# Supporting Flexible Reification of Design Patterns *

Wuwei Shen
Department of Computer Science
Western Michigan University
Kalamazoo, Michigan 49008, USA

Dae-Kyoo Kim
Department of Computer Science and Engineering
Oakland University
Rochester, MI 48309, USA

Jian Liu
Chen Zhao
Institute of Software, Chinese Academy of Sciences
Beijing 100190, China

## Abstract

*Design patterns have been widely accepted as a solution for solving recurring design problems in object-oriented development. Reifications of design patterns can vary from one development environment to another, and use of inappropriate reifications may impose a serious threat to quality of a software system. In this paper, we propose an approach to applying the profile mechanism in reifying a design pattern. Central to this approach are stereotypes that are defined in a profile and used to represent different roles in a design pattern. Developers can apply these stereotypes in their application model when design patterns are used. The advantage of the profile mechanism is that developers can 1) define their own reification of a pattern in a profile based on a specific software system, and 2) find errors in an application model via the conformance checking of the model against the profile. More importantly, we apply our existing tool called ICER, which is based on the profile mechanism, to provide automatic checking for the application of design patterns. To illustrate the advantage of the profile mechanism supported by ICER, we show the different reifications for the* `Observer` *pattern. Last, experimental results show that ICER does not suffer from the scalability problem as the size of an application model increases.*

*This work was supported in part by National High Technology Research and Development Program of China (863 Plan) under grant No. 2009AA010313. Also, part of the work was done during the first author's sabbatical leave at UNU-IIST and it was supported by the HTTS project funded by Macau Science and Technology Development Fund.

## 1 Introduction

Design patterns, as a technique to reuse successful designs and architectures, have demonstrated great success in supporting adaptability and extensibility during software development. Design patterns not only provide good generic solutions to recurring problems but also increase the adaptability and extensibility of a software system. In practice, when these patterns are applied to software systems, software developers are left with plenty of room for reification when modeling a software system. As stated by Coplien [1], "the structure of patterns are not themselves solutions, but they generate solutions". The solutions generated by a pattern can vary, depending on many factors during software development. Sometimes, a reification of a design pattern can be a viable solution in one development environment while the reification cannot be in another environment.

For instance, the `Observer` pattern can be reified based on [2], where the relationship between classes `Observer` and `ConcreteObserver` is reified by inheritance. But, if an implementation language does not support multiple inheritance, the inheritance solution becomes infeasible when a concrete observer class has another parent class in the model. To implement a software system in an implementation language which only supports single inheritance, developers should make some changes for the reification of the `Observer` pattern accordingly.

Another factor that can affect the application of a design pattern is the characteristics of an application. Sometimes, a design pattern cannot work well for a certain development environment; and so the modification of the pattern should be made. For instance, the `Visitor` pattern proposed by [2] is quite effective

1530-1362/10 $26.00 © 2010 IEEE
DOI 10.1109/APSEC.2010.24

in handling a group of elements in an object structure when new operations are added to a software system. But if a software system keeps adding new elements to the object structure, all the visitor classes will be changed to include a set of new operations, each of which deals with a newly-added element. Obviously, the reification of the `Visitor` pattern suggested by [2] is not a feasible solution for a software system which constantly changes its structure. Consequently, reifications of the variations should be created.

Thus, the reification of design patterns is not unique. Even when developers sometimes know a correct reification of a design patterns, they can still inadvertently invite some errors during the application of the design pattern, especially when a software system becomes complex. Helping developers detect errors based on the reification of a design pattern is valuable in supporting design patterns. But, any support for reifying a design pattern should consider the following issues. Firstly, as important participants, software developers are familiar with all aspects of the software system being developed. So, they should propose the factors which can have impact on the application of design patterns. Based on these factors, *it is the developers who choose their correct reification of a design pattern.* Thus, how developers represent a pattern reification is the first issue to be resolved. Secondly, after giving an appropriate reification of a design pattern, developers can apply the reification in an application model. Thus, the relationship between a pattern reification and the model where the reification is applied is another issue to be addressed.

Fortunately, the profile mechanism in UML [3] provides a method of solving the above issues. A profile which belongs to the metamodel layer (i.e. the $M_2$-layer) can be applied by developers in giving their own reification of a design pattern. Stereotypes in a profile are used to define different roles in a design pattern while the Object Constraint Language (OCL) [4] is used to represent additional constraints to complement the reification of a design pattern. According to a profile, an application model should stereotype model elements which participate in the pattern. As a result, the application model establishes an `instance-of` relation between the profile and itself.

The `instance-of` relation provides a solid foundation for tool support. It is the `instance-of` relation that allows us to find whether an application model conforms to a profile via the conformance checking. We have developed a tool called ICER which supports conformance checking based on the IBM Rational Rose. In this paper, we apply the ICER tool in validating the conformance of design patterns.

This paper is organized as follows. Section 2 presents the problem statement followed by the ICER solution. Section 3 demonstrates how the Observer Pattern can be reified in two different ways. Section 4 discusses the scalability issue of ICER using some experimental results. Last, section 5 gives some discussions on related work and draws a conclusion.

## 2 ICER Solution

In this section, we first present a problem and then show how ICER attacks this problem.

### 2.1 Problem Statement

Many approaches [5, 6, 7, 8, 9] provide a set of pre-defined constraints to enforce the error detection. These ad-hoc methods do improve the quality of a software system. For example, in the observer pattern [2], class *ConcreteSubject* requires an operation called *SetState*. This can be implemented as a constraint on class *ConcreteSubject*. Let us consider a clock application where there exists a digital clock that extends a clock and the digital clock displays the current time in the digital format. So, classes *ClockTimer* and *DigitalClock* are designed to represent and show the current time respectively. Obviously, we can apply the *Observer* pattern to this example, as shown in Figure 1 where classes *Observer* and *Subject* are introduced as a parent class. However, class *ClockTimer*, as a concrete subject in Figure 1 fails to provide operation *SetState*. This kind of errors can be easily detected by the tools supporting the above requirement.

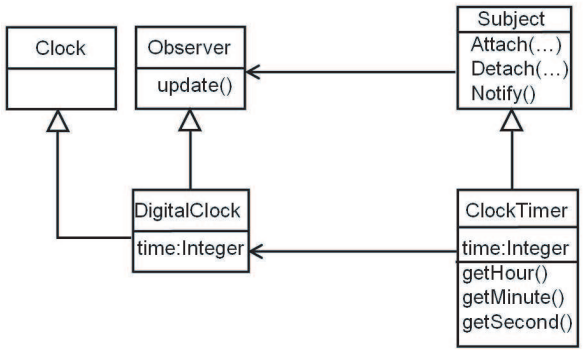


**Figure 1. A class diagram for a clock application.**

However, there exist some problems in these ad-hoc methods because they seriously undermine the flexibility required in software development. Let us assume the following situation after operation *SetState*

was added by a C++ team. The diagram, which is a correct model, was given to a Java team that developed the same application in the Java environment. According to Java, the Java team must remove multiple inheritances for any class. Namely, class *DigitalClock* cannot have two parent classes. Otherwise, the generated program cannot be compiled by a Java compiler. Obviously, the reification of the *Observer* pattern is affected by the implementation environment. A tool supporting the ad-hoc method becomes hard to adjust to a new constraint rule unless the tool is recompiled and generated after revising the existing constraint.

As a general-purpose modeling language, UML does not precisely give guidelines of how to apply its diagrams such as class diagrams in a development process. Thus, the constraints on UML (class) diagrams designed in a development process are decided by many factors such as the implementation platform. So the lack of extension and adaptation of a *fixed* set of constraint rules imposed on certain development environment can be of little value to software developers who choose a different environment.

While developers sometimes know what a valid model should be designed, they still invite some inconsistencies/errors when a software system becomes more and more complex. While a Java team might easily identify the error if class *DigitalClock* has another parent class in Figure 1, it becomes very hard for developers to find such errors in a model with more than several hundreds of classes and relationships. Hence, the tool-based support of detecting inconsistencies/errors in UML models has significantly increased the success of complex software projects. However, the most challenging problem is how a tool can help developers introduce and validate constraints for a specific software system.

## 2.2 ICER Solution

The four-layer architecture provides a viable means to solve the above issues and also lays a foundation for the ICER tool. UML is based on a four-layer metamodeling architecture where each successive layer is labeled from $M_3$ to $M_0$, which are usually named the meta-metamodel, metamodel, model, and user objects respectively. A model at the $M_i$ layer can be regarded as an instance of a model at the $M_{i+1}$ layer. The UML metamodel belongs to an $M_2$-layer model in the four-layer architecture. Also we call an $M_1$-layer model an application model in the remaining text. So any UML application model can be regarded as an instance of the UML metamodel.

A profile is an important mechanism to extend the UML metamodel. Central to a profile is the stereotypes that brand model elements for special purpose. The advantage of the profile mechanism is that the restrictions on how a profile extends the UML metamodel is purely additive and thus most UML tools can support profiles easily. Since developers should be provided with a method to introduce their constraints on some model elements, *stereotypes* are a viable means to brand these model elements to enforce new constraints.

According to the UML specification [3], stereotypes can have a tag definition and constraint. Both a tag definition and stereotype apply to all model elements branded by that stereotype. We choose the Object Constraint Language (OCL) as a language to represent constraints on a stereotype. Constraints on a stereotype impose semantics and restrictions on its instances. When an application model includes the stereotypes, the *instance-of* relationship is implied and the conformance checking ensures all constraints given in a profile be satisfied by an application model.

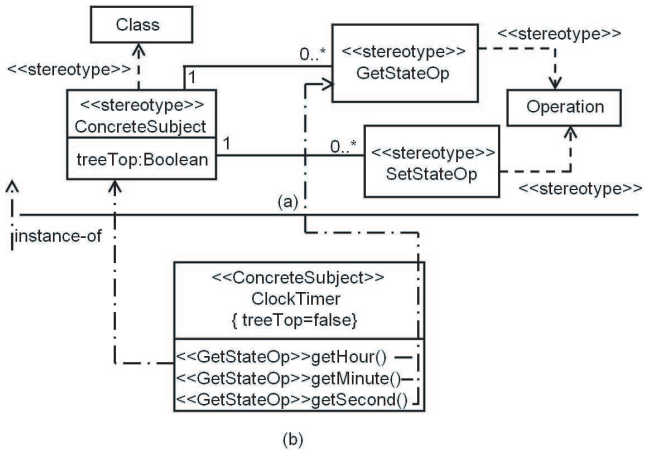


**Figure 2. Example of a UML profile.**

Let us back to the clock example to illustrate how a profile can be applied to support "class *ConcreteSubject* requires an operation called *SetState*". While readers might guess how the diagram in Figure 1 is designed based on the standard observer pattern diagram given in [2], a profile is an excellent method explicitly showing how an original pattern is applied to an application model. For clarity, we call each model element in an original pattern a role. In general, each role required in an original pattern is introduced as a stereotype in a profile. Therefore, an instance of the stereotype plays the role in an application model of this pattern. For instance, role *ConcreteSubject*, which represents the subjects observed in the original observer

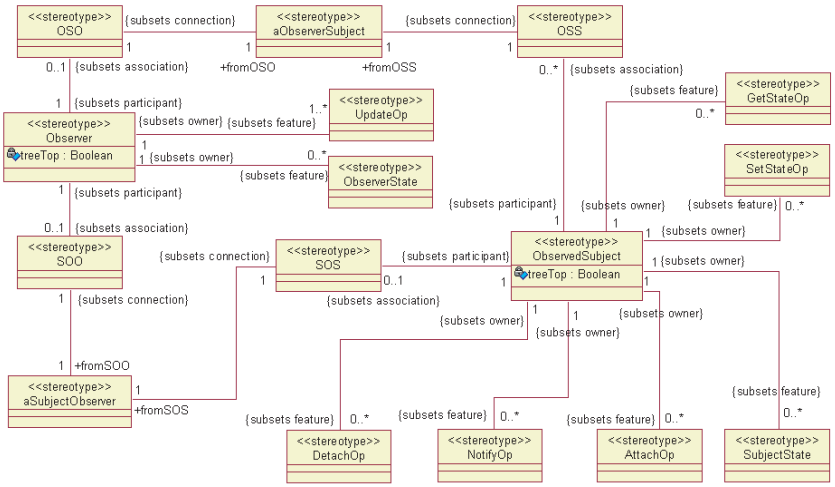


**Figure 3. A Profile for the observer pattern.**

pattern, is introduced by a stereotype *ObservedSubject* in Figure 2(a). A tag definition *treeTop* is introduced to shown whether an instance of *ObservedSubject* plays role *Subject* in the original Observer pattern. Therefore, it is not necessary to introduce a new stereotype for role *Subject*. In the clock example, class *ClockTimer* in Figure 1 actually represents objects that are observed. So, in the application model of the observer pattern shown in Figure 2(b), class *ClockTimer* is an instance of stereotype *ObservedSubject*. In such a way, the relationship between a model element such as class *ClockTimer* and its corresponding role such as *ConcreteSubject* in an original pattern is obviously shown.

Likewise, in the original observer pattern, role *ConcreteSubject* has two kinds of operations, i.e. role *SetState* and role *GetState*. They are represented by stereotypes *SetStateOp* and *GetStateOp* respectively in the profile shown in Figure 2(a). The association between *ObservedSubject* and *GetStateOp* represents an instance of *ObservedSubject* can have an operation that instantiates *GetStateOp*. The real number of the operations is given by the multiplicity of the association at the end of *GetStateOp*. In this case, the number could be zero or any integer value. So does the association between *ObservedSubject* and *SetStateOp*.

In the clock application, we concentrate on the observed timer *ObservedSubject* shown in Figure 2(b). Class *ClockTimer* has three methods that are an instance of *GetStateOp*. Once an application is given, an *instance-of* relationship between the application model and its profile can be established as shown in Figure 2.

However, the constraints given in a class diagram is sometimes not enough. For example, in the original observer pattern, method *SetStateOp* is only required for class *ConcreteSubject* that is a child class of *Subject*. This constraint cannot be given by the multiplicity of the association between *ObservedSubject* and *SetStateOp* in Figure 2 because stereotype *ObservedSubject* does not distinguish whether an instance of *ObservedSubject* is a root class in a inheritance hierarchy or not. To this end, we write the following OCL constraint:

Stereotype *Context ObservedSubject*
*inv: not self.treeTop implies (self.getStateOp* $\rightarrow$ *size()* $>$ *0 and self.setStateOp* $\rightarrow$ *size()* $>$ *0)*

The conformance checking checks not only constraints given in a class diagram such as multiplicity but also OCL constraints such as the above one. Obviously, class *ClockTimer*, as an instance of stereotype *ConcreteSubject*, does not satisfy the above constraint because class *ClockTimer* has no instance of *SetStateOp*-class *ClockTimer*'s *size*() is zero. Thus, the conformance checking reports a warning message on class *ClockTimer*.

## 3 Different Reifications of the Observer Pattern

In this section, we use the *Observer* pattern to illustrate how the solutions of the *Observer* pattern can be reified in two profiles based on two implementation platforms. The profiles for the rest of the design patterns given by GoF are available at [10].

### 3.1 The First Reification of the observer pattern

The observer pattern [2] is a commonly used behavioral pattern that defines a one-to-many dependency

between subject and observer objects. When a subject changes its state, all its observer objects are notified and updated accordingly. Many research works [11, 12, 13, 14] follow the description given in the GoF Book [2] and reify the observer pattern as follows. The generalization relationship between *Observer* and *ConcreteObserver* is reified as inheritance as suggested. More specifically, the *Observer* and *Subject* are implemented as abstract classes while the *ConcreteObserver* and *ConcreteSubject* are concrete classes that inherit from the abstract classes *Observer* and *Subject* respectively. There is one unidirectional association from *Subject* to *Observer* while another unidirectional association connects from *ConcreteObserver* to *ConcreteSubject*.

Based on this particular realization, we design a UML profile which introduces sixteen stereotypes, each of which plays one role in the observer pattern. Table 1 summarizes the stereotypes[1] defined in the profile. The profile for the observer pattern is shown in Figure 3. Any model intended to apply the above realization at the application level should be an instance model of the profile. In this profile, stereotypes *Observer* and *ObservedSubject* play the roles *Observer* and *Subject* respectively. In order to reduce the number of stereotypes in a profile, we define a tag *treeTop* in stereotype *Observer* to differentiate whether an instance of *Observer* or *ObservedSubject* is abstract or concrete. If the value of *treeTop* in an instance of *Observer* is true, then the instance is an abstract *Observer* class at the application level. Otherwise it is a concrete *Observer* class. Likewise, *treeTop* is introduced as a tag definition for stereotype *ObservedSubject*.

Stereotypes *AttachOp*, *DetachOp*, *NotifyOp*, *UpdateOp*, *GetStateOp*, and *SetStateOp* are introduced as operations required in different stereotypes. For the same reason, stereotypes *ObservedState* and *SubjectState* represent an attribute required in stereotype *Observer* and *ObservedSubject* respectively. However, the associations such as the one between *Subject* and *Observer* in the observer pattern should be enforced by several stereotypes. Based on the UML metamodel, the associations added between stereotypes in the profile should specialize the usage of the associations of the UML metamodel. Stereotype *aSubjectObserver* represents an association which is required to connect an abstract class *ObservedSubject* and an abstract class *Observer*. In order to represent this connection, we introduce stereotype *SOS* and *SOO* that extends metaclass *AssociationEnd*. Stereotype *SOS* denotes one

[1]Here we still use the metaclasses in UML v1.5 to clarify the roles played by a stereotype such as Operation. But we can switch to UML 2.0 easily.

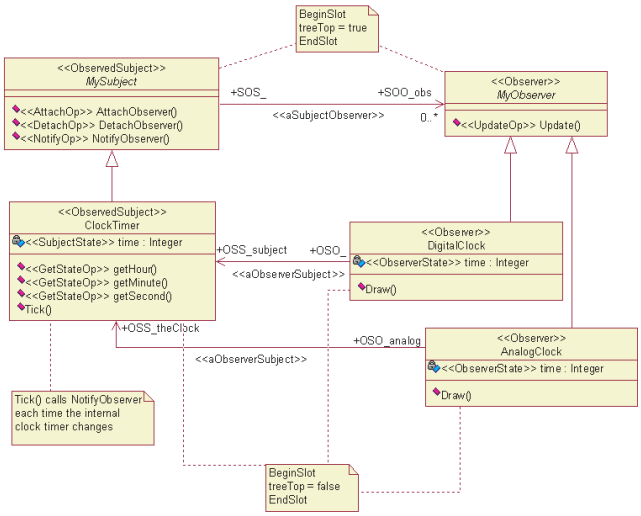


**Figure 4. A UML Model for the Clock Application Based on the observer pattern.**

end of the association *aSubjectObserver* at stereotype *ObservedSubject* while stereotype *SOO* represents the other end at stereotype *Observer*. More details about the stereotypes can be found at [10].

While a profile provides some constraints such as the multiplicity for an association, there exist some constraints that cannot be represented graphically in the profile. We use the OCL to express these constraints. To illustrate how OCL is applied, we only discuss one constraint on stereotype *Observer*. Readers are referred to [10] to find all OCL constraints related to the profile shown in Figure 3. The constraint specifies that a concrete class in the hierarchy must have one association end connecting to *aObserverSubject* and at least one state observed by itself (*ObserverState*) and it is shown as follows:

*Context Observer inv:*

*not self.isAbstract implies (self.oSO → size()=1 and self.sOO → size()=0 and self.observerState→size()>0)*

The observer pattern can be applied to many applications. For example, one application requires two types of clocks, digital clock and analog clock, to represent the time in a digital and analog format respectively. Both clocks monitor a real timer. Applying the Observer profile to this application, we introduce two concrete Observer classes, i.e. *DigitalClock* and *AnalogClock*, instantiated from the *Observer* stereotype with the tag *treeTop* set to *false*. Similarly, one concrete Subject class, i.e. *ClockTimer*, which is an instance of the stereotype *ObservedSubject* with the tag *treeTop* set to *false*, is introduced. Also, two abstract

**Table 1. The stereotypes and their declaration.**

| Stereotype | Base Metaclass | Tag | Description |
|---|---|---|---|
| Observer | Class | treeTop:Boolean | An instance (of this Stereotype) plays Observer role |
| ObservedSubject | Class | treeTop:Boolean | An instance plays Observed Subject Role |
| AttachOp | Operation | | An instance is a method in an instance of abstract ObservedSubject |
| DetachOp | Operation | | An instance is a method in an instance of abstract ObservedSubject |
| NotifyOp | Operation | | An instance is a method in an instance of abstract ObservedSubject |
| UpdateOp | Operation | | An instance is a method in an instance of abstract Observer |
| GetStateOp | Operation | | An instance is a method in an instance of concrete Observer |
| SetStateOp | Operation | | An instance is a method in an instance of concrete Observer |
| ObserverState | Attribute | | An instance is an attribute of an instance of concrete Observer |
| SubjectState | Attribute | | An instance is an attribute of an instance of concrete ObservedSubject |
| aSubjectObserver | Association | | An instance connects to an instance of abstract Observer and abstract ObservedSubject |
| SOS | AssociationEnd | | An instance is the end of an instance of aSubjectObserver on an instance of abstract ObservedSubject |
| SOO | AssociationEnd | | An instance is the end of an instance of aSubjectObserver on an instance of abstract Observer |
| aObserverSubject | Association | | An instance connects to an instance of concrete Observer and concrete ObservedSubject |
| OSS | AssociationEnd | | An instance is the end of an instance of aObserverSubject on an instance of concrete ObservedSubject |
| OSO | AssociationEnd | | An instance is the end of an instance of aObserverSubject on an instance of concrete Observer |

classes, each of which is instantiated from the *MySubject* and *MyObserver*, are added. We also add a unidirectional association from the class *MySubject* to the class *MyObserver*, a unidirectional association from the class *DigitalClock* to the class *ClockTimer*, and a unidirectional association from the class *AnalogClock* to the class *ClockTimer*. The class digram resulting from this application is shown in Figure 4. With the help of the ICER tool, we can check whether the class diagram is a correct instance of the profile. In other words, the diagram realizes the Observer pattern based on the above reification. The details about the checking performed by the ICER tool will be discussed in the next section.

### 3.2 The Second Reification of the Observer Pattern

However, using inheritance to realize the relationship between an abstract Observer class and a concrete Observer class is not applicable in some cases. For instance, if an implementation language (e.g. Java) does not support multiple inheritance, a concrete Observer class cannot inherit from an abstract Observer class when the concrete class has already a parent class.

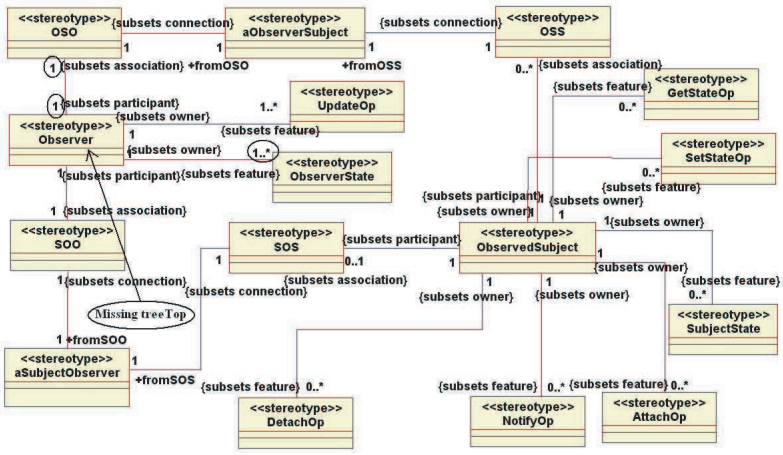


**Figure 5. Another Profile for the Realization of the observer pattern.**

Thus, one of the solutions is to have a concrete Observer class copy all the operations required by the abstract Observer class in the observer pattern and connect a unidirectional association from an abstract Subject class to the concrete Observer class itself. Obviously, in this realization, the (concrete) Observer class takes the responsibility of an abstract Observer class in the original observer pattern.

To realize the observer pattern using concrete Observer classes without any abstract Observer class, we define another profile in Figure 5. All the differences between the two profiles are marked with a circle. In terms of the number of stereotypes, this profile has the same number of stereotypes as the first profile in Section 2.1. However, the tag *treeTop* associated with the stereotype *Observer* required in the first profile is no longer necessary in the second profile because there is no generalization between the Observer classes.

Furthermore, in order to realize the associations from a concrete Observer class, we demand that in the second profile there be two unidirectional associations related to an Observer class. While the second profile still uses the first profile's stereotyped class names representing the associations and their association ends, the multiplicities for the associations between classes *OSO* and *Observer* and class *SOO* and *Observer* are changed. Instead of the value of $\{0, 1\}$ required for the multiplicity at the end of *OSO* and *SOO* class in the first profile, we require the value to be 1 for the multiplicity for the association end at class *OSO* and *SOO*, shown in Figure 5. Thus the multiplicity requires that each stereotype *Observer*'s instance have two association ends so that the instance connects to two different associations. Another difference is that the multiplicity at the stereotype *ObserverState* end for the association between *ObserverState* and *Observer* is $1..\star$ instead of $0..\star$. The reason is that each *Observer*'s instance, which is a concrete observer class at the application level, must have at least one *ObserverState* feature role. Finally, we give the OCL constraint $not(self.isAbstract)$ to require that there be no abstract Observer class in the realization.

As an example of a realization of the Observer profile in Figure 5, we apply the same example of the clock application to design a class diagram shown in Figure 6. The class diagram is a valid instance of the second profile. In this realization, the model has one concrete observer object, i.e. *DigitalClock*, to monitor a timer object. Also, the concrete observer requires that it not only define the operation *Update*, which is an instance of the stereotype *UpdateOp*, but also implement the operation.

Now we show how ICER reports errors. Assume a UML class diagram shown at the top of Figure 7 that is designed based on the profile shown in the bottom of Figure 7 (dark area). To specify why the UML diagram does not conform to the profile, we use a dotted dash line ending with an arrow to represent an *Instance-of* relationship between a model element and its corresponding meta-element. For the sake of clarity, we only give some related *Instance-of* relations for classes and their associations while the rest of relations between classes and their features are skipped.

In Figure 7, class *DigitalClockB* is an instance of the stereotype *Observer*, shown by ③. Since class *DigitalClockB* is not abstract, the OCL subexpression $self.oSO \rightarrow size() = 1$ in the third OCL invariant for *Observer* requires the number of the instances of *Observer* be equal to one, which is satisfied, shown by the *Instance-of* relation ① in Figure 7. However, the OCL

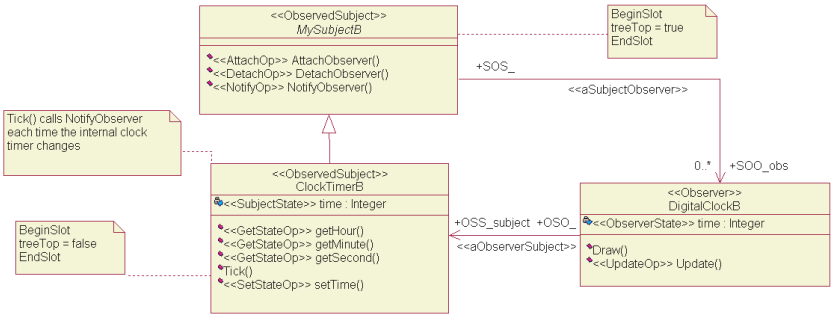


**Figure 6. A UML Model for the Clock Application Based on the Profile in Figure 5.**

subexpression $self.sOO \rightarrow size() = 0$ in that OCL invariant is not satisfied, shown by the *Instance-of* relation ② in Figure 7, because there is a unidirectional association from class *MySubject* to class *DigitalClockB* via the association end whose role name is *SOO*. Actually, the main reason of the error in the UML class diagram is that there is no abstract base class in the hierarchy of observer classes in the class diagram, which is required by the profile.

## 4 Experimental Results

To investigate the scalability issue of ICER , we studied the performance of ICER from the following aspect: to evaluate how the size of an entire model affects the performance of ICER. The evaluation of ICER was based on applying various design patterns that include the Factory Method pattern, the Decorator pattern, the Adapter pattern, the Prototype pattern, the State pattern, and the Composite pattern simultaneously to the UML metamodel. The UML metamodel, named UML2-Super-MDL-041007.mdl, was downloaded from [3].

To investigate the impact of the size on the performance, we created three instance packages with different number of metaclasses. Namely, we wrote a program that randomly chose 100, 200, and 300 metaclasses and their relationships from the UML metamodel and kept them in the three models respectively. The total sizes of these three models are 1697KB, 3070KB, and 3802KB respectively. Since the above three models have the same profile package and instance package but different sizes, we ran ICER on these three models to see the impact of the size of a model on the performance of ICER. Our evaluation is based on the two kinds of checking: graphical constraint checking and OCL constraint checking. We instrumented the code into the ICER's original code to inspect the time for the graphical constraint checking and we wrote another program to monitor the time to check all OCL constraints.

To evaluate both kinds of constraint checking, we ran ICER on each model 50 times and all measurements performed on Dell XP 1.99GHz CPU and 1 GB RAM. The total times to validate the instance-of relation in the three models are 810ms, 807ms, and 809ms. From these empirical results, we conclude that the performance of ICER is not related to the size of an entire model when a profile and an instance package are given.

## 5 Related Work and Conclusion

Supporting design patterns has drawn great attention in the community. The work by France *et al.* [5] proposed an extension of the UML that specifies a design pattern at the metamodel-level. A conformance relation is established between a pattern specification at the metamodel-level and an application model at the model-level. However, using an UML extension requires users to learn the extension and its supporting tools which usually do not tie with other CASE tools. The advantage of the profile mechanism proposed in this paper is that it does not take any extra effort for developers to learn the mechanism. More importantly, like any class model, a profile can be regarded as a package which consists of class diagrams. Also, the tool support for design patterns can be easily built based on UML CASE tools.

Another significant work in using UML extension mechanisms to define frameworks and design patterns is the UML-F by Fontoura *et al.* [6]. The UML-F is a UML profile for developing and adapting frameworks to support architecture reuse. The UML-F comprises of four layers of interrelated tags – tags for patterns, tags for construction principles, basic modeling tags, and presentation tags. Sanada and Adams [15] extended the UML-F by introducing additional tagged values and stereotypes. Also, unlike the UML-F, they used

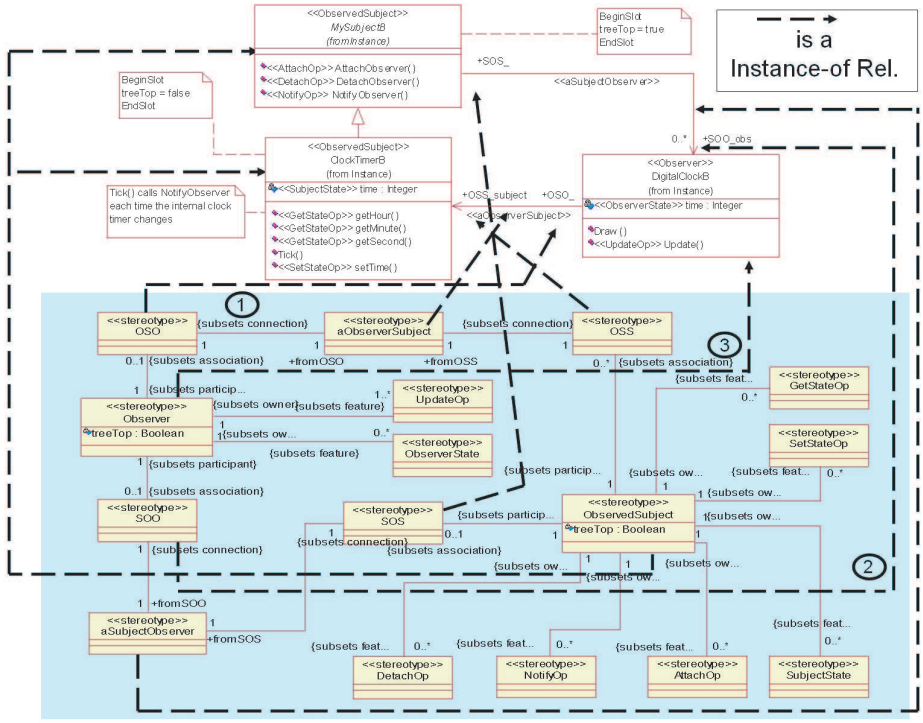


**Figure 7. Instance-of Relationship Between a Model and Its Profile.**

UML Collaborations to represent patterns. However, the use of Collaborations results in more complexity in designs over the UML-F.

Mak *et al.* [16] proposed an extension to UML 1.5 to precisely model design patterns. In brief, they made use of the meta-modeling techniques to model the structures of pattern leitmotif by using collaboration diagram to specify the collaboration among ModelElements as well as a set of OCL well-formed rules. It is not clear how a tool can support their technique. We argue, without the application of the profile mechanism, it would be hard to build the automatic support which can be easily tied with most existing UML CASE tools.

Dong and Yang [7] propose three stereotypes, each of which extends Class, Operation and Attribute metaclass in the UML metamodel. When used for a specific pattern, these stereotypes are decorated with the name of pattern elements, assuming the name is known. These approaches basically define a set of stereotypes to be used in designs for quality attributes such as extensibility and traceability. However, none of them is capable of validating the correctness of designs to the profile applied. Our approach together with the aid of the ICER tool not only supports the quality attributes, but also allows one to check the validity of a design to a profile.

A new visual modeling language called DPML [14] has been proposed to represent and apply design patterns. But since DPML introduces some new notations such as hexagon and inverted triangles, users of DPML must be familiar with these notations. Likewise, the application of design patterns also closely ties with their DPTool. However, the ICER tool can accept a model represented in the XML format so it is open to any kind of UML CASE tools.

The calculus of refinement of component and object-oriented systems, known as rCOS, is a framework [17] that supports component-based software development based on a set of refinement rules via the graph transformation [18]. Qian *et al.* [8] investigated how design patterns and refactoring rules are used in a formal method by formulating and showing them as refinement laws in rCOS.

The application of design patterns allows successful design solutions to be reused in software systems; and because they are usually applied to different development environments, the reification of design patterns is not unique. Developers should be provided with an ability to give their own reification of design patterns. In this paper, we have shown that the profile mechanism is the solution to define a reification of a design pattern. As an example, we demonstrated how to apply a profile in reifying a design pattern via the `Observer`

pattern and the `Visitor` pattern. We provide profiles for most GoF design patterns at [10], and developers are encouraged to apply these profiles as a template to software development when a different reification is required.

The profile mechanism promotes the implementation of a tool to support design patterns. Applying the `instance-of` relation, the ICER tool can check whether an application model is designed based on design patterns via the conformance checking. Since the ICER tool uses the XML format of UML models as input, it can be applied to many UML CASE tools which can export the XML format. Even for those tools which do not support some features such as OCL, users of the ICER tool can still use those tools after they manually add the missing information such as OCL constraints to the corresponding model elements via the XML tags.

Class diagrams are arguably the most widely used object-oriented modeling diagram today. So, the ICER tool supports the conformance checking of a class model and a profile when applying some design patterns. In the future, we can extend the ICER tool to read some other UML diagrams such as sequence diagrams to support the dynamic aspect of a model when design patterns are used. But our overall experience has shown the promise of the ICER tool when it provides users with the flexibility in reifying design patterns.

## References


[1] J. O. Coplien, *Software Pattern,*. New York: SIGS Management Briefings, SIGS Books, 1996.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[3] OMG UML Superstructure Specification, version 2.0, 05-07-04. Object Management Group.

[4] Jos B. Warmer, Anneke G. Kleppe, *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley, 1998.

[5] R. France, D.-K. Kim, S. Ghosh, and E. Song, "A UML-Based Pattern Specification Technique," in *IEEE TSE*, vol. 30 of *No. 3*, pp. 193–206, March 2004.

[6] M. Fontoura, W. Pree, and B. Rumpe, *The UML Profile for Framework Architectures*. Addison-Wesley, 2002.

[7] J. Dong and S. Yang, "Extending UML To Visualize Design Patterns In Class Diagrams," in *Proceedings of the Fifteenth SEKE*, (California, USA), 2003.

[8] L. Quan, Z. Qiu, and Z. Liu, "Formal use of design patterns and refactoring," in *ISoLA*, pp. 323–338, 2008.

[9] W. Shen, K. Wang, and A. Egyed, "An efficient and scalable approach to correct class model refinement," in *IEEE TSE*, vol. 35, pp. 515–533, July/August, 2009.

[10] ICER Application: GoF Design Patterns. `http://www.cs.wmich.edu/~OODA/patterns/index.html`.

[11] A. Guennec, G. Sunye, and J. Jezequel, "Precise Modeling of Design Patterns," in *Proceedings of the 3rd Inter. Conf. on (UML)*, (York, UK), pp. 482–496, Springer-Verag, LNCS 1939, 2000.

[12] A. H. Eden, *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, University of Tel Aviv, 1999.

[13] A. Lauder and S. Kent, "Previse Visual Specification of Design Patterns," in *Proceedings of the ECOOP'98*, vol. 1445 of *LNCS*, pp. 114–134, 1998.

[14] D. Mapdlsden, J. Hosking, and J. Grundy, "Design Pattern Modelling and Instantiation Using DPML," in *Proceedings of the 40th International Conference on Tools Pacific*, 2002.

[15] Y. Sanada and R. Adams, "Representing Design Patterns and Frameworks in UML, Towards a Comprehensive Approach," *Journal of Object Technology*, vol. 1, no. 2, 2002.

[16] J. K. H. Mak, C. S. T. Choy, and D. P. K. Lun, "Precise modeling of design patterns in uml," in *Proceedings of the 26th ICSE*, pp. 252 – 261, IEEE Computer Society, 2004.

[17] Z. Chen, Z. Liu, A. P. Ravn, V. Stolz, and N. Zhan, "Refinement and verification in component-based model-driven design," *Sci. Comput. Program.*, vol. 74, no. 4, pp. 168–196, 2009.

[18] L. Zhao, X. Liu, Z. Liu, and Z. Qiu, "Graph transformations for object-oriented refinement," *Formal Asp. Comput.*, vol. 21, no. 1-2, pp. 103–131, 2009.