Design patterns: representation, detection and code generation from a meta-model

Hervé Albin-Amiot ¹, Yann-Gaël Guéhéneuc ²

École des Mines de Nantes ¹
4 rue Alfred Kastler
BP 20 722
44 307 Nantes cedex 3
France
{albin, guehene}@emn.fr

¹ Soft-Maint
4, rue du Château de l’Éraudière
BP 72 438
44 324 Nantes cedex 3
France

² Object Technology International Inc.
2670 Queensview Drive
Ottawa, Ontario, K2B 8K1
Canada

ABSTRACT. Design patterns have been quickly adopted by the “object-oriented” community, in particular with the publication of “Design Pattern: Elements of Reusable Object-Oriented Software” [1]. They offer elegant and reusable solutions to recurring problems of conception. Their use increases productivity and development quality. But these solutions, at the frontier of programming languages and models of conception, suffer from a lack of formalization. Thus, their application remains empirical and is often “manually” performed. This article reviews various techniques of automatic applications of design patterns. Then it presents how a meta-model can be used to obtain a representation of design patterns. Finally, this article describes how this meta-model can automatically generate code from this representation and detect design patterns in programs.

KEYWORDS: design patterns, code generation, detection, Java
1. Introduction

Since their emergence [1, 23], usage of design patterns is now widespread. Their contribution covers definition, design and documentation of class libraries and frameworks. Even if it does not exist any consensus about the way to support their application (using tools, languages… [20]), it seems now that this task should be automated or at least be assisted. Moreover, manual application is often problematic because it is tedious and error prone. Also, as mentioned in [17], traceability remains the essential drawback of this hand (and sometimes hard) coding task. In other words, when a pattern is applied, the resulting implementation does not generally provide a mean to come back to the pattern for which it was required because the pattern code is mixed with the user application code.

The application of a design pattern can be decomposed in three distinct activities: the choice of the right pattern, which fulfils the user requirements, its adaptation to these requirements (the term instantiation is commonly used to identify this task), and finally, the production of the code required for its implementation. The first step cannot be easily automated because human expertise is a primordial factor to accomplish it. However, a valuable help can be proposed to the developer to successfully complete the two remaining steps. Most of the research works on this topic are based on this assumption. Two kinds of works can be distinguished. Those based the representation of design pattern for a given implementation language and those for a given modeling language [19]. In both cases the first issue concerns the necessary extension of an existing base language or the definition of a new language. If the support of the application of patterns is based on an implementation language [10], the traceability between the applied patterns and the resulting code is de facto ensured. The problem of this approach is the use of a non-standard implementation language that limits its portability. For this reason this approach has not been adopted here. The second approach, based on a modeling language, does not have the problem of using a specific implementation language but in most case it does not address either code generation or traceability. In an iterative development, where source code and model are not synchronously manipulated, it’s important to provide a univocal link between them.

In this paper we present a solution based on a modeling language where patterns are described in terms of “entities” and “elements” which are defined in the core of a meta-model dedicated to the representation of patterns. In order to evaluate it more rapidly, it has not been defined as an extension of an existing meta-model (like UML for example). The proposed meta-model experimented in Java and built following the JavaBeans “formalism” (properties, idioms and introspection), provides a way to describe structural and behavioral aspects of design patterns. From this description, the meta-model provides the required machinery to produce code and to detect instantiated patterns from code.

The intended contribution of this approach is the reification of design patterns as first-class modelling entities. Such a reified design pattern can then be used to produce the associated code implementation according to the context of application and can also be used to detect its own occurrence in user code. The way a design pattern is applied or detected is deduced from its declaration not from external specifications or hints. This adds an extra layer of abstraction.

The rest of this paper is organized as follow: section two proposes a taxonomy of known techniques to produce automatically code from patterns. Section three introduces our solution: the meta-model and how it is constituted. Section four illustrates, through the example of the Composite pattern, how the meta-model is used to represent a given pattern, instantiate it, produce associated code and detect it from code. Finally, the fifth section concludes on the usage of such a meta-model and questions this approach.

2. A taxonomy

2.1. The different techniques

The application of patterns can be automated in different ways and can be characterised according to many criteria (tools [8], supported patterns…). However, techniques are generally relevant either to models or to languages. The need to explicit patterns appears both at the design level and at the implementation level. This section is based on this assumption. It draws up list of the known techniques, gives their limitations and the results that can be obtained. Techniques to automate code production from patterns are:
1) Based on the implementation language: we distinguish four different techniques.

- Implementation language based definition: elements needed to instantiate patterns or to make their instantiations explicit are directly added to the common set of features supported by every kind of OO language. For example, OpenJava [14] extends the Java programming language to provide direct support for pattern instantiation. LayOM [10] proposes a layered system with message sends filters. Each pattern is represented with a particular kind of layer. Thus, its usage is easily identifiable. This approach ensures traceability: patterns exist at the source code level. However, it requires the use of a dedicated non-standard programming language.

- Implementation using meta-programming facilities: patterns are defined as non-functional aspects and are expressed like class properties in a class-based language. This implies that a wide range of design patterns is conceived as class descriptions and thus can be described as meta-classes [14]. With this solution, usual mechanisms for reuse are available and patterns associated code is not mixed within user code. However, reflection is not necessarily available in every OO language and this approach is not useable for every kind of pattern. For example, the simple Singleton pattern is easily implemented and reused this way but the Bridge pattern implementation is not so obvious.

These two approaches presented above view patterns like idioms. In fact, they are based on the postulate that a design pattern is essentially the expression of a missing language construct. Even if some patterns can be seen like idioms ([22] proposes a classification and identifies those concerned), we think that this conclusion is not valid for every pattern.

- Source code parameterisation generation consists in giving the general architecture of a design pattern with "holes" to be filled up by the developer. The general architecture may be given using any programming language or with a neutral programming language that can be then transformed into a standard programming language. The main problem of this approach is the lack of traceability between a design pattern and its implementation. The produced source code does not guarantee a backward link to the implemented design pattern. Moreover, the source code generator must be manually parameterised with information (names, actors...) that could be automatically deduced from the state of the application. Finally, the source code produced may be difficult to integrate "as-is"; for example, in [4] the source code is used using "copy" and "paste".

- Source code transformation consists in associating each design patterns with a set of transformation rules. This approach is rather complex but requires a minimum of interaction with the user and produces source code that is directly usable. However, once the transformation done, they are no way to know which design pattern has been applied, thus a lack of traceability. The same technique can also be used at the design level to transform user model [5].

2) Technique based on the modelling language or formalism using the meta-model technique. This approach consists in defining a meta-entity that the design pattern is obtained from through an instantiation (pseudo-) link. Since design patterns are way too informal to be model "as-is", they are modelled according to different aspects, depending on their purposes (application, validation [2], structural representation, composition...). Most of the works realised on design pattern application are based on the definition of a meta-model. Among other are the use of fragments [5], the use of a pattern [3] (the meta-model is a "meta-pattern") or a meta-model derived from a meta-pattern [9]. The use of a meta-model ensures that a pattern exists as a first-class entity. However, the number of aspects supported by a single meta-model is reduced (validation and application, structural representation [5], validation and conception [9]) and requires theoretically the use of several meta-models.

None of the aforementioned techniques offers a complete support to the different uses of design patterns. The reason is the intrinsic informal nature of design patterns and the several ways they can be used. The choice of one of those techniques depends only on the reasons why design patterns are being used and the accepted trade-offs.

### 2.2. The chosen technique

In our work, we single out several essential properties our solution must possess. First, the application of a design pattern must be independent of the programming language used for the implementation, this excludes the reification at the language level and the use of a non-standard programming language. Second, the programming language being used does not necessarily supports behavioural reflection, the use of meta-programming is not a solution generic enough. Last, the design patterns must be reified throughout the different stages of the development process. The approaches without reification such as code generation and transformation are thus excluded. The approach base on a meta-model is the only approach that satisfies the
criteria mentioned. However, it does not guarantee traceability per-se. But it is possible to propose a mechanism of detection that would solve the traceability problem.

Our solution consists in defining a meta-model for design pattern instantiation, source code production and detection.

3. Presentation of the meta-model using a simple example

Throughout this section, we will focus on the Composite pattern. The Composite pattern comprises "objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of object uniformly" ([1] page 163). We follow the Composite pattern presented in the "Implementation" section of [1], in paragraph "declaring the child management operations". This is a common use of the Composite pattern (for instance, see classes Component and Container of the Java AWT\(^1\)). The Composite pattern is a relatively simple structural pattern and is often used as example [19] either for code production [2] or for detection [11, 12]. Despite its simplicity, this design pattern includes all the common constituents found in the structural patterns of [1]: classes, interfaces, methods and fields; inheritance, association and delegation relationships. Moreover, its purpose and usage are unambiguous, whereas other patterns (for example, Facade or Interpreter) may present ambiguities in the ways they are used and implemented.

3.1. The meta-model in a nutshell

It exists several meta-models for representing design patterns but none is specifically designed towards code generation or detection. This is the main contribution of this article: a meta-model to handle uniformly instantiation and detection of design patterns. [9] introduces meta-models for design patterns instantiation and validation but without support for code generation. In the tool PatternGen developed by G. Suyène, the meta-model does not support source code production (another module handles the production of code) and it offers no patterns detection. In [5], the fragments-based system, as it is, allows only representation and composition of design patterns.

The meta-model embodies a set of entities and the interaction rules between them. All the entities needed to describe the structure and behaviour of the structural design patterns introduced in [1] are present. Figure 1 shows a simplified version of the meta-model.

![Figure 1: Simplified UML class-diagram of the meta-model](image)

A model of pattern is reified as an instance of a subclass of class Pattern. It consists in a collection of entities (instances of PEntity), representing the notion of participants as defined in [1]. Each entity contains a collection of elements (instances of PElement), representing the different relationships between entities. If needed, new entities or elements can be added by subclassing the PEntity or PElement classes.

The meta-model defines in details the semantics of a pattern. A pattern is composed of one or more classes or interfaces, instances of PClass and PInterface and subclasses of PEntity. An instance of PEntity contains methods and fields, instances of PMethod and PField. The association and delegation relationships are expressed as elements of PElement. An association (class PAssoc) belongs to PEntity and references another

---

\(^1\) Abstract Window Toolkit
This relationship is simple since all the associations used in [1] are binary and mono-directional. For example, an association that links a class B to a class A is defined using two instances of class PClass, A and B, and one instance of class PAssoc. The instance of class PAssoc belongs to A and references B. Delegation is expressed in a similar way using the class PDelegatingMethod. For example, the delegation of the behaviour of a method foo of A to a method bar of B is realised using an instance of class PDelegatingMethod. The instance of class PDelegatingMethod belongs to A and references the method bar of B. The PDelegatingMethod object also references the association between A and B to deduce from it the cardinality of the message send: simple or "multicast".

Moreover, the meta-model makes extensive use of the JavaBeans [21] formalism. This formalism describes a set of idioms and mechanism. The JavaBeans introduce a mechanism of listener. A listener is an object associated to a property. When the property changes, every listener attached to this property is notified and may either accept the modification or reject it. If it rejects the modification, the property is not modified. This mechanism of listener (and veto-listener) is used to enforce integrity checks while creating a model. Thus, errors can be detected early in the definition process, before instantiation and detection. For example, if a PClass declared as concrete is asked to become abstract, it notifies all its elements of this change. The PClass is changed if and only if all its elements have accepted the modification.

The idioms are naming conventions for methods and fields. For instance, sets of methods with get and set prefixes imply simple read / write properties. Sets of methods with add and remove prefixes2 imply list properties. Fields are always declared private. These naming conventions are used for three different purposes:

- Declaration: associated with introspection, they are used to discover pattern-related operations, such as addLeaf for the Composite pattern, or addState for the State pattern.
- Application: the code generated for a given pattern complies with these conventions. For example, all the fields in the source code of a generated pattern are declared private.
- Detection: the pattern detection mechanism is based on these naming conventions. For example, if a set [private field, add, remove methods modifying the private field] is detected in a class, it implies an association relationship between this class and the type of the private field.

3.2. Instantiation of the Composite pattern

We now further present the meta-model and its usage through the example of the Composite pattern. The figure below presents the general procedure to instantiate a pattern.

![Diagram of the Composite pattern instantiation process](image)

1. The first step consists in specialising the meta-model to obtain a meta-model with all the structural and behavioural needed constituents. In the case of the Composite pattern, the meta-model previously presented is sufficient. However, it would be necessary to add a new PEntity, called ImmutablePClass, to implement a Composite pattern which leaves are immutable. (In this case, a new subclass ImmutablePClass to the class PClass would be added.)

2. The second step consists in the instantiation of the meta-model. The meta-model of the Composite pattern is instantiated into an abstract model. This abstract model corresponds to a reification of the Composite pattern and holds all the needed information related to the pattern. Thus, the Composite pattern takes reality and

---

2 Not to be confounded with the add... and remove... methods associated with the JavaBeans listeners.

---

5/12
becomes a first-class object. This differs from [3, 10, 12] and is an important gain of our approach. Since abstract models of design patterns are first-class objects, it is possible to reason with them and to use them as normal objects. Thus it is possible to introspect them and modify their structure and behaviour both statically and dynamically.

Figure 3 (left) shows the structure of the Composite pattern as in [1]. From the structure and the notes, we obtain a new class diagram where all informal indications are explicit. This class-diagram, shown figure 3 (right), is the abstract model of the Composite pattern.

![Composite Pattern Diagram]

**Figure 3:** Structures of the Composite patterns as in [1] and of its related abstract model

This abstract model is expressed in a declarative manner into the class Composite. The table below shows the declaration of the Composite pattern. This declaration is obtained from the class-diagram figure 3 (right). Such a declarative description of design patterns is to be related to the work on *tricks* by Eden, Yehudai and Gil [5]. With the purpose of automating design patterns application in mind, they define the notion of tricks, language independent constructs that can be seen as links between programming language idioms and design patterns. The application of a design pattern is then defined as a set of tricks to be successively applied on the classes that must conform to the design pattern. In our approach, a design pattern is described using low-level design constructs (such as association, delegation...) that are related to tricks. However, a design pattern is described as such (according to the definition in [1]), not relatively to the way it will be applied (code generation, source transformation [5]...). The way a pattern is applied is deduced from its declaration. This adds an extra layer of abstraction and allows a same declaration to be used for both application and detection.

<table>
<thead>
<tr>
<th>Definition of the abstract model of Composite pattern</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Composite extends Pattern {</td>
<td>Declaration of the abstract model. This takes place in the constructor of class Composite, subclass of Pattern.</td>
</tr>
<tr>
<td>Composite(...) {</td>
<td></td>
</tr>
<tr>
<td>iComponent = new PInterface(&quot;Component&quot;)</td>
<td>Declaration of interface Component</td>
</tr>
<tr>
<td>mOperation = new PMethod(&quot;operation&quot;)</td>
<td>Declaration of method operation, defined into interface Component</td>
</tr>
<tr>
<td>iComponent.addPElement(mOperation)</td>
<td>The interface Component is added to the pattern definition</td>
</tr>
<tr>
<td>this.addPEntity(iComponent)</td>
<td>Association targeting Component with cardinality 1+</td>
</tr>
<tr>
<td>anAssoc = new PAssoc(&quot;children&quot;, iComponent, 2)</td>
<td>Declaration of class Component</td>
</tr>
<tr>
<td>cComposite = new PClass(&quot;Composite&quot;)</td>
<td>Class Composite implements interface Component</td>
</tr>
<tr>
<td>cComposite.addShouldImplement(iComponent)</td>
<td>Association « children » links Composite and Component</td>
</tr>
<tr>
<td>cComposite.addPElement(anAssoc)</td>
<td>The method operation defined into class Composite implements the method operation of interface Component and is linked to this interface through the association anAssoc</td>
</tr>
<tr>
<td>aPDelegatingMethod = new PDelegatingMethod(</td>
<td>Class Composite is added to the pattern</td>
</tr>
<tr>
<td>&quot;operation&quot;, anAssoc)</td>
<td>Declaration of class Leaf</td>
</tr>
<tr>
<td>aPDelegatingMethod.attachTo(mOperation)</td>
<td>Class Leaf implements interface Component</td>
</tr>
<tr>
<td>cComposite.addPElement(aPDelegatingMethod)</td>
<td>The public interface of class Leaf is automatically generated (creation of a method operation)</td>
</tr>
<tr>
<td>this.addPEntity(cComposite)</td>
<td>Declaration of other services that will be discovered using introspection...</td>
</tr>
<tr>
<td>cLeaf = new PClass(&quot;Leaf&quot;)</td>
<td></td>
</tr>
<tr>
<td>cLeaf.addShouldImplement(iComponent)</td>
<td></td>
</tr>
<tr>
<td>cLeaf.assumeAllInterfaces()</td>
<td></td>
</tr>
<tr>
<td>this.addPEntity(cLeaf)</td>
<td></td>
</tr>
</tbody>
</table>
void addLeaf(String leafName) {
    PClass newPClass = new PClass(leafName)
    newPClass.addShouldImplement(
        (IInterface) getActor("Component")
    )
    newPClass.assumeAllInterfaces()
    newPClass.setName(leafName)
    this.addPEntity(newPClass)
}  

For example, the service addLeaf, specific to the Composite pattern

Implementation of methods getName, getIntent and getApplicability

Abstract models are stored inside a pattern repository (class PatternRepository, figure 1). This repository helps to access defined design patterns and to assess the relevance of the solution they represent.

The third step consists in the instantiation of the abstract model into a concrete model. The concrete model represents the pattern applied to fit within a given application. The concrete model used has been introduced in [1] and used in [9]. It defines a hierarchy of graphical components as shown figure 4.

![UML diagram of the concrete model](image)

Figure 4: UML diagram of the concrete model

The instantiation of the abstract model is realised by instantiating the class Composite defined in ⑦. Then, each participants of the pattern is named to match the concrete application (figure 4). An alternative (⑧) to this parameterisation is to clone an abstract or a concrete model to obtain a new concrete model. The advantage of such an operation is to allow fundamental modifications of the model, such as modifications that shortcuts the normal behaviour of the model, while ensuring the integrity of the initial model.

(The following source code is given as an example since it can be deduced from the pattern abstract model and the context. A visual manipulation tool (such as PatternBox\(^7\) not further described therein) would dynamically generate it using the user inputs.)

<table>
<thead>
<tr>
<th>Declaration of the concrete model</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite pComposite = new Composite()</td>
<td>Declaration of a new concrete model</td>
</tr>
<tr>
<td>pComposite.getActor(&quot;Component&quot;).setName(&quot;Graphic&quot;)</td>
<td></td>
</tr>
<tr>
<td>pComposite.getActor(&quot;Component&quot;) .getActor(&quot;operation&quot;).setName(&quot;draw&quot;)</td>
<td>Definitions of the actors' names</td>
</tr>
<tr>
<td>pComposite.getActor(&quot;Leaf&quot;).setName(&quot;Text&quot;)</td>
<td></td>
</tr>
<tr>
<td>pComposite.getActor(&quot;Composite&quot;).setName(&quot;Picture&quot;)</td>
<td></td>
</tr>
<tr>
<td>pComposite.addLeaf(&quot;Line&quot;)</td>
<td>New leaves are added to the concrete model</td>
</tr>
<tr>
<td>pComposite.addLeaf(&quot;Rectangle&quot;)</td>
<td></td>
</tr>
</tbody>
</table>

⑦ The final step consists in code generation. The meta-model defined a visitor pattern used to generate code. So far, generators for Java and Claire [16] are implemented. The Java source code obtained from the concrete model class-diagram figure 4 is presented below.

---

\(^7\) Available at [http://www.emn.fr/albin](http://www.emn.fr/albin)
3.3. Detection of the Composite pattern

The detection system is designed to work on code produced by instantiation of abstract models and on design pattern implemented "by hand". It does not use any marking system and does not require any detection-specific information to be present in the code. The detection is based principally on structural information but can be extended to include other information. For example, in the case structural information is not enough, the algorithm may be specialised to use a marking system. The detection is based on a repository of all the constituents of a pattern. This repository, instance of class TypesRepository (figure 1) contains all the PEntities and PEElements currently defined.

The three steps are:

① The class PatternIntrospector, in charged of the detection, submits all the syntactic elements (classes, interfaces, methods...] found in the given user's code to the types in the TypesRepository. The PatternIntrospector builds a concrete model that represents the given code using the constituents defined into the meta-model. The following code is an extract from a user's code:

```java
/* Graphic.java */
public interface Graphic {
    public abstract void draw();
}

/* Picture.java */
public class Picture implements Graphic {
    // Association: children
    private Vector children = new Vector();
    public void addGraphic(Graphic aGraphic) {
        children.addElement(aGraphic);
    }
    public void removeGraphic(Graphic aGraphic) {
        children.removeElement(aGraphic);
    }
    public void draw() {
        // Method linked to: children
        for(Enumeration enum = children.elements();
            enum.hasMoreElements();
        ) {
            ((Graphic)enum.nextElement()).draw();
        }
    }
}

/* Text.java */
public class Text implements Graphic {
    public void draw(){}
}

/* MyClass.java */
public class MyClass {
    public void dummy() {
    }
}

/* Leaf1.java */
public class Leaf1 extends Component1 {
    public void dummy() {
    }
}
```

② The class PatternIntrospector, in charged of the detection, submits all the syntactic elements (classes, interfaces, methods...) found in the given user's code to the types in the TypesRepository. The PatternIntrospector builds a concrete model that represents the given code using the constituents defined into the meta-model. The following code is an extract from a user's code:

```java
/* Component1.java */
public interface Component1 {
    public void dummy();
}

/* Leaf1.java */
public class Leaf1 extends Component1 {
    public void dummy() {
        }
}

/* MyClass.java */
public class MyClass {
    public void dummy() {
    }
}
```
The PatternIntrospector transforms it into the following concrete model:

Figure 6: Simplified UML class diagram of the given user code

Each abstract model (found into the PatternsRepository, figure 1) takes this concrete model as input and analyses it. Each constituent of the abstract model (entities and elements) gets the concrete model as input through its recognize method. It detects which constituents of the concrete model match it. Then it passes the remaining not recognised constituents on to the following abstract model constituents. The recognizeRequestOrder method defines the order in which abstract model constituents are called. This mechanism allows distinguishing a set {get, set, private field} from an association.

The general algorithm is presented into the next table:

<table>
<thead>
<tr>
<th>Detection algorithm, class PatternIntrospector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let C: list of the user's classes</td>
</tr>
<tr>
<td>Let P: pattern being recognised</td>
</tr>
<tr>
<td>Let E: list of the entities (PEntity) ordered following recognizeRequestOrder</td>
</tr>
<tr>
<td>Let L: list of the elements (PElement) ordered following recognizeRequestOrder</td>
</tr>
<tr>
<td>Let s: list of the syntactic elements (classes, interfaces...) found by introspection</td>
</tr>
<tr>
<td>Let T: list of instanced of PElement</td>
</tr>
<tr>
<td>Let U: class being examined</td>
</tr>
<tr>
<td>S = C</td>
</tr>
<tr>
<td>T = Ø</td>
</tr>
<tr>
<td>For each e of E, while size(S) &gt; 0</td>
</tr>
<tr>
<td>S = e.recognize(S, P)</td>
</tr>
<tr>
<td>T = P.listPEntities()</td>
</tr>
<tr>
<td>For each t of T</td>
</tr>
<tr>
<td>U = Class.forName(t.getName())</td>
</tr>
<tr>
<td>S = U.getDeclaredConstructors() + U.getDeclaredMethods() + U.getDeclaredFields()</td>
</tr>
<tr>
<td>For each 1 of L, while size(S) &gt; 0</td>
</tr>
<tr>
<td>S = l.recognize(S, P)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method recognize of PEntity(e.recognize(S,P))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let N: list of non-recognised entities</td>
</tr>
<tr>
<td>N = S</td>
</tr>
<tr>
<td>For each s of S</td>
</tr>
<tr>
<td>If s = e Then</td>
</tr>
<tr>
<td>P.addPElement(new(s))</td>
</tr>
<tr>
<td>N = N - {s}</td>
</tr>
<tr>
<td>Return N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method recognize of PElement(l.recognize(S,P))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let N: list of non-recognised elements</td>
</tr>
<tr>
<td>N = S</td>
</tr>
<tr>
<td>For each s de S</td>
</tr>
<tr>
<td>If s = l Then</td>
</tr>
<tr>
<td>P.getActor(s.getDeclaringClass()).getName().addPElement(new(s))</td>
</tr>
<tr>
<td>N = N - {s}</td>
</tr>
<tr>
<td>Return N</td>
</tr>
</tbody>
</table>

Then, the abstract model examines and determines which entities (instances of PEntity) of the submitted model can be associated to its different roles. The following criteria are applied:

1. A role of the abstract model must be fulfilled by a constituent of same type in the model built from the user's code.
2. The model built from the user's code must contain (at least) as many entities as the abstract model.
3. For each role attributed from the abstract model, the corresponding entities in the model being built must contain (at least) as many elements as the entity from the abstract model.
4. Inheritance links.
5. Realisation links.
6. Association and aggregation links.

In our example, the concrete model shown figure 6 is analysed by the Composite pattern abstract model, which finds the following (partial) results after having applied the three first criteria:
Once the three first criteria have been applied to reduce the search space, the algorithm verifies for each remaining criterion (binary constraint) which entity (value), for a given role (variable), has a supporting entity verifying this constraint. This is similar to the arc-consistence problem in CSP. In our case, if we obtain the arc-consistence, one valid instance of the pattern represented by the abstract model used for examination has been detected.

The following example applies on the detection of a variant of the composite pattern as shown figure 6. The following table summarizes the different stages of the detection:

<table>
<thead>
<tr>
<th>Criteria</th>
<th>« Component »</th>
<th>« Composite »</th>
<th>« Leaf »</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &amp; 2 &amp; 3</td>
<td>MyInterface Component1 Component2</td>
<td>MyClass Composite1 Composite2</td>
<td>MyClass Composite1 Composite2 Leaf1 Leaf2</td>
<td>The constraints of type and size reduce the search space. Classes Leaf1 and Leaf2 cannot play the role of « Composite » since they have only on element each when « Composite » requires two (an instance of PMethod and an instance of PAssoc). The marker interface MyLonelyInterface is removed since it does contain no element when « Component » requires at least one (an operation method).</td>
</tr>
<tr>
<td>2 &amp; 3</td>
<td>MyInterface Component1 Component2</td>
<td>MyClass Composite1 Composite2</td>
<td>MyClass Composite1 Composite2 Leaf1 Leaf2</td>
<td>No reduction can be performed.</td>
</tr>
<tr>
<td>3</td>
<td>MyInterface Component1 Component2</td>
<td>Composite1 Composite2</td>
<td>MyClass Composite1 Composite2 Leaf1 Leaf2</td>
<td>Class MyClass cannot play the role of « Composite » since it does not have an association link with MyInterface, MyClass is removed for the role of « Composite »</td>
</tr>
<tr>
<td>4 &amp; 5</td>
<td>Component1 Component2</td>
<td>Composite1 Composite2</td>
<td>MyClass Composite1 Composite2 Leaf1 Leaf2</td>
<td>MyClass as « Composite » was the support of MyInterface for the role of « Component ». MyInterface is therefore removed.</td>
</tr>
<tr>
<td>5 &amp; 6</td>
<td>Component1 Component2</td>
<td>Composite1 Composite2</td>
<td>Composite1 Composite2 Leaf1 Leaf2</td>
<td>MyInterface was the support of MyClass for the role of « Leaf », so MyClass is removed for the role of « Leaf »</td>
</tr>
<tr>
<td>6 &amp; 6</td>
<td>Component1 Component2</td>
<td>Composite1 Composite2</td>
<td>Composite1 Composite2 Leaf1 Leaf2</td>
<td>No other reduction can be performed.</td>
</tr>
</tbody>
</table>

After the computations, the algorithm found two instances of the Composite pattern, as expected:

* Pattern: Composite: 2 instance(s)
  Component: Component1
  Composite: Component1, Composite2
  Leaf: Leaf1
  ...
  Component: Component2
  Composite: Composite2
  Leaf: Leaf2, Component1
4. Limitations and future

4.1. Instantiation

The main limitation of this approach is the lack of behavioural aspects. Theoretically, behavioural aspects are easy to integrate into the meta-model. However, experience proves that this may be difficult using structural elements. Another problem concerns the integration of the generated code within the user code. This integration necessitates modelling the user application. This model could be realised using a programming language but we do not want to use a specific programming language since it would be non-standard. A solution would be to transform the wanted implementation to fit the user code. Instantiation would include the generation of the strict implementation of the pattern, and then this implementation would be integrated into the user code using source code transformation. We are currently investigating the definition of an engine able to automatically transform user source code according to a pattern declaration.

4.2. Detection

The second main limitation of this approach involves the detection of behavioural element in addition to structure elements. This is a related problem to the one defined in the previous sub-section: we want to define a pattern specification language, which formalised instantiation, generation and detection, therefore we need both structural and behavioural descriptive elements.

Another problem of this approach concerns the integration of the user's code specificity into the detection mechanism. How to distinguish variants of a design pattern? The current approach (based on a constraint programming AC*-like algorithm) should be able to make such distinctions. However, we experienced limitations with such a strict constraint system. The solution we are currently investigating is based on a constraint system with dynamic constraint relaxation. From a strict design pattern declaration and according to some priorities, the constraint system is able to automatically relax unresolved constraints to find solutions.

5. Conclusion

As stated in sections 1, several approaches have been proposed to instantiate and detect design patterns. Those approaches are either at the implementation level or at the design level.

At the implementation level, design patterns are represented through syntactic constructions. Therefore, instantiation and detection are straightforward. The traceability is de facto ensured. However, this approach requires the use of a dedicated implementation language, which may be too much a constraint.

At the design level, design patterns are represented through high-level constructs (such as classes, methods...) independently from a specific implementation language. However, instantiation (code generation) is not always supported and there is no insurance for a 1 to 1 match between design constituents and implementation constructs (a unique class of a pattern, at design level, may be spread out into several classes at the implementation level). Thus, there is a need for mechanism ensuring the traceability.

The meta-model described here offers a uniform way to define patterns at the design level. The structure and properties of a design pattern are defined using the constituents offered by the meta-model. The same abstract model obtained can then be used for both instantiation and detection. There is no dissemination of design pattern-related information over the design or the implementation. An abstract model contains all the information related to its instantiation and detection, thus ensuring its traceability.

However, the techniques we have presented in these pages seem to be valuable essentially for structure-based patterns. The main reason is that each meta-entity needs to have code equivalence and ability to detect in code its corresponding syntactic construct. To reach this goal, each description must be structure-based in order to generate code (generation always provides structure and not behaviour) and prevent us from an uncertain dynamical source analysis. The problem of describing behaviour using structural elements can be addressed using a strict separation between patterns representation and the code producer / detector system using two meta-models. A first one could be used to instantiate pattern using structural and behavioural bricks without code

---

Arc consistency
equivalence and a second one would provide bricks to produce and detect architecture parcels. Moreover, this separation would allow us to represent patterns using formalisms such as contracts [25] or constraints applied on role and actor, even if this formalism does not retain sufficient information to produce code. The main drawback of this hypothetical solution is its complexity: it requires definition of several interacting meta-models.

6. Bibliography

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides ; Design Patterns : Elements of Reusable Object-Oriented Software ; Addison-Wesley Professional Computing Series, 1995.
[16] Yves Caseau, François Laburthe ; CLAIRE: Combining Objects and Rules for Problem Solving ; JICSLP96 workshop on multi-paradigm logic programming, 1996.
[21] Sun Microsystems ; JavaBeans™ API Specifications ;