Formalizing the Decoupling Constraints of Design Patterns

Mikal Ziane, Gilles Ardourel, Marianne Huchard, Salima Chantit

Formalizing the Decoupling Constraints of Design Patterns

Mikal Ziane[* and ***], Gilles Ardourel[**], Marianne Huchard[**], Salima Chantit[*]

[*] Laboratoire d'Informatique de l'Université Paris 6 (LIP6)
Pole IA 8, rue du Capitaine Scott 75015 Paris, France
Prenom.Nom@lip6.fr

[**] Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM)
161 rue Ada 34392 Montpellier Cedex 5, France
nom@lirmm.fr

[***] Université René Descartes, Paris, France

Abstract. In this paper we adapt the Access Graph notation of Ardourel and Huchard for static access rights to express decoupling constraints. These constraints are pervasive in design patterns albeit in a very informal and often not very explicit way. This new formalism will be used in the RNTL LUTIN project to detect design problems and propose solutions using program or model transformations.

1. Introduction

Design patterns have been quite successful in disseminating design experience, using a "form that people can use effectively" [1]. A pattern describes a recurring problem and the core of a solution to that problem. While many researchers have proposed to formalize the solutions of patterns, **formalizing their problems has received very little attention** [2].

This lack of attention is probably due to the fact that the problems of design patterns are defined in rather vague terms which sometimes leads to debates among design-pattern experts themselves. Indeed, patterns were not meant to be formalized, they "are to be executed by architects with insight, taste, experience, and a sense of aesthetics" [3].

As a consequence, tool support for design patterns is quite superficial. If many tools can reproduce stereotyped solutions when a specific pattern has been chosen, none is able to

- identify design problems depending on the quality objectives of the designer,
- and propose a solution to solve this problem.

With current tools, object-oriented designers have to realize that they are facing a problem, and must then find a proper pattern to solve it, if such a pattern exists.

Unfortunately, it was predicted that "the number of design patterns will grow to a level, where it becomes impossible to maintain an impression of which design patterns exist, let alone to know what problems these design patterns actually solve" [4]. In fact, Gamma et al. already acknowledged that "With more than 20 design patterns in the catalog to choose from, it might be hard to find the one that addresses a particular design problem" [5]. Today, several hundred patterns have already been published [6]. Consequently designers will have more and more problems to find patterns matching their design problems and useful patterns might be overlooked. On the other hand, over-engineering is also a common attitude. Some designers apply patterns to situations in which these patterns are not useful and in fact add unnecessary complexity [7].

Tool support to match problems with patterns is thus required. Unfortunately, if automatic detection of object-oriented design problems has been tried with promising results [8], very little work has been done to try and match problems and patterns.

Gustavsson and Ersson proposed to reorganize the Intent section of patterns in [1] so that tools could more easily find needed patterns. While this preliminary work is an interesting testimony of the need of such tools, the proposed reorganization is still extremely informal.
Guéhéneuc and Amiot did try to detect and correct design defects using patterns [9]. However, their work relies on the rather surprising assumption that "design patterns represent good architectures and that pieces of code similar to design patterns represent potential places for improvements". As we said before, design patterns are needed in some situations and add useless complexity in other situations. Moreover, it is not clear why the initial design where a pattern would be useful would necessarily be similar to the situation where the pattern has been applied.
In [10], two of us proposed to **define the problem of design patterns as a quality constraint**. Such a constraint defines which designs are desirable and which are not. We focused on a very common kind of quality objective in design patterns: requiring that two pieces of software are decoupled or loosely coupled so that they can evolve independently.
Like encapsulation, decoupling constraints are essential to lower the cost of some maintenance tasks by preventing many changes to propagate over the whole design.
More precisely, in [10] two of us proposed to **define the decoupling objectives pervasive in design patterns as static access right constraints**. This proposal however was only sketched and in this paper we make it more precise. In order to do this we adapt the formalism which the other two of us proposed in [11].
Assuming that object-oriented designers have declared the decoupling constraints they need, the automatic detection of common design problems becomes possible: a problem occurs when an access violates a declared constraint. Program transformations can then reproduce the solutions of design patterns and only the solutions which match the declared decoupling constraints will be proposed to the designer to choose from [10].
This paper is organized as follows. Section 2 introduces the new formalism and stresses the differences with the Access Graph notation of [11]. It also proposes complementary notations to make the formalism easier to use. Section 3 then shows a few examples of using the notation to detect design problems and filter out the solutions of a transformational engine as explained with more details in [10].

2. Making decoupling constraints explicit

Patterns aim at enforcing some quality criteria. The problem that a pattern solves thus occurs when these criteria are not met. In order to detect when these criteria are not met and in order to filter out the solutions produced by program or model transformation these criteria must be expressed unambiguously [10].
Thavildari and Kontogiannis annotate patterns and transformations with soft-goal contributions or requirements [12]. For instance the ABSTRACTION transformation is labeled High Control Flow Consistency (+), High Cohesion (++), High Data Consistency (++) and Low I/O Complexity (-). Patterns are labeled similarly so that transformations that may help solve a pattern's problem can be identified.
While this can be useful to identify candidate transformations this is not enough to detect which parts of a design are problematic and to filter out the candidate solutions that do not really solve the problem. A quality constraint distinguishes between "good" and "bad" designs.

Since many GOF patterns aim at decoupling two pieces of software so that they can evolve separately we have focused on expressing decoupling. **We have thus defined a *hidden* meta-predicate** which restricts the use of one or more names in one or more namespaces.
In other words, this meta-predicate aims at specifying static access control constraints, independently from a specific object-oriented programming or modeling language. In order to define this predicate, we adapt the definitions of Ardourel and Huchard for static access control [11]. The next two sections detail the Access Graph notation for access control, then the hidden meta-predicate.
The way to express *static access control* or *visibility* varies from one object-oriented programming language to the other. Eiffel relies on names of clients (classes that access a property) in access control expressions, where Java and C++ rely on special sets of anonymous clients for a given class *C*, such as all classes (*public*), subclasses of *C* (*protected*) or classes of the same package (default protection in Java). UML also distinguishes between public and private visibilities (among others) but does not define them precisely.
The decoupling constraints of design patterns cannot always be expressed precisely using these programming or modeling languages: they were not designed for this. A notation is then required to unambiguously express these constraints.
In previous work, two of us introduced the Access Graph formalism [11] to support language-independent reasoning on access control in modeling and programming languages. The formalism has clear semantics and is conceived to support designers' intuition or software analysis [15] with its graphical counterpart. An extension of UML has recently been studied [16]. In this paper we present an adaptation as well as a generalization of the formalism in [11] to support decoupling constraints.
We define accesses to names as tuples *(NS1,NS2,n)* where:

- *NS1* and *NS2* are namespaces (classes, packages, methods, etc.),
- *n* is the name of a set of local properties (including the type for attributes, and the signatures for methods) in namespace *NS2*. When the set is a singleton, we use the property's qualified name.

The reason for considering names of sets of properties and not only names of properties is the fact that in some cases all the versions of the same polymorphic function must be treated as a whole.
The new formalism, contrary to [11] does not distinguish among different kinds of accesses (read, write, call…). What is relevant here is the fact that if n is changed or removed in NS2 then NS1 has to be changed too, which increases the cost of maintenance.
Like in [11] we only consider **static** accesses. More precisely, if *NS2* is a class and *n* the name of an instance variable or method, the access is an access via an object of static type *NS2*. For example, when we consider an access to an instance variable *v*, using an access expression *o.v*, we are only concerned with the static type of object *o*, not its dynamic type. An access using an expression *o.m()*, where *m* is an instance method, also considers the static type of *o*. It can be authorized if *o* has for static type a class *C*, but prohibited if *o* has for static type a subclass of *C*, independently of the invoked method (code): it can be the same in the two cases (inherited in the subclass), or *m* can be abstract in *C* and implemented in the subclass. If a class B has a subclass C, then every method of B and C are local properties, whether they are defined, inherited or redefined.

Tuples *(NS1,NS2,n)* are represented in **diagrams** by an edge with origin *NS1*, target *NS2*, and label *n*. A prohibited access is noted by a crossed arrow.
We also need a **textual** notation to specify which accesses are prohibited. Let AA be the set of allowed accesses for a given program (or model) according to the rules of the program or model's language. What we want to do is to let designers define a subset CAA (constrained allowed accesses) of AA depending on their quality objectives (related to decoupling). Hence we note ***hidden(NS1, NS2, n)*** to declare that access (NS1,NS2,n) is not in CAA (whether or not it is in AA).
Let us now summarize what are the differences between the new formalism and the Access Graph (AG) notation of [11]. The new formalism, like the textual counterpart of the AG notation is a first-order language based on classical set theory. In the new formalism, namespaces and not only classes have properties and bear accesses. So while we keep most of the definitions of [11] (such as the sets of classes and the set of properties) we additionally assume the existence of the set N of namespaces. The map properties is extended to N. As explained above the AccessKinds of AG are nor relevant here and are thus ignored. The distinction in AG between instance-level and class-level accesses is not relevant here.
An *atomic decoupling constraint* if of the form hidden(NS1, NS2, n) where (NS1, NS2, n) is a well-formed access. It is of course possible to express more complex decoupling constraints using first-order formulas. A few examples are given in the next section.
We do not expect that object-oriented designers directly express complex decoupling constraint using a low level logical language with quantifiers. Higher-level definitions will have to be introduced to express common needs easily. The HiddenSubclasses predicate in an example of such definition. We expect the lower-level notation to be used by a limited set of software-quality experts or by tool providers to introduce new high-level definitions. The concrete syntax of our formalism could be adapted to be integrated in an extension of OCL, the constraint language of UML, for example.
The **graphical representation** was also used and was in fact central in the Access Graph notation because new constraints where not expected to be defined very often (the focus was on expressing the access policies or mechanisms of programming languages). In the present formalism the textual notation is probably more adapted to express designers' needs. On the other hand the graphical notation could also be useful to quickly see the impact of a constraint. A few examples of this graphical notation are given below.

The hiddenSubclasses predicate

If a designer does not want class 'Client' to (directly) access the properties of the subclasses of the 'Product' abstract class, nor the type names defined in these subclasses, he or she may declare the following constraint, (see Constraint 1). This could be expressed more easily by hiddenSubclasses('Client','Product') by properly defining the hiddenSubclasses predicate. Similarly a unary version of HiddenSubclasses could be introduced to forbid access to all classes: hiddenSubclasses('Product'). The hiddenSubclasses should however be defined carefully so that a subclass of 'Product' is not denied access to its own properties (including the inherited ones).

```
for all SC in subclassesOf ('Product')
for all n in namesOf(SC)
        hidden ('Client', SC, n)
where
'Client' and 'Product' are classes
subclassesOf (X) is the set of subclasses of class X
namesOf (X) is the set of names of sets of properties owned by class X including the name of X itself
```

Constraint 1 : class 'Client' must not use the names owned by the subclasses of 'Product'

3. Detecting problems and filtering out undesirable solutions

Once quality constraints have been written by the designer (or by a senior designer in charge of quality), a tool can try and detect constraint violations in the current program or model (UML interaction diagrams include method calls). This is what a compiler does with encapsulation-related constraints. A transformational engine can then use the forbidden access occurrences as a target and propose solutions to the designer[10, 13]. This is a significant improvement over previous approaches where designers had to be aware that they had a design problem and had to know which pattern could solve it.

The Prototype pattern

Consider the C++ code of Code example 1, where Circle and Triangle are subclasses of the Graphic abstract class and assume that a constraint, e.g. hiddenSub-classes('Graphic'), forbids the calls to new Triangle and new Circle. See also Figure 1 which highlights forbidden accesses. On this figure the edges are not labeled to denote that all the names owned by the subclasses are hidden.

```
Graphic* GraphicTool::createObject(Icon* anIcon)
{ if (/* anIcon represents a triangle */)
        return new Triangle;
     else if (/* anIcon represents a circle */)
       return new Circle;
 }
```

Code example 1: creational code which depends on the classes to instantiate.

Let us assume that both problem occurrences are treated together, a transformational engine can then reproduce the solutions of the Prototype and Factory Method patterns [14].

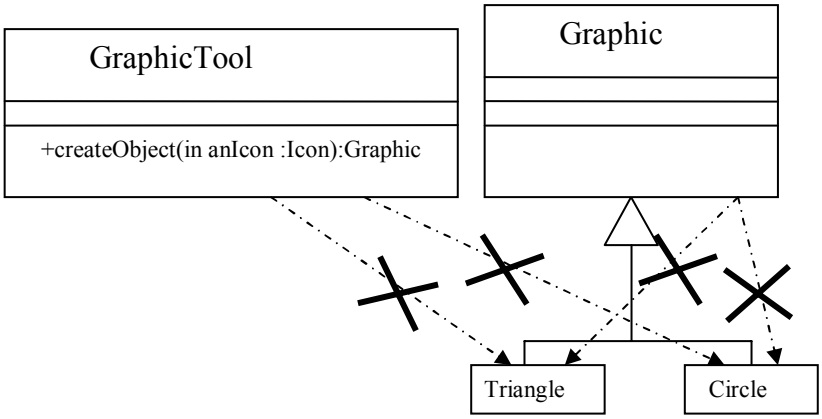

Figure 1 - GraphicTool cannot access the subclasses of Graphic

If the decoupling constraints are not carefully defined the engine might propose surprising (not to say bizarre) solutions using any legal means to find a term semantically equivalent to the original program fragment. So the constraints are crucial not only to detect design problems but also to filter out undesirable solutions produced by a transformational engine.
In this example, the hiddenSubclasses('Graphic') severely restricts the use of its subclasses. The only class whose methods can evaluate *new Triangle* is class Triangle itself. A fold transformation will thus displace this call to a new method of class Triangle, correctly reproducing the solution of the Prototype pattern [13]. The solution of the Factory Method pattern can be reproduced by the same transformation by slightly relaxing the decoupling constraint: a hierarchy parallel to that of Graphic must be allowed to access the subclasses of Graphic. This solution is sometimes interesting when such a hierarchy already exists or when the Graphic hierarchy cannot be modified (the Prototype pattern's solution adds a virtual function to Graphic to create the instances of the subclasses).

The Composite pattern

Suppose we have the situation described in Figure 2. The initial design includes classes *Picture*, *Graphic*, *Text*, *Line* and *Client*. Class *GraphicTool* contains a reference to a set of pictures. A picture is composed of other pictures, or of several kinds of graphics (such as texts or lines). Suppose that we have several *add* methods with different signatures (names and parameter type lists) and various *draw* methods.
The Composite pattern is useful when the designer does not want a class (here GraphicTool) to depend on which entities are atomic and which ones are composite. In fact there are several interpretations of what problem the pattern solves. One interpretation, depicted on Figure 2, is that the classes of the aggregation graph must be hidden except for Graphic which is an abstract class whose role is precisely to hide its subclasses. It can be translated into hiddenSubclasses('Graphic') except for the fact that the access to Picture must also be banned. A predicate to hide a set of classes could thus be defined (in terms of atomic decoupling constraints).

A more subtle interpretation considers that only some methods, e.g. *draw*, must be used "uniformly" by client classes such as GraphicTool. This can be expressed by hiding the specific versions of the method which must be used uniformly. More precisely, imposing that the *draw* methods be used uniformly means that access to each of these methods is banned !
Let us assume first that all the draw methods have first been declared as forming a coherent polymorphic function, e.g. by writing polymorphic(draw, [Text, Line, Picture], polyDraw) where polyDraw is the name this polymorphic function. Then banning access to these methods can be done by writing hidden(PolyDraw) assuming the following definition:

for all (m,c) in P,
for all NS1,
    hidden(NS1, c, name(m))
where NS1 is a namespace and P is a set of couples (method, class) forming a valid polymorphic function (their signature only varies on one parameter's type).

Definition 1: banning access to the specific methods of a polymorphic function

This constraint might be surprising but recall that we are only considering static access. So accessing Text::draw means calling draw on an expression of **static** type Text. This constraint is indeed able to detect problematic code which tests on the type of graphic objects to call specific methods. On the other hand this constraint lets the transformational engine reproduce a solution compatible with that of the Composite pattern (see Figure 3).
This solution consists in introducing a new virtual function *draw* on a new abstract class above Graphic and Picture and in replacing calls to specific versions of draw by calls to this virtual function. The decoupling constraint is respected because dynamic binding replaces static binding. To be quite correct, our current prolog prototype produces a slightly more involved solution than that of Figure 3. Identifying Graphic and this new abstract class to reach the simpler solution of Figure 3 is beyond the scope of this paper.

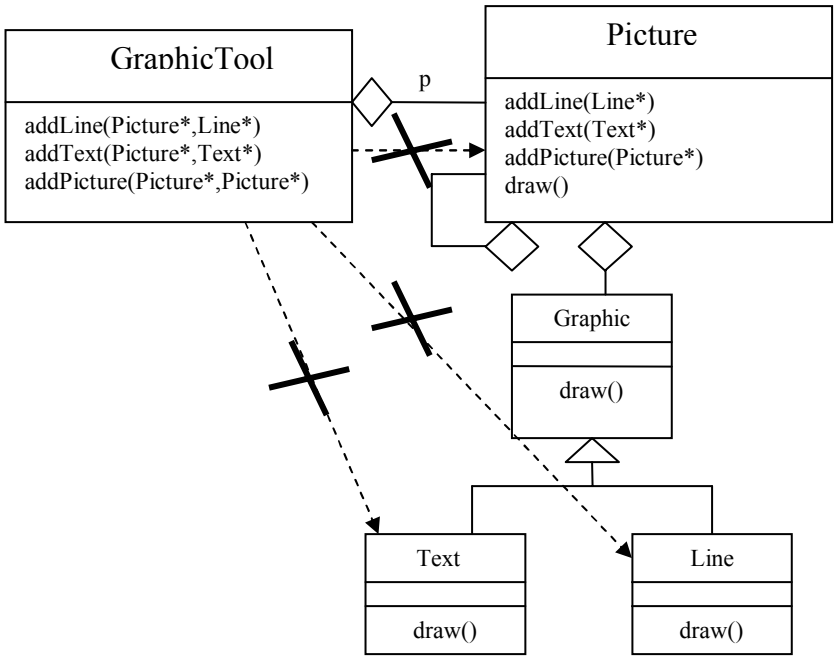

Figure 2 – A situation where the Composite pattern can be useful

We have not mentioned the treatment of the addLine, addText and addPicture methods yet. The problem, still relevant of the Composite pattern, is slightly different from that of the draw methods. The problem here only occurs in the stricter interpretation of the pattern which is illustrated on Figure 2: the whole classes Text, Line and Picture are hidden to clients such as GraphicTool.

In this case the signatures of the addLine, addText and addPicture methods in GraphicTool violate the decoupling constraint: the class names are used as types in the signatures. With the additional information that these methods form a coherent polymorphic function a simple transformation can infer more general types and replace these methods and the corresponding aggregations with the more general add(Graphic*, Graphic*), add(Graphic*) methods and the associated aggregation.

The reader will notice that the solution does respect the constraint that clients (such as GraphicTool) must not mention the hidden classes Text, Line and Picture. The reader will also notice that the original Access Graph notation did not take this kind of access (a type name in a method's signature) into account. It did not either take the type of the aggregation p into account (nor the type of instance or class variables). Finally, remember that all the solutions proposed by a transformational engine should be validated by the designer.

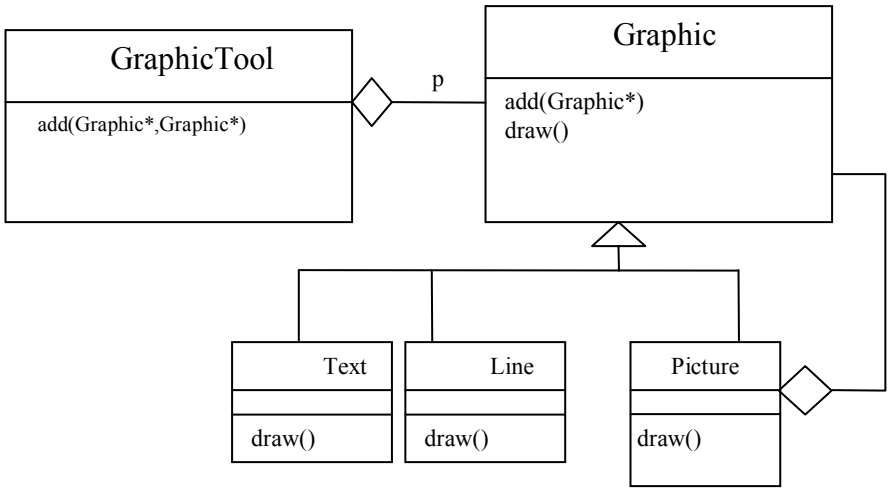

Figure 3 – The canonical solution of the Composite pattern

4. Conclusion

In this paper we adapted the Access Graph notation of Ardourel and Huchard for static access rights to express decoupling constraints. This new formalism will be used in the RNTL LUTIN project to detect design problems and propose solutions using program or model transformations. This is a major improvement on state-of-the-art tool support for design pattern which assumed that designers know which pattern to apply to which problem. A prototype is prolog is able to correctly produce the solutions of the examples of the paper, according to the declared constraints (expressed in prolog directly), even though the automatic detection of problems is not yet implemented. Many more patterns that the few mentioned in this paper rely, at least partially, on decoupling constraint. respecting the very general but vague principle "program to an interface not to an implementation" was indeed announced as one of the fundamental objectives of [1]. We believe that formalizing decoupling constraints by constraints on static access rights contributes to make this principle more precise so that it can be better supported by tools.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Pattern - Elements of Reusable Object-Oriented Software*: Addison Wesley, 1995.

[2] A. H. Eden, "Precise Specification of Design Patterns and Tool Support in Their Application," in *Department of Computer Science*: Tel Aviv University, 2000.

[3] J. O. Coplien, "Software Design Patterns: Common Questions and Answers," presented at Object Expo, New York, 1994.

[4] E. Agerbo and A. Cornis, "How to preserve the benefits of Design Patterns," presented at OOPSLA, 1998.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Pattern - Elements of Reusable Object-Oriented Software*: Addison Wesley, 1995.

[6] L. Rising, *The Pattern Almanac 2000*: Addison Wesley, 2000.

[7] J. Kerievsky, *DRAFT of Refactoring To Patterns (available at http://www.industriallogic.com/xp/refactoring/):* Industrial Logic, Inc., 2002.

[8] O. Ciupke, "Automatic Detection of Design Problems in Object-Oriented Reengineering," presented at TOOLS, Santa Barbara, California, 1999.
[9] Y. Guéhéneuc and H. Albin-Amiot, "Using design patterns and constraints to automate the detection and correction of inter-class design defects," presented at TOOLS USA, 2001.
[10] M. Ziane, S. Chantit, and S. Ammour, "How Could Tools Help Solve the Problems of Design Patterns?," presented at submitted to TOOLS USA 2003 available at http://www-poleia.lip6.fr/~ziane/zca.pdf, 2003.
[11] G. Ardourel and M. Huchard, "Access Graphs, Another View on Static Access Control for a Better Understanding and Use," *Journal of Object Technologies*, vol. 1, pp. 95-116., 2002.
[12] L. Tahvildari and K. Kontogiannis, "A Software Transformation Framework for Quality-Driven Object-Oriented Re-engineering," presented at the IEEE International Conference on Software Maintenance (ICSM), Montreal, 2002.
[13] M. Ziane, "Towards Tool Support for Design Patterns Using Program Transformations," presented at Langages et Modèles à Objets (LMO'01), Le Croisic, France, 2001.
[14] M. Ziane, "A Transformational Viewpoint on Design Patterns," presented at IEEE Automated Software Engineering Conference (ASE), Grenoble, 2000.