A model-driven framework for representing and applying design patterns

Ghizlane El Boussaidi and Hafedh Mili
Université du Québec à Montréal, Canada
el_boussaidi.ghizlane@courrier.uqam.ca, hafedh.mili@uqam.ca

Abstract

Design patterns encode proven solutions to recurring design problems. To use a design pattern properly, we need to 1) understand the design problem the pattern resolves, 2) recognize an instance of this problem in the model at hand, and 3) to transform the model to produce the proposed solution. We argue that an explicit representation of the design problem solved by a pattern is key to supporting each one of these tasks. We propose to represent a design pattern using a triple (MP, MS, T) where MP is a model of the design problem solved by the pattern, MS is a model of the solution proposed by it, and T is a rule-based representation of the transformations embodied in the application of the pattern. In this paper, we describe the principles underlying our approach and the current implementation using the Eclipse Modeling Framework™ and JRules™.

1. Introduction

Design patterns provide models of solutions to recurring design problems in precise contexts. They have raised a lot of interest in the past years, and several approaches were proposed to specify and apply them. A number of the existing approaches focused on representing the structure of patterns but neglected the pattern application aspect [4][5]. Furthermore, these approaches limited themselves to the representation of the solutions proposed by the patterns and, to a lesser extent, to maintain the integrity of these solutions when the system at hand evolved. Other approaches focused on the application part but did not represent the structural aspects explicitly [2][8]. None of these approaches tackled the problems of 1) identifying opportunities for applying the pattern, and 2) mechanizing the implementation of its solution.

In this paper, we propose an approach for representing and applying design patterns based on an explicit representation of design problems patterns are intended to solve. Indeed, we argue that the explicit representation of the design problem solved by a pattern provides a better characterization of the pattern and its usage context. It also enables to better assess the suitability of the pattern to the problem at hand. Finally, it enables to better understand the consequences of applying the pattern by describing the models of the system at hand before and after the application of the pattern.

We provide an overview of the principles underlying our approach in section 2. In section 3, we use the Visitor pattern to illustrate the representation of design problems and to introduce our language for describing the design problems solved by patterns. Section 4 describes the representation of solution models. The rule-based representation of transformation rules is provided in section 5. The implementation is described in section 6. We discuss related work and outstanding issues in section 7 and conclude in section 8.

2. Overview of the approach

Figure 1 illustrates the principles of our approach. We represent a design pattern by a triple (MP, MS, T). MP is a model describing the structure of the problems solved by the pattern. MP is a meta-model to the extent that its instances are models. MS depicts the structure of the solution proposed by the pattern to solve the problem. T specifies the transformation inherent to the application of the pattern.

Our approach relies on the existence of a catalog of such triples in a modeling workbench. Given an input model, the first step consists of detecting potential instances of the problems solved by the patterns of the catalog in this model, i.e. matching the MP components of the patterns, to the model at hand. A successful match marks the entities of the input model by the roles they play in the model of the problem. The transformation rules are then applied to the so marked model. The outcome is the input model where an instance of a design problem has been replaced by the corresponding instance of the solution.
Applying patterns

Transformed model

Marked model

Solution (meta)models (MS)

Problem (meta)models (MP)

Transformation rules (T)

Input model

Solution instances

Problem instances

Representing patterns

Applying patterns

Matching problem models to an input model is actually a two-step process. The first step does an initial marking that recognizes things such as abstract classes, abstract methods, etc. The second step consists of matching so marked input models to the patterns inherent in problem models.

3. Representing design problems

3.1. Example: the visitor pattern

Consider a compiler that represents a program as an abstract syntax tree (AST). The nodes of a tree could represent assignment statements, variables, etc. The compiler needs to perform different operations on trees such as type checking, code generation, and the like. Most of these operations have node specific implementations. A plausible object-oriented design would model this application using the class hierarchy illustrated in figure 2. However, if the underlying language (i.e. AST) is stable but the list of operations that the compiler wishes to perform is not, then this solution is not adequate. Indeed, to add an operation such as “prettyPrint()”, we need to add an abstract method to class Node, and provide an implementation in each class in the hierarchy. The visitor pattern proposes a way of making the change more localized.

Our goal is to capture this problem in a meta-model whose instances are models such as the one in figure 2. Such a meta-model is shown in figure 3. This meta-model depicts a class hierarchy where the root is the class called AbstractClass and the sub-classes are represented by the class ConcreteClass. It also shows the operations that are affected by the pattern. In this case, each ConcreteClass implements all of the abstract operations of the AbstractClass from which it inherits. We specify this as a constraint—called “Homomorphism_with”—between the association ‘inherits_from’, between classes, and the association ‘implements’, between the corresponding operations. To keep the meta-model simple, we did not show the parameters and return types of the operations.

Figures 2 and 3 illustrate the approach.

Figure 2. The node class hierarchy

Figure 3. A meta-model of the problem solved by the Visitor pattern

However, the essence of the problem resolved by the visitor pattern is the evolution scenario for which the class hierarchy is not adequate, namely, the future addition of operations on existing node types. We represent this evolution by adding the symbol “++” to the cardinalities of the associations ‘has_message’ and ‘has_method’ between AbstractClass and its operations, on one hand, and ConcreteClass and its methods, on the other. Accordingly, the model shown in figure 3 specifies that both the number of operations per AbstractClass and the number of methods per ConcreteClass are geared for change.

3.2. A language for specifying problem models

Our language for describing problem models is based on the UML meta-model, to which we added a number of constructs to represent design problems. A key construct, illustrated in the previous example, represents likely evolution scenarios of models over time. We call these time-varying aspects time hotspots, and a number of design patterns aim at making models more resilient to evolution. By studying the pattern catalog proposed by Gamma et al.[6], we were able to express most evolution scenarios as evolving cardinalities of meta-model level associations, e.g. the number of operations supported by a class hierarchy, the number of subclasses of a given class, etc. The complete language extensions are described in [7].

4. The solution model

The Visitor pattern proposes a solution to the design problem described in section 3.1. This solution consists of creating one class per operation, but grouping all of the implementations of that operation for all the node types in that class. Objects of this class, called a visitor somehow figure out which method to invoke depending on the node type they are looking at. When we apply this solution to the example of figure 2, we get two class hierarchies (i.e. node types and visitors),
as shown in figure 4. Hence, adding a new operation consists of adding a new class in the visitors hierarchy.

```java
Node
   Accept(NodeVisitor v)
AssignmentNode
   Accept(NodeVisitor V)
VariableRefNode
   Accept(NodeVisitor V)
```

NodeVisitor
   VisitVariableRefNode(VariableRefNode)
   VisitAssignmentNode(AssignmentNode)

GenerateCodeVisitor
   VisitVariableRefNode(VariableRefNode)
   VisitAssignmentNode(AssignmentNode)

TypeCheckVisitor
   VisitVariableRefNode(VariableRefNode)
   VisitAssignmentNode(AssignmentNode)

V.VisitAssignmentNode(this)
V.VisitVariableRefNode(this)

Figure 4. The solution proposed by the visitor pattern

Figure 5 shows a representation of the solution. We used a similar approach to that for problems. The model shows a class hierarchy of visitors corresponding to operations and a class hierarchy of elements representing the entities on which the operations are executed. To specify that new visitors may be added, the 'implements' association between ConcreteVisitor and Visitor has the cardinality 0..*,++.

Figure 5. A meta-model of the solution

5. Representing transformation

Fundamentally, models are graphs, and model transformation is a special kind of graph transformation. Our approach to transformation specification can be seen likened to a rule-based implementation of graph grammars. At a very coarse level, the transformation inherent to the application of a design pattern can be represented by a single rule whose left hand side (LHS) represents the problem model (MP), and whose right hand side (RHS) represents the solution model (MS). However, such a rule would be complex and do not promote the reuse of some elementary transformations that are common to several patterns. Therefore, we decided to break up this complex rule into a set of elementary rules while preserving the semantics of the original rule. To do so, we propose two heuristic. The first heuristic enables to break up a rule by simplifying its RHS part. The second heuristic enables to simplify the LHS of a rule.

For example, the transformation inherent to the application of the visitor pattern, can be written using a single rule R: MP ⇒ MS. The first heuristic enables us to break this rule into two sets of rules. The first set of rules generate the elements of the solution, one element at a time. The second set of rules generate the relationships between the generated elements. The LHS of all these rules is the same as the rule R (i.e. MP). In other words, we generate the solution model/graph by generating the vertices, one by one, and then by generating one or several edges at a time. This heuristic is semantic-preserving since we do not lose any context in the simplification. Figure 6 shows some of the rules obtained by applying this heuristic to the rule R. Rules R1 and R2 generate two distinct entities, whereas Ri generates the inheritance relationship between them.

Figure 6. Applying the first heuristic to rule R

Our second heuristic aims at simplifying the LHS of the transformation rules by keeping only the necessary and sufficient context to generate the solution model properly. Consider the rule R1 in figure 6. While this transformation is performed within the context of an instance of the problem model, the generated class Element depends only on the class AbstractClass of the problem. We refer to this as the necessary context. For a given rule, we define the sufficient context as the smallest set of LHS elements that would enable the production of a correct entity. The sufficient context is greater than the necessary context, but smaller than the context embodied in the entire problem model. We have no precise way of determining the sufficient context a-priori. Experimentation should help us refine this heuristic. The application of this heuristic to the rules R1, R2 and Ri gives the rules shown in figure 7.

We also defined some transformation rules which concern the peripheral entities that use the entities to which the pattern applies. For example, if a class A is transformed to a class B, we replace all the external references (e.g. association end) to A by references to B.
6. Implementation

We implemented our language for representing patterns as a meta-model [7]. To do so, we used the Eclipse Modeling Framework™ (EMF) which includes a package called ECore that implements a lightweight version of MOF. Our meta-model was implemented as an extension to ECore. We also implemented our transformation rules using ILOG JRules. JRules is a hybrid object-rule system where the condition and action parts of rules refer to Java objects and methods. In our case, conditions and actions refer to elements of problem and solution models represented as instances of the ECore meta-model.

7. Discussion

Related work: Contrary to approaches that were proposed to represent or apply design patterns [2] [4] [5] [8], our approach proposes an explicit representation of a pattern as well as the transformation inherent to its application. None of these approaches was interested in specifying the problems solved by the patterns. Regarding transformations, our approach is more connected to graph transformations based approaches. Current approaches [1] [3] use graphs specified by UML class diagrams. Our approach is similar to these approaches since LHS and RHS parts of our rules are typified graphs whose types are defined by the problem and solution models related to the patterns. However, contrary to these approaches, we do not have to control the application of rules. In fact, we trust the control techniques provided by the rule-based engine, e.g. priorities management, rules chaining, etc.

Outstanding issues: We are interested in potential conflicts when different instances of the same pattern or different patterns apply to the same model. We plan to use a pattern instance numbering which can help, but not always. We also must tackle the problems raised by incomplete input models. Indeed, it is necessary to ensure that the transformation rules of a pattern are launched only when all the elements described in the related problem model are detected.

We believe that rules could be used as constraints to facilitate the automatic recognition of problem instances in models. Finally, we plan to extend our approach to capture the behavioral aspect of patterns, which is an important aspect in several patterns, namely, the behavioral ones.

8. Conclusion

We proposed an approach for representing and applying design patterns. Our approach aims at providing developers with a repository of reusable models specifying patterns and a rule-based environment to use these models. The key element of our approach is the explicit representation of the problems solved by design patterns. Indeed, representing design problems insures a better understanding of the patterns and the mechanization of their application. Hence, we proposed to specify a design pattern as a reusable artifact characterized by a model of problems, a model of solutions and a set of rules describing the transformation inherent to its application. Our approach is declarative since the transformation rules are expressed in term of the elements of problem and solution models and not in term of the elements of the models to which we wish to apply the design pattern.

9. References