Visualizing Design Patterns With A UML Profile

Jing Dong and Sheng Yang
Department of Computer Science
University of Texas at Dallas, Richardson, TX 75083, USA
{jdong,syang}@utdallas.edu

Abstract

In this paper, we present a UML profile which defines new stereotypes, tagged values and constraints for visualizing design patterns in UML diagrams. These new stereotypes and tagged values are attached to a modeling element to explicitly represent the role the modeling element plays in a design pattern so that the user can identify the pattern in a UML diagram.

1. Introduction

Design patterns [5] have become increasingly popular among software developers since the early 1990s. They help developers communicate architectural knowledge, help people learn a new design paradigm, and help new developers ignore traps and pitfalls that have traditionally been learned only by costly experience. Design patterns are usually modeled and documented in the Unified Modeling Language (UML) [1]. However, UML does not keep track of pattern-related information when a design pattern is applied or composed. The model elements, such as classes, operations, and attributes, in each design pattern usually play certain roles that are manifested by their names. The application of a design pattern may change the names of its classes, operations, and attributes to the terms in the application domain. Thus, the role information of the pattern is lost. It is not obvious which model elements participate in this pattern so that it is hard for a designer to identify design patterns in software system designs [10][3]. There are several problems when design patterns are implicit in software system designs: first, software developers can only communicate at the class level instead of the pattern level since they do not have pattern-related information in system designs. The benefits of design patterns are compromised because the designers cannot communicate with each other in terms of the design patterns they use and their design decisions and tradeoffs. Second, each pattern often documents some ways for future evolutions, which are buried in system designs. Third, it may require considerable efforts on reverse-engineering design patterns from software system designs [7].

UML provides extension mechanisms that allow us to define appropriate labels and markings for the UML model elements. In order to retain the pattern-related information even after the pattern is applied or composed, we define an extension to UML. In this extension, pattern-related information is explicit so that a design pattern can be easily identified when it is applied and composed. The extensions have been defined mainly by applying the UML built-in extensibility mechanisms, such as stereotypes, tagged values and constraints.

2. A New UML Profile

A UML profile is a stereotyped package that contains model elements that have been customized for a specific domain or purpose by extending the meta-model using stereotypes, tagged values and constraints. A stereotype, denoted by <<stereotype-name>>, allows the definition of extensions to the UML vocabulary. It groups tagged values and constraints under a meaningful name. When a stereotype is branded to a model element, the semantic meaning of the tagged values and the constraints associated with the stereotype are attached to that model element implicitly. Tagged values extend model elements with new kinds of properties with the format of a pair of name and an associated value, i.e., {name=value}. Constraints add new semantic restrictions to a model element by the Object Constraint Language (OCL) [12].

2.1 Stereotypes and Tagged Values

To explicitly visualize design pattern structure and behavior in class and state diagrams, respectively, we define a UML profile which includes four stereotypes: PatternClass, PatternAttribute, PatternOperation and PatternStateMachine, whose base classes are Class, Attribute, Operation and StateMachine, respectively. The first three stereotypes are used in class diagrams, while the fourth is used in state diagrams. Each stereotype also defines one tagged value which defines exactly what role a class, an attribute or an operation plays in a design pattern. The name of the tagged value is "pattern" and the value of the tagged value is a tuple in the format of <name:string[instance:integer], role:string>. The "name" in the tuple is the pattern name in which a model element, such as class, attribute or operation, participates. The name field of PatternAttribute and PatternOperation can be omitted if the class plays a role only in one pattern,

and this omission will not create any ambiguity. But the name field of PatternClass is mandatory. Sometimes there are more than one instance of a pattern in the system, and it is useful to distinguish the instances. The “instance” in the tuple indicates the instance of the pattern the model element participates. The “role” in the tuple shows the role that a model element plays in the pattern.

A model element may simultaneously play different roles in different patterns. In this case, a new tagged value with the format as “name[instance],role” is branded to the model element for each additional pattern it participates.

The constraint of the PatternClass stereotype is defined formally in OCL as follows:

<<PatternClass>>:
self.baseClass = Class and self.taggedValue -> exists
(tv:taggedValue | tv.name = "pattern" and tv.dataValue =
"tuple<name:string[instance:integer],role:string>")

where the base class of the PatternClass stereotypes is Class. It has a tagged value with the name of “pattern” and the value of “name[instance],role”. The types of “name” and “role” are string and the type of “instance” is integer. The constraints of the PatternAttribute, PatternOperation and PatternStateMachine stereotypes are defined similarly. These stereotypes, together with their tagged values and constraints, form a new UML profile for design patterns.

Consider an example class OracleConnPool which has an attribute OracleConnection and an operation createCon(). This class participates in both the Abstract Factory and the Singleton patterns. It plays the role of “ConcreteFactory” in the second instance of the Abstract Factory pattern and the role of “Singleton” in the first instance of the Singleton pattern. Thus, the stereotype <<PatternClass { <AbstractFactory[2], ConcreteFactory> <Singleton[1], Singleton> } >> is attached to the OracleConnPool class. Similarly, since the OracleConnection operation only participates in the first instance of the Singleton pattern, it is only attached by the <<PatternAttribute{<Singleton[1], UniqueInstance>}>> stereotype. Since the createCon() operation, plays the role of “createProduct” in the second instance of the Abstract Factory pattern and the role of “instance” in the first instance of the Singleton pattern, a <<PatternOperation>> stereotype is attached to it as shown in Figure 1.

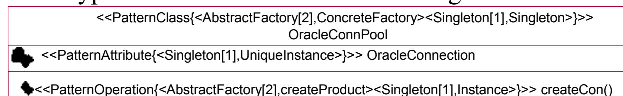

Figure 1. The OracleConnectionPool Class

Figure 2 shows an example of the state diagram that presents an object of the OracleConnection class which is attached by <<PatternStateMachine{<AbstractFactory[2], ConcreteProduct>}>>. The stereotype describes that this object plays a role of “ConcreteProduct” in the second instance of the Abstract Factory pattern.

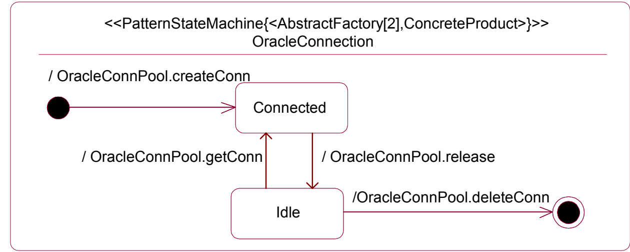

Figure 2. The state machine of OracleConnection

2.2 Constraints

There are several constraints for the stereotypes and tagged values we defined in the previous section. For brevity, we only provide three constraints in this section. We refer to [2] for other constraints.

1. The name fields of the tagged value in <<PatternOperation>> can be omitted if no ambiguity is added into the class diagram.

   <<PatternOperation>>:
   self.taggedValue.dataValue.name -> isEmpty or
   self.taggedValue -> exists (tv:taggedValue |
   tv.dataValue.name -> notEmpty)

2. Within a class, if the name field of a tagged value in <<PatternOperation>> is omitted, then the tagged value in <<PatternOperation>> has the same name as that in <<PatternClass>>.

   <<PatternOperation>>:
   self.taggedValue.dataValue.name -> isEmpty implies
   self.taggedValue.dataValue.name =
   PatternClass.taggedValue.dataValue.name

3. The role field of the tagged value in <<PatternStateMachine>> is mandatory.

   <<PatternStateMachine>>:
   self.taggedValue.dataValue.role -> notEmpty

2.3 Virtual Meta Model

A virtual meta model (VMM) is the UML expression of a formal model with a set of UML extensions. A VMM can graphically represent the relationship among the newly defined elements, i.e., PatternClass, Pattern-Attribute, PatternOperation and PatternStateMachine, and those defined by UML specification.

The VMM for the newly defined extensions is represented as a set of class diagrams. Figure 3 shows the VMM for the <<PatternClass>> stereotype. The VMM represents a Stereotype as a Class stereotyped <<stereotype>> and a TaggedValue associated with a Stereotype as an Attribute of the Class that represents the Stereotype. The Attribute is stereotyped <<TaggedValue>>. The Attribute name is the name of the tagged value. The value of a tagged value is enclosed between “<” and “>” signs which means the format of the value is <name[instrance],role>. The multiplicity

following the Attribute name ([1..*]) indicate that the tagged value may have one or more values. The VMMs for <<PatternOperation>>, <<PatternAttribute>> and <<PatternStateMachine>> are defined similarly.

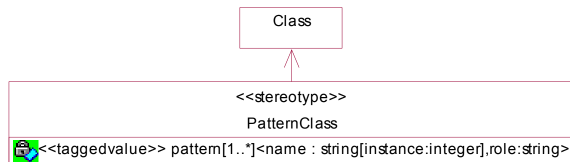

Figure 3. Virtual Meta Model for PatternClass

3. Related Work

Explicitly visualizing design patterns in UML has been investigated in [1][10], where all approaches can only represent the role a class plays in a pattern and cannot distinguish multi-instances of a pattern. In addition to class, we can represent the role an attribute (operation) plays in a design pattern and distinguish multi-instances of a design pattern.

UML extension mechanisms have been used to expand the expressive power of UML to model object-oriented framework [4], software architecture [9][6], and agent-oriented systems [11] when the original UML is not sufficient to represent the semantic meaning of the design. We extend UML with a new profile to visualize the pattern-related information hidden in a UML diagram. We define new stereotypes, tagged values and provide the constraints applied to these stereotypes and tagged values.

A visual modeling language (DPML) has been proposed to model and apply design patterns in [8]. Unlike DPML, our goal is to explicitly visualize the hidden dependency of the design pattern to its instance link. We also support design pattern composition, dynamic aspects of design patterns, OCL-style constraint language, and the overlapping of design pattern instances. In contrast to our approach that constraints UML, DPML provides new notations, such as hexagon and inverted triangle, for modeling design patterns. Although it improves the expressiveness for some special concepts, e.g., dimension, it ties up the user to their DPTool, thus, is hard to take advantage of the popular UML tools, e.g., Rational Rose, which provide more features, such as code generation. It is also a tradeoff between expressiveness and generality to provide complex visual notations just for modeling one (kind of) design pattern(s), e.g., the dimension concept only for the Abstract Factory pattern.

4. Conclusion

In this paper, we introduced a UML profile for the explicit visualization of design patterns in system designs. The application of a design pattern may change the names of classes, operations, and attributes participating in this pattern to the terms of the application domain. Thus, the roles that the classes, operations, and attributes play in this pattern have lost. Without explicitly representing pattern-related information, the designers are force to communicate at the class and object level, instead of the pattern level. The design decisions and tradeoffs captured in the pattern are lost too. Our approach uses the UML extension mechanisms to define a UML profile for visualizing design patterns. Four new stereotypes, PatternClass, PatternAttribute, PatternOperation and PatternStateMachine, are defined. Each stereotype has a tagged value. Several constraints are also defined. Using this new UML profile, one can identify pattern-related information, such as how many design patterns are used in the system, what is the role of each model element, how many instances of a design pattern are applied, and where is each instance of design pattern in the UML diagram. Thus, the notations provided in this paper help on the explicit representation of design patterns.

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[2] J. Dong and S. Yang. Visualizing Design Patterns With A UML Profile. *Technical Report UTDCS-11-03, Computer Science Department, University of Texas at Dallas*, 2003.

[3] J. Dong and K. Zhang. Design Pattern Compositions in UML. *Software Visualization - From Theory to Practice, Kluwer Academic Publishers*, pages 287–308, 2003.

[4] M. Fontoura, W. Pree, and B. Rumpe. UML-F: A Modeling Language for Object-Oriented Frameworks. *Proceedings of the ECOOP*, pages 63–82, July 2000.

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

[6] M. M. Kande and A. Strohmeier. Towards a UML Profile for Software Architecture Descriptions. *Proceedings of the UML, LNCS1939, Springer-Verlag*, pages 513–527, 2000.

[7] R. Keller, R. Schauer, S. Robitalille, and P. Page. Pattern-Based Reverse-Engineering of Design Components. *Proceedings of the 21st ICSE*, pages 226–235, May 1999.

[8] D. Mapdlsden, J. Hosking, and J. Grundy. Design Pattern Modelling and Instantiation Using DPML. *Proceedings of the 40th International Conference of Object-Oriented Languages and Systems (TOOLS Pacific), 2002*

[9] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Trans. on Software Engineering and Methodology*, 11(1):2–57, January 2002.

[10] J. Vlissides. Notation, Notation, Notation. *C++ Report*, April 1998.

[11] G. Wagner. A UML Profile for Agent-Oriented Modeling. *Proceedings of the 3rd International Workshop on Agent-Oriented Software Engineering, Bologna, Italy*, July 2002.

[12] J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.