, une nouvelle entité, *ImmutablePClass*, spécialisant *PClass*, serait rajoutée.

Ensuite, la description de la méta-entité *Composite* (point ②, Fig. 3) se fait de manière impérative en suivant la sémantique définie par le méta-modèle, c’est-à-dire en énumérant ses différents constituants. La méta-entité *Composite* consiste en une particularisation de la méta-entité générique *Pattern* et se traduit donc par une spécialisation de celle-ci. La méta-entité *Composite* a principalement deux rôles : i) décrire les modèles de patterns qui pourront en être issus ; ii) fixer les règles d’interactions avec les instances du pattern *Composite* résultants de l’instanciation de la méta-entité *Composite*. On voit ici clairement la similitude avec la déclaration d’une classe dans un langage à objets. En terme de structure, la méta-entité *Composite* est construite à partir du schéma de classes proposé par le GoF (reproduite figure 2), et de toutes les informations informelles traduisant ce que Eden appelle le *leitmotiv* [EDE00] et qui représentent en fait la solution du pattern, c’est-à-dire une synthèse des rubriques *Structure*, *Participants* et *Collaboration* du GoF. Une représentation graphique de cette description est fournie figure 4. En terme de comportement, la méta-entité *Composite* décrit les services permettant d’assurer l’adaptation de ses instances a un contexte donné. Cela comprend, par exemple pour *Composite*, la définition de méthodes comme `addLeaf()`, `removeLeaf()` ou encore `addComposite()`.

Une déclaration partielle de la méta-entité *Composite* est donnée par la description de la classe Java *Composite* suivante :

Définition de la méta-entité <i>Composite</i>	Commentaires
<code>class Composite extends Pattern {</code>	Meta-entité <i>Composite</i>
<code>Composite(...) {</code> ...	Déclaration du <i>leitmotiv</i> du pattern <i>Composite</i> , faite par le constructeur de la classe <i>Composite</i> , sous-classe de <i>Pattern</i>
<code>iComponent = new PInterface("Component")</code>	Déclaration de l'interface <i>Component</i>
<code>mOperation = new Pmethod("operation")</code> <code>iComponent.addPElement(mOperation)</code>	Déclaration de la méthode <code>operation()</code> , définie dans l'interface <i>Component</i>
<code>addPEntity(iComponent)</code>	Ajout de l'interface <i>Component</i> en tant qu'élément constitutif du pattern
<code>anAssoc = new PAssoc("children", iComponent, 2)</code>	Association ayant pour cible <i>Component</i> et de cardinalité > 1
<code>cComposite = new PClass("Composite")</code>	Déclaration de la classe <i>Composite</i>
<code>cComposite.addShouldImplement(iComponent)</code>	La classe <i>Composite</i> a pour interface <i>Component</i>
<code>cComposite.addPElement(anAssoc)</code>	L'association <code>children</code> lie <i>Composite</i> et <i>Component</i>
<code>aPDelegatingMethod = new PDelegatingMethod("operation", anAssoc) aPDelegatingMethod.attachTo(mOperation) cComposite.addPElement(aPDelegatingMethod)</code>	La méthode <code>operation()</code> définie dans la classe <i>Composite</i> implémente la méthode <code>operation()</code> de l'interface <i>Component</i> et est liée via le lien d'association <code>anAssoc</code> à cette même interface
<code>addPEntity(cComposite)</code>	Ajout de la classe <i>Composite</i> au pattern
<code>cLeaf = new PClass("Leaf")</code>	Déclaration de la classe <i>Leaf</i>
<code>cLeaf.addShouldImplement(iComponent)</code>	La classe <i>Leaf</i> a pour interface <i>Component</i>
<code>cLeaf.assumeAllInterfaces()</code>	L'interface publique de la classe <i>Leaf</i> est générée automatiquement (création d'une méthode <code>operation()</code>)
<code>addPEntity(cLeaf)</code>	Ajout de la classe <i>Leaf</i> au pattern
<code>}</code>	Déclaration de services propres au pattern ...
...	
<code>void addLeaf(String leafName) {</code> <code> PClass newPClass = new PClass(leafName)</code> <code> newPClass.addShouldImplement((PInterface)getActor("Component"))</code> <code> newPClass.assumeAllInterfaces()</code> <code> newPClass.setName(leafName)</code> <code> addPEntity(newPClass)</code> <code>}</code>	Exemple de la méthode <code>addLeaf()</code>
...	
<code>}</code>	

Un modèle abstrait du pattern *Composite* (point ③, Fig. 3) est obtenu par instantiation de la méta-entité correspondante, c'est-à-dire par instantiation de la classe Java *Composite*. On entend par modèle abstrait le *leitmotiv* préconisé dans le GoF et on parlera de modèle concret pour désigner la version liée au contexte d'application de ce même *leitmotiv*. Chaque modèle abstrait contient toute la logique du pattern ainsi qu'un lien vers le système de génération. Tous les modèles abstraits sont stockés dans un entrepôt de patterns (décrit par la classe *PatternsRepository*, Fig. 1). Cet entrepôt permet d'accéder à l'ensemble des

informations relatives à un pattern et de déterminer si la solution qu'il propose est bien celle attendue.

3.3.2. Application du pattern Composite

A partir du modèle abstrait, il est facile d'obtenir une réalisation particulière, un modèle concret, par paramétrage de ses constituants (point ④, Fig. 3). Par exemple, la hiérarchie de composants graphiques illustrant le pattern *Composite* dans le GoF et reprise dans [PAG 96] (voir la figure 5, ci-contre) se traduit par la suite d'expressions :

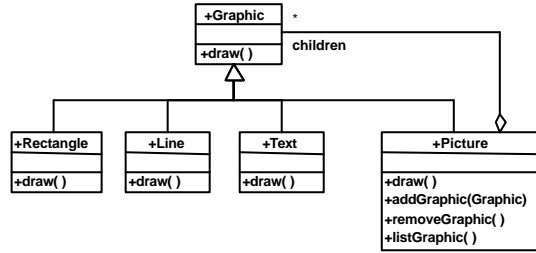


Figure 5. Schéma UML du modèle concret

Déclaration d'un modèle concret de <i>Composite</i>	Commentaires
<code>pComposite.getActor("Component").setName("Graphic")</code>	Définitions du nom des acteurs
<code>pComposite.getActor("Component").getActor("operation").setName("draw")</code>	
<code>pComposite.getActor("Leaf").setName("Text")</code>	
<code>pComposite.getActor("Composite").setName("Picture")</code>	
<code>pComposite.addLeaf("Line")</code> <code>pComposite.addLeaf("Rectangle")</code>	Ajout de nouvelles classes feuilles par utilisation de la méthode <code>addLeaf()</code> , spécifique à ce pattern

Cette « traduction » est réalisée automatiquement à l'aide de l'outil de programmation visuelle PatternsBox, dont un aperçu est donné figure 6.

La génération de code est réalisée à la réception du message `build()` (point ⑤, Fig. 3) par l'objet Java représentant le modèle concret du pattern. Celui-ci délègue l'appel à l'instance courante de la classe `PatternBuilder`, qui génère alors le code source, en itérant sur les divers constituants du pattern et en appliquant à chacun le message `toString()`.

3.3.3. Détection du pattern Composite

Le système de détection proposé est prévu pour fonctionner sur du code produit par application des différents modèles de pattern décrits par notre méta-modèle, et aussi sur des patterns qui ont été implémentés à la main. Il n'utilise aucun système de marquage du code et ne dispose d'aucune information particulière facilitant la détection. La détection est essentiellement basée sur la reconnaissance d'informations structurelles et peut être facilement étendue pour prendre en compte d'autres types d'information. Le système de détection utilise un entrepôt de types (la classe `TypesRepository`, Fig. 1) contenant tous les constituants disponibles pour la description d'un pattern, c'est-à-dire l'ensemble des `PEntities` et des `PElements` défini au sein du méta-modèle.

La détection est décomposée en deux étapes (duales des étapes associées aux points ④ et ⑤, Fig. 3) :

- 1) La classe `PatternIntrospector`, en charge de la détection, fournit tous les

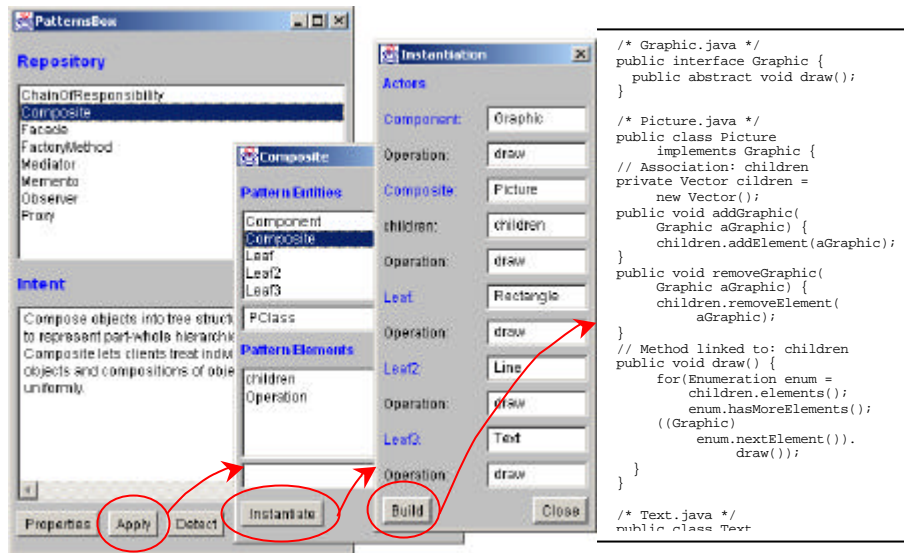


Figure 6. Vue partielle du processus de génération dans l'outil PatternsBox (points ③, ④ et ⑤, Fig. 3)

éléments syntaxiques trouvés dans le code source de l'utilisateur (classes, interfaces, méthodes...) aux différents types présents dans le `TypesRepository`. Chacun des types détermine, par l'intermédiaire de la méthode `recognize()`, les éléments qui lui correspondent. L'algorithme de reconstruction est le suivant :

Algorithme de reconstruction, classe <i>PatternIntrospector</i>
<pre> Soit C : liste initiale des classes utilisateur Soit P : modèle de schéma en cours de construction Soit E : liste des entités (PEntity) existantes Soit L : liste des éléments (PElement) existants Soit S : liste des entités ou éléments (classes, interfaces...) trouvés dans le code utilisateur Soit T : liste d'instances de PEntity Soit U : classe en cours d'examen S = C T = ∅ Pour chaque e de E et tant que size(S) > 0 S = e.recognize(S, P) T = P.listPElement() Pour chaque t de T U = Class.forName(t.getName()) S = U.getDeclaredConstructors() ∪ U.getDeclaredMethods() ∪ U.getDeclaredFields() Pour chaque l de L et tant que size(S)>0 S = l.recognize(S, P) </pre>

Méthode <code>recognize()</code> de <i>PEntity</i> (<code>e.recognize(S,P)</code>)	Méthode <code>recognize()</code> de <i>PElement</i> (<code>l.recognize(S,P)</code>)
<pre> Soit N : liste des entités non reconnues N = S Pour chaque s de S Si s = e alors P.addPEntity(new(s)) N = N - {s} Retour N </pre>	<pre> Soit N : liste des éléments non reconnus N = S Pour chaque s de S Si s = l alors P.getActor(s.getDeclaringClass(). getName()).addPElement(new(s)) N = N - {s} Retour N </pre>

A partir de ces informations, la classe `PatternIntrospector` construit un modèle concret, équivalent au code de l'utilisateur, uniquement avec les constituants définis par le méta-modèle. Une fois le modèle concret obtenu, celui-ci est comparé aux différents modèles abstraits de patterns présents dans le système et répertoriés dans l'entrepôt de patterns (la classe `PatternsRepository`, Fig. 1), afin de déterminer quels patterns ont été reconnus.

- 2) Chaque modèle abstrait examine le modèle concret soumis et détermine quelles entités (`PEntity`) de ce modèle peuvent être associées à ses différents rôles. Les critères suivants sont appliqués par défaut (un paramétrage est possible) :
 - un rôle du modèle de référence doit être rempli au niveau du modèle soumis par un acteur (entité) de même type (critère *a*) ;

- le modèle examiné doit contenir au moins autant d'entités que le modèle de référence (critère *b*) ;
- pour chaque rôle attribué du modèle de référence, l'entité correspondante du modèle soumis doit contenir au moins autant d'éléments que celle du modèle de référence (critère *c*) ;
- les liens d'héritage (critère *d*), de réalisation (critère *e*), d'association et d'agrégation (critère *f*) du modèle de référence doivent être présents au niveau du modèle soumis.

Après élagage (trois premiers critères), l'algorithme vérifie pour chaque critère restant (contrainte binaire) si une entité (valeur), pour un rôle donné (variable), dispose d'une entité support vérifiant la contrainte. On remarquera la similitude avec le problème de vérification de la consistance d'arc en programmation par contraintes. Dans notre cas, si nous arrivons à vérifier par filtrage la propriété de

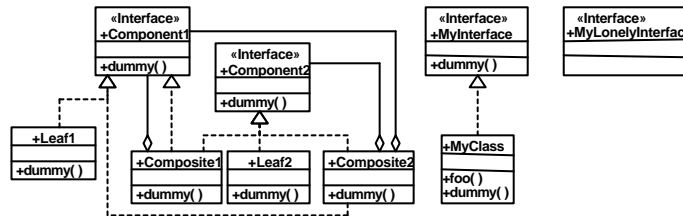


Figure 7. Schéma UML (simplifié) des classes à reconnaître

consistance d'arc, alors au moins une instance valide d'un pattern a été détectée. Nous utilisons pour le filtrage l'algorithme AC-1 proposé par Waltz en 1972 [WAL 72]. Sa complexité est en $O(n \cdot e \cdot d^3)$, avec *n* le nombre de rôles, *e* le nombre de critères et *d* le nombre maximum d'instances potentielles de PEntity pour un rôle donné. Cet algorithme est très coûteux mais il est simple à implémenter et s'avère suffisant pour les besoins de notre expérimentation.

L'exemple suivant porte sur la détection de la variante du pattern Composite présentée figure 7. Nous détaillons les différentes étapes de la détection sous forme d'un tableau récapitulant les réductions pratiquées :

	Critères	« Composite »	« Composite »	« Leaf »
1	<i>a, b, c</i>	MyInterface Component1 Component2	MyClass Composite1 Composite2	MyClass Composite1 Composite2 Leaf1 Leaf2
Les contraintes de type et de taille permettent d'obtenir cette première répartition. Leaf1 et Leaf2 ne peuvent tenir le rôle de « Composite » car elles ne disposent chacune que d'un élément alors que « Composite » en requiert deux (une instance de PMethod et une instance de PAssoc). L'interface marqueur MyLonelyInterface est supprimée car elle ne contient aucun élément alors que l'acteur « Composite » en requiert au moins un (méthode operation() dans notre cas).				
2	<i>d, e</i>	MyInterface Component1 Component2	MyClass Composite1 Composite2	MyClass, Composite1, Composite2, Leaf1, Leaf2

Aucune réduction n'a pu être effectuée.				
3	<i>f</i>	MyInterface Component1 Component2	Compositel Composite2	MyClass, Compositel, Composite2, Leaf1, Leaf2
La classe <i>MyClass</i> ne peut tenir le rôle de « <i>Composite</i> » car elle n'a pas de lien d'association avec <i>MyInterface</i> . <i>MyClass</i> est donc supprimée pour le rôle « <i>Composite</i> ».				
4	<i>d, e</i>	Component1 Component2	Compositel Composite2	MyClass, Compositel, Composite2, Leaf1, Leaf2
<i>MyClass</i> dans son rôle de « <i>Composite</i> » était le support pour <i>MyInterface</i> dans son rôle de « <i>Component</i> », cette dernière est donc supprimée.				
5	<i>d, e</i>	Component1 Component2	Compositel Composite2	Compositel, Composite2, Leaf1, Leaf2
<i>MyInterface</i> était le support de <i>MyClass</i> dans son rôle de « <i>Leaf</i> », <i>MyClass</i> est donc supprimée pour le rôle « <i>Leaf</i> ».				
6	<i>d, e, f</i>	Component1 Component2	Compositel Composite2	Compositel, Composite2, Leaf1, Leaf2
Aucune autre réduction n'a pu être effectuée.				

En sortie de l'algorithme, conformément à ce qui était attendu, deux instances valides du pattern *Composite* ont été reconnues :

```
* Pattern: Composite: 2 instance(s)
Component: Component1
Composite: Compositel, Composite2
Leaf: Leaf1
---
Component: Component2
Composite: Composite2
Leaf: Leaf2, Compositel
```

4. Limitations de l'approche

Pour vérifier que notre méta-modèle répondait effectivement à nos objectifs, c'est-à-dire : i) décrire un pattern ; ii) générer l'implémentation correspondante ; iii) le détecter dans du code existant, nous avons réalisé une expérimentation portant sur la modélisation de certains patterns du GoF. Au terme de celle-ci, il nous est apparu intéressant d'étendre le méta-modèle et les mécanismes associés dans les trois directions que nous allons illustrer au travers des patterns : *Singleton*, *Facade* et *Composite*.

4.1. Expressivité du méta-modèle

L'intention du pattern *Singleton* est de « garantir qu'une classe n'a qu'une instance et qu'elle fournisse un point d'accès global à cette unique instance » (cf. page 127 du GoF). Il existe plusieurs solutions possibles pour traduire cette intention, dont trois sont référencées dans le GoF (simple, avec sous-classement, et avec registre). La description du pattern *Singleton*, à l'aide de notre méta-modèle,

nécessite de choisir une des solutions possibles, c'est-à-dire de donner une interprétation de son intention. L'intention du pattern est donc « entremêlée » avec sa description : intention et solution sont indiscernables. Pour dissocier ces deux aspects, il est nécessaire de compléter le méta-modèle.

4.2. Application : de la génération à la modification de source

Le pattern *Facade* répond au besoin « d'unifier les interfaces d'un sous-système en une interface commune de plus haut niveau » (cf. page 185 du GoF). Notre méta-modèle rend compte facilement de l'application de ce pattern mais la structure préconisée ne présente que peu d'intérêt, au-delà de l'aspect pédagogique : il s'agit surtout d'un exemple. Pour que cette application ait un réel intérêt technique, il serait nécessaire de compléter le mécanisme de génération par un système de transformation de source, capable de modifier une application existante en l'adaptant aux spécifications du pattern.

4.3. Détection : d'une forme exacte à une forme approchée

L'algorithme de détection a permis de reconnaître les mises en œuvre complètes du pattern *Composite* dans du code source, il s'est donc avéré un bon outil pour aider à la documentation et à la compréhension de ce code. Par contre, cet algorithme est incapable de détecter des implémentations approximatives ou incomplètes de ce pattern. *Composite* est un pattern très populaire et il est fréquent d'en trouver des formes approchées durant l'analyse de bibliothèques. Il apparaît donc critique de reconsidérer notre mécanisme de détection pour lui permettre d'identifier les micro-architectures (ensembles d'entités) dont l'organisation se conforme « presque » à la solution de ce pattern.

5. Conclusion

Notre objectif final est d'outiller l'environnement de développement Visual Age pour Java, afin de lui adjoindre deux assistants logiciels basés sur la mise en œuvre pratique des patterns. Le premier supporte une programmation visuelle à l'aide de patterns (choix, paramétrage, instanciation et génération de code), le second s'intéresse à la correction semi-automatisée des défauts de conception basée sur une application *a posteriori* des patterns.

Pour atteindre cet objectif, nous avons construit un premier méta-modèle permettant de coupler application et détection de patterns. Après une première expérimentation consistant à s'assurer de la possibilité d'utiliser un même méta-

modèle pour cataloguer, appliquer et détecter les patterns du GoF, nous avons été amenés à :

- associer à ce méta-modèle un nouveau méta-modèle complémentaire : PIL (*Pattern Instantiation Language*), dédié exclusivement à la représentation de l'intention des patterns. Il s'agit principalement de séparer l'aspect intentionnel de l'aspect solution ;
- associer au système de génération de code, un système de transformation source-à-source pour améliorer du source existant par l'application de ces patterns [ALB 01a, ALB 01b, ALB 01c] ;
- utiliser la programmation par contraintes avec explication pour résoudre de manière semi-automatisée la détection de formes approximatives d'un pattern, dans un source Java [GUE 01a], à l'aide d'un CSP généré à partir du modèle abstrait d'un pattern [GUE 01b].

Bibliographie

- [ALB 01a] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, N. Jussien; *Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together*; Proceedings of ASE'01, to appear, 2001.
- [ALB 01b] H. Albin-Amiot, Y.-G. Guéhéneuc; *Design Patterns: A Round-Trip*; ECOOP'01, 11th Workshop for PhD Students in Object-Oriented Systems, 2001.
- [ALB 01c] H. Albin-Amiot, Y.-G. Guéhéneuc; *Design Patterns application: Pure-Generative Approach vs. Conservative-Generative Approach*; OOPSLA'2001 Workshop on Generative Programming, 2001.
- [BAU 96] G. Baumgartner, K. Läufer, V.F. Russo; *On the Interaction of Object-Oriented Design Patterns and Programming Languages*; Technical Report CSD-TR-96-020, Purdue University (USA), 1996.
- [BOR 99] I. Borne, N. Revault; *Comparaison d'outils de mise en œuvre de design patterns*; Revue l'Objet « Patrons orientés objet », éditions Hermès vol. 5, no. 2, 1999.
- [BOS 97] J. Bosh; *Design Patterns & Frameworks: On the Issue of Language Support*; LSDPF'97: Workshop in ECOOP'97, 1997.
- [BRO 95] K. Brown; *Design reverse-engineering and automated design pattern detection in Smalltalk*; Technical Journal, 1995.
- [BUD 96] F. J. Budinsky, M. A. Finnie, J. M. Vlisside, P. S. Yu; *Automatic code generation from design patterns*; IBM Systems Journal, vol. 35, no. 2, 1996.
- [CHA 00] C. Chambers, B. Harrison, J. Vlissides; *A Debate on Language and Tool Support for Design Patterns*; POPL'00, pages 277-289, 2000.
- [DES 99] P. Desfray; *Automation of Design Pattern: Concepts, Tools and Practices*; UML'98: Beyond the Notation, Bézivin - Muller (Eds.), Lecture Notes in Computer Science, vol. 1618, Springer-Verlag, 1999.
- [DUC 97] S. Ducasse; *Réification des schémas de conception: une expérience*; LMO'97, pp. 95-110, 1997.
- [EDE 00] Amnon H. Eden, *Precise Specification of Design Patterns and Tool Support in Their Application*; PhD Thesis, Tel Aviv University, 4th Revision May 9, 2000.
- [FLO 97] Gert Florijn, Marco Meijers, Pieter van Winsen; *Tool Support for Object-Oriented Design Patterns*; ECOOP'97, pages 472-495, 1997.
- [GAM 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides; *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley Professional Computing Series, 1995.

- [GUE 01a] Y.-G. Guéhéneuc, H. Albin-Amiot ; *Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-Class Design Defects*; Proceedings of TOOLS USA pp296-305
- [GUE 01b] Y.-G. Guéhéneuc, N. Jussien ; *Quelques Explications Pour Les Patrons*; Proceedings of the 7th JNPC, 2001.
- [JB 97] Sun Microsystems ; *JavaBeans™ API Specifications* ; disponible à <http://www.javasoft.com>, 1997.
- [PAG 96]B.-U. Pagel, M. Winter ; *Towards Pattern-Based Tools* ; EuroLop'96, 1996.
- [PRE 95]W. Pree ; *Design Patterns for Object-Oriented Software Development* ; ACM Press, Addison-Wesley, 1995.
- [RAP 00]P. Rapicault ; *Instanciation et vérification de patterns de conception : un méta-protocole* ; LMO'00, pp. 43-58, 2000.
- [SOU 95]J. Soukup ; *Implementing Patterns* ; Pattern Languages of Program Design, pages 395-412, Coplien - Schmidt (Eds), Addison-Wesley, 1995.
- [SUN 99]G. Sunyé ; *Génération de code à l'aide de patrons de conception* ; LMO'99, pp. 163-178, 1999.
- [SUN 00]G. Sunyé, A. Le Guennec, J.-M. Jézéquel ; *Design Pattern Application in UML* ; ECOOP'00, pages 44-62, 2000.
- [TAT 98]M. Tatsubori, S. Chiba ; *Programming Support of Design Patterns with Compile-time Reflection* ; OOPSLA'98 Reflection Workshop, 1998.
- [WAL 72] D. L. Waltz ; *Generating Semantic Descriptions from Drawing of Scenes with Shadows* ; Technical Report AI-TR-271, MIT Cambridge, 1972.
- [WUY 98] Roel Wuyts ; *Declarative Reasoning about the Structure of Object-Oriented Systems* ; TOOLS-USA'98, 1998.