Visualizing Design Patterns in Their Applications and Compositions

Jing Dong, *Member*, *IEEE*, Sheng Yang, *Member*, *IEEE*, and Kang Zhang, *Senior Member*, *IEEE*

Abstract—Design patterns are generic design solutions that can be applied and composed in different applications where pattern-related information is generally implicit in the Unified Modeling Language (UML) diagrams of the applications. It is unclear in which pattern instances each modeling element, such as class, attribute, and operation, participates. It is hard for a designer to find the design patterns used in an application design. Consequently, the benefits of design patterns are compromised because designers cannot communicate with each other in terms of the design patterns they used and their design decisions and trade-offs. In this paper, we present a UML profile that defines new stereotypes, tagged values, and constraints for tracing 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. Based on this profile, we also develop a Web service (tool) for explicitly visualizing design patterns in UML diagrams. With this service, users are able to visualize design patterns in their applications and compositions because pattern-related information can be dynamically displayed. A real-world case study and a comparative experiment with existing approaches are conducted to evaluate our approach.

Index Terms—Design pattern, UML, model-driven architecture, Web service, visual tool.

1 Introduction

Applying design patterns [6], [8], [19], [21], [41] in software designs support the reuse of expert design experiences to solve recurring problems. Design patterns help designers communicate architectural knowledge, help people learn a new design paradigm, and help new developers avoid traps and pitfalls that have traditionally been learned only by costly experiences. Design patterns are usually modeled and documented in natural languages and visual notations such as the Unified Modeling Language (UML) [5], [43], [52]. UML is a family of modeling notations for specifying, visualizing, constructing, and documenting artifacts of software-intensive systems. It provides a collection of visual notations to capture different aspects of the system under development.

Each design pattern normally contains several participants such as classes, attributes, and operations, which play certain roles manifested by their names. When the design pattern is applied or composed in an application, the role names of its participants may be adapted to reflect the application domain. Thus, pattern-related information, represented by the role names, is lost. It is hard to identify in which patterns a modeling element, such as class, attribute, and operation, participates in an application design. The designers are not able to trace this information in the application design. For instance, Fig. 1 shows a software system design containing six design patterns: Abstract Factory, Session Facade, Business Delegate, Service Locator, Singleton, and Adapter. It is hard to identify the participants of each design pattern in this diagram because all pattern-related information is implicit. 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 because they do not have pattern-related information in system designs. Second, each pattern often documents some ways for future evolutions [2], which are buried in the system design. The designers are not able to change the design using relevant pattern-related information. Third, each pattern may preserve some properties and constraints. It is hard for the designers to check whether these properties and constraints hold when the design is changed. Fourth, it may require considerable effort on reverse-engineering design patterns from software systems [24], [26], [28], [29], [39].

As the de facto standard for object-oriented modeling, the UML is defined within a general four-layer metamodeling architecture: metametamodel, metamodel, model, and user objects. The metametamodel layer defines a language for specifying the metamodel layer. The metamodel layer, in turn, defines a language for specifying the model layer. Similarly, the model layer is used to define models of specific software systems. The user objects layer is used to define software systems of a given model. For example, the UML metamodel defines all legal UML specifications. The UML model defines a model of software systems that may be instantiated into user objects. The metamodel and model layers are most relevant to modeling design patterns in UML. A UML profile may be used to define an extension to the UML at the metamodel level.

In this paper, we present a UML profile for design patterns, which extends the UML with new stereotypes,

• *The authors are with the Department of Computer Science, University of Texas at Dallas, 2601 North Floyd Road, Richardson, TX 75083. E-mail: {jdong, syang, kzhang}@utdallas.edu.*

Manuscript received 28 May 2006; revised 20 Dec. 2006; accepted 15 Mar. 2007; published online 9 Apr. 2007.
Recommended for acceptance by B. Cheng.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0115-0506.
Digital Object Identifier no. 10.1109/TSE.2007.1012.

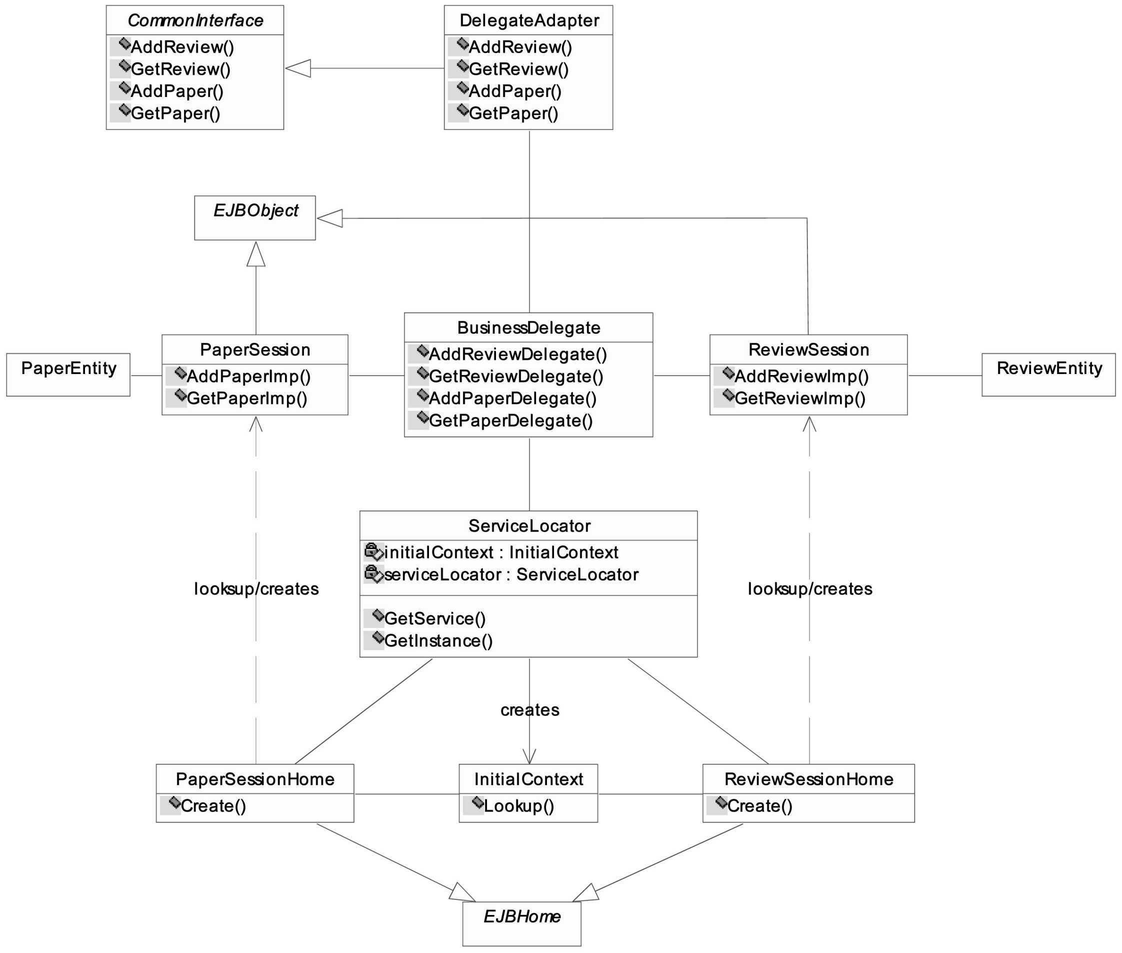


Fig. 1. Conference manager system design.

tagged values, and constraints to explicitly visualize pattern-related information. The UML extension is defined in the UML metamodel. In this way, the role each modeling element plays in a design pattern can be represented statically using the UML profile. In addition, we provide a tool that can be used to dynamically visualize pattern-related information. This tool is deployed as a Web service for on-demand visualization and can be seamlessly integrated with UML tools such as Rational Rose [53]. A case study on a real-world system is conducted to illustrate the features and evaluate the scalability of our approach. In addition, an experiment is performed to compare the graphic complexity metrics of different existing approaches.

The remainder of this paper is organized as follows: In the next section, we detail the problem of missing pattern-related information and some current solutions. In Section 3, we discuss the related work. Section 4 presents our proposed extension of UML with a profile. In Section 5, we describe our techniques for on-demand visualization and its usage as a Web service. We conduct a comparative study using different graphic complexity metrics in Section 6. In Section 7, we use a case study to illustrate the usage of proposed extension and on-demand visualization and evaluate the scalability of our approach. In the last section, we conclude this paper and discuss the future work.

2 Background and Motivating Examples

Consider Fig. 1 as an example of a software system design. This system manages peer-reviewed conferences and workshops, including paper/abstract submission, committee formation, paper reviews, online technical committee meetings, report compilations, author notification, preliminary conference program creation, and so forth. There are multiple types of users in this system: author, reviewer, conference chair, and administrator. Authors can submit papers to a conference and read the reviews of their papers. Reviewers may submit reviews for different papers. The program chair manages the conference, assigns reviewers to papers, and makes the final decision on the acceptance or rejection of the papers. The system may also allow a third party to batch upload papers and review through the provided interfaces. In this partial design of the system, there are six design patterns: Abstract Factory, Session Facade, Business Delegate, Service Locator, Singleton, and Adapter, including two instances of the Session Facade pattern. The participants of each design pattern are normally manifested by their role names, which have been changed to fit the application domain. It is hard for designers to identify each design pattern instance in this diagram. Therefore, the benefits of design patterns are compromised.

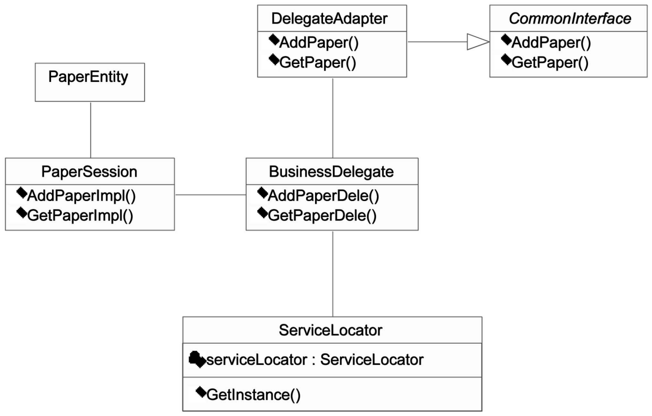


Fig. 2. The composition of the Business Delegate, Session Facade, and Adapter patterns.

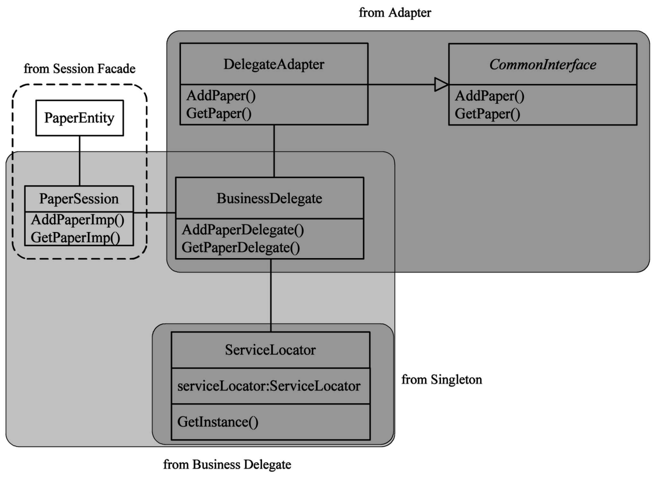


Fig. 3. Venn diagram-style pattern annotation.

Several approaches have been proposed to solve this problem [5], [20], [33], [48]. To illustrate these approaches, we consider a part of this design, as shown in Fig. 2, which include four design patterns: Business Delegate [3], Session Facade [3], Singleton [21], and Adapter [21], applied in the system design.

The ServiceLocator, BusinessDelegate, and PaperSession classes participate in the Business Delegate pattern. The goal of the Business Delegate pattern is to abstract the business of the client side and hide the implementation of business services. It reduces the coupling between the client and the system business services. In particular, the BusinessDelegate class provides control and protection for the business services. The ServiceLocator class provides the API lookup (naming) services for the class BusinessDelegate to locate the business service.

The PaperSession and PaperEntity classes participate in the Session Facade pattern that abstracts the underlying business object interactions and provides a service layer that exposes only the required interface. In the example shown in Fig. 2, the PaperSession class controls the interactions between the client and the participating business data and business objects. It provides a coarse-grained method to perform the required business function. The PaperEntity class is used to represent the business data and manage the data to ensure its integrity.

The BusinessDelegate, DelegateAdapter, and CommonInterface classes participate in the Adapter pattern that can convert the interface of a class in the interface that its client expects. The DelegateAdapter and CommonInterface classes wrap the BusinessDelegate class and provide the uniform interface to the third party.

The ServiceLocator class is also a Singleton in the Singleton pattern. It ensures that only one instance of ServiceLocator is available at any time.

In an intuitive approach based on Venn diagrams [48], all participating classes are bounded with a particular shade of color for each pattern, as shown in Fig. 3. This annotation for visualizing patterns works fine with small systems. When the system becomes bigger, especially when a class participates in multiple patterns, the overlapping area becomes hard to distinguish. In addition, it is unclear what role each modeling element, such as class, attribute, and operation, plays in the corresponding patterns. The approach only identifies the boundary of each design pattern.

The UML provides a collaboration annotation by attaching dashed ellipses to the general UML diagrams. Collaboration annotation can be used to highlight design patterns. Dashed lines attached with participant names are used to associate the patterns with their participating classes. As shown in Fig. 4, there are four dashed ellipses representing four design patterns. All the participants are connected to the corresponding dashed ellipses. Compared to Venn-style annotation, this annotation can explicitly represent the participant role a class plays. However, the roles that an operation (attribute) plays in a pattern are still not addressed. Moreover, the dashed lines clutter the representation. The pattern information and the class structure are mixed and hard to distinguish. It is also hard to scale up.

To improve the diagrammatic presentation by removing the cluttering dashed lines, a graphical notation, called "pattern:role annotations" has been proposed [48]. The idea is to tag each class with a shaded box containing the pattern and/or participant name(s) associated with the given class. If it will not cause any ambiguity, only the participant name is shown for simplicity. Fig. 5 shows that the pattern-related annotations appear in shaded boxes as if they are on a different plane from the class structure. This notation is more scalable than the previous notations and highly readable and informative [48]. However, not only a class but also an operation or attribute may play some roles in some design patterns. This notation cannot represent the roles that an operation (attribute) plays in a design pattern. Even though the approach may scale better than previous approaches, the additional note boxes (with pattern-related information) still increase the size of the original UML diagrams. There also can be multiple instances of the same design pattern in a system design, which cannot be distinguished by this approach. Moreover, the problems related to shading arise. The gray backgrounds do not fax and scan well as identified as a limitation of the approach [48]. In addition, they may not print well in some printers with low resolution because the gray backgrounds can make the words inside the shaded box illegible.

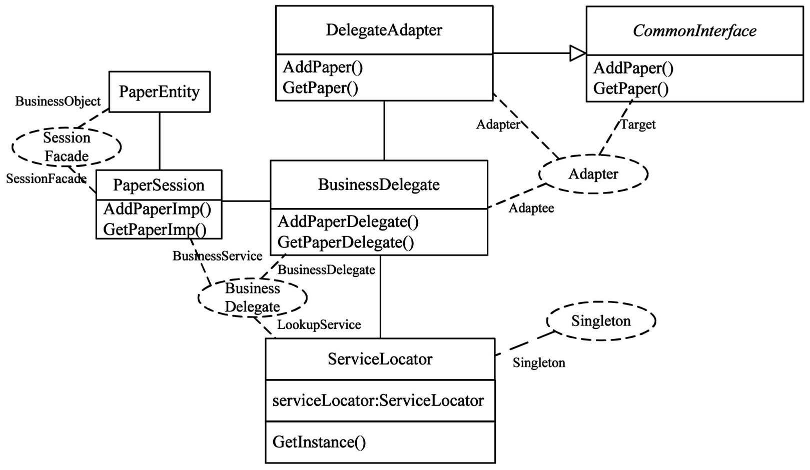


Fig. 4. UML collaboration annotation.

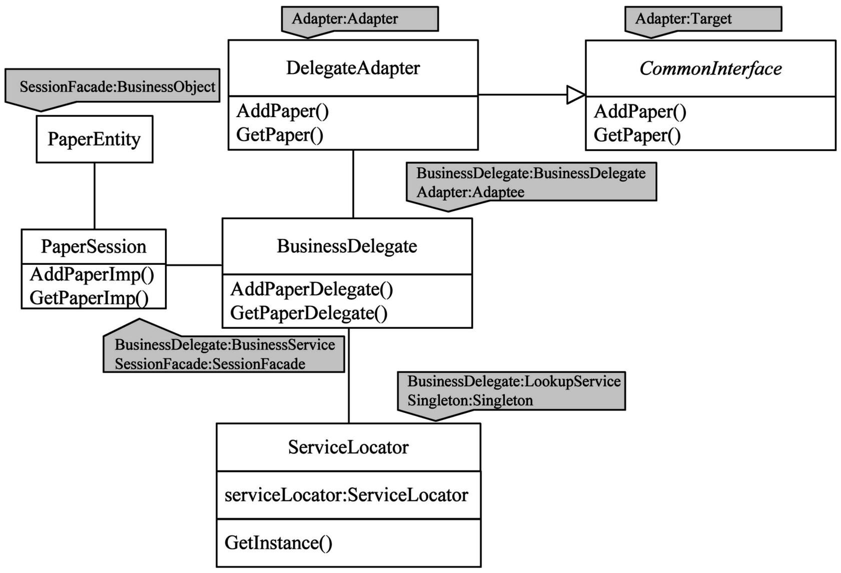


Fig. 5. Pattern:role annotation.

In summary, all existing solutions tend to attach static notations and/or textural information to UML diagrams, which may inflate the original diagram with pattern-related information. They cannot distinguish different instances of the same design pattern. Most of these solutions cannot attach information to attributes/operations. They only addressed the issues in class diagrams instead of behavioral diagrams, such as collaboration or sequence diagrams, which are typically important part of design pattern descriptions. In this paper, therefore, we plan to achieve the following goals:

- providing a scalable approach with the graphic complexity close to that of the original UML diagram;
- representing the roles an operation/attribute plays in addition to the roles a class plays in a design pattern;
- providing mechanisms to distinguish different instances of the same design pattern in a UML diagram;
- supplying tool support that is compatible with commonly used UML tools, such as Rational Rose [53] and ArgoUML [51]; and
- allowing the user to concentrate on a particular part of a large diagram.

In the remainder of this paper, we first discuss the related work. We then introduce our approach on a UML profile for design patterns. We describe a tool, called VisDP, which can dynamically display pattern-related information in UML diagrams. Finally, we conduct a spatial measurement comparison and a case study to evaluate our techniques and tool.

3 Related Work

Explicitly visualizing design patterns in UML has been investigated in [48], where all approaches surveyed can only represent the role a class plays in a pattern, not the roles of an attribute (or operation). They cannot distinguish multi-instances of a pattern either. These approaches provided new notations and/or text information attached to the UML diagrams to represent pattern-related information. Nevertheless, the additional information is static and does not scale well. In this paper, we presented a UML profile that can represent the role an attribute (operation), as well as a class, plays in a design pattern and distinguish multi-instances of a design pattern. Our approach can be seamlessly integrated with the UML standards. We propose to dynamically visualize design patterns to overcome the scalability problem of all the static approaches. In this way, we hide all pattern-related information and allow the user to visualize design patterns on demand. All pattern-related information is displayed only when requested.

France et al. [20] specialized the UML metamodel to obtain a pattern specification. Pattern-related information is defined as roles in subtypes of UML metamodel in a separate diagram. Thus, the generic specifications of each design pattern are represented in a separate diagram from the applications of the corresponding pattern. The application of a design pattern is mapped to its generic specifications by dotted lines with arrow heads. The user is able to visualize pattern-related information from these dotted lines. In this case, pattern specifications always need to be presented for the user to visualize pattern-related information in a UML diagram. The pattern specification diagram and the dotted lines bound to the corresponding pattern application are static information added on normal UML diagram. The main goal of the approach is to find a practical way for specifying design patterns. Our goal, on the other hand, focuses on providing a scalable solution on visualizing design patterns. We also take into account the composition of patterns and multi-instance of patterns. We consider our approaches to complement those of France et al.

Reiss [42] proposed a specification language for defining design patterns that breaks a design pattern down into elements and constraints over a database storing the structural and semantic information of a program. Each system has a database to store the design patterns defined in this language. Design pattern instances can be created, found, maintained, and edited by querying the database. Based on this language, a tool is developed to allow the user to identify and create pattern instances in the source code. The objective of his approach is to facilitate the application and discovery of design patterns, instead of visualizing them based on their generic specifications.

Logic-based languages have been presented to express unambiguously the solutions proposed by design patterns [1], [16], [37]. Visual notations consisting of icons (ovals, triangles, and squares) are proposed to make the formal languages more accessible to novice users [16]. The main goal of these formal approaches is to reduce ambiguity in the specification of design patterns in informal languages, not to display instances.

General design recovery frameworks have been investigated [22], [23]. Tool support for applying (forward engineering) and discovering (reverse engineering) design patterns has also been developed [17]. This tool is based on the fragment model and fragment database. Although the fragment structure diagram can be used to visualize the role each class (attribute or operation) plays in a design pattern, it does not keep the topology of the original UML diagram, and so the class model information is lost. The tool cannot visualize a program purely in terms of pattern instances.

Lander and Kent [31] propose an approach to specifying a design pattern in type model and role model in addition to class model to tackle the impure pattern modeling problem due to the difficulty of expressing nondeterministic number of concrete classes using UML. When several patterns are composed, the three models and the mapping among them become very complex. The main goal of their work is to provide a more expressive approach to specifying design patterns although their approach can trace design pattern from type/role model to class model. Instead of introducing new types of diagrams, the goals of our approach are scalability, practice, and easy to learn and use.

Design Pattern Modelling Language (DPML) has been proposed to model and apply design patterns [33]. In contrast to our approach that constraints UML, DPML provides new notations such as hexagon and inverted triangle for modeling design patterns. It improves the expressiveness for some special concepts, for example, dimension, of design patterns. A tool is also provided to draw the new notations. Unlike DPML, our goal is to explicitly visualize the hidden dependency of a design pattern to its instance link. We also support design pattern composition, dynamic aspects of design patterns, object constraint language, and the overlapping of design pattern instances.

Several limitations of using the UML parameterized collaboration diagram to specify and apply design patterns are discussed in [47]. A tool has been provided for the generation and reconstruction of design patterns to overcome the limitations. In addition, two stereotypes, <<Clan>> and <<Tribe>> have been defined to model recurring constraints of design patterns [32]. The <<meta>> stereotype is defined with some well-formedness rules in OCL to improve the graphic representation of pattern occurrences. Unlike their goals (generation and reconstruction), our approach intends to visualize and recognize design patterns in their applications.

UML extension mechanisms have been used to expand the expressive power of UML to model object-oriented framework [18], software architecture [27], [35], and agent-oriented systems [49] 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 and tagged values and provide the constraints applied to these stereotypes and tagged values.

UML has been used to visually specify interactive multimedia application [44] and support for dynamic modeling [25] with an extension for behavioral specifications. The concept of visual scripting in VISOME [40] is interesting. The scripting mechanism supports the visual composition of high-level functions for software modeling and assists automatic synthesis of different UML diagrams. The usability of UML tools can also be enhanced by adding speech recognition capability [30]. None of these approaches, however, address the visualization issues of design patterns in UML.

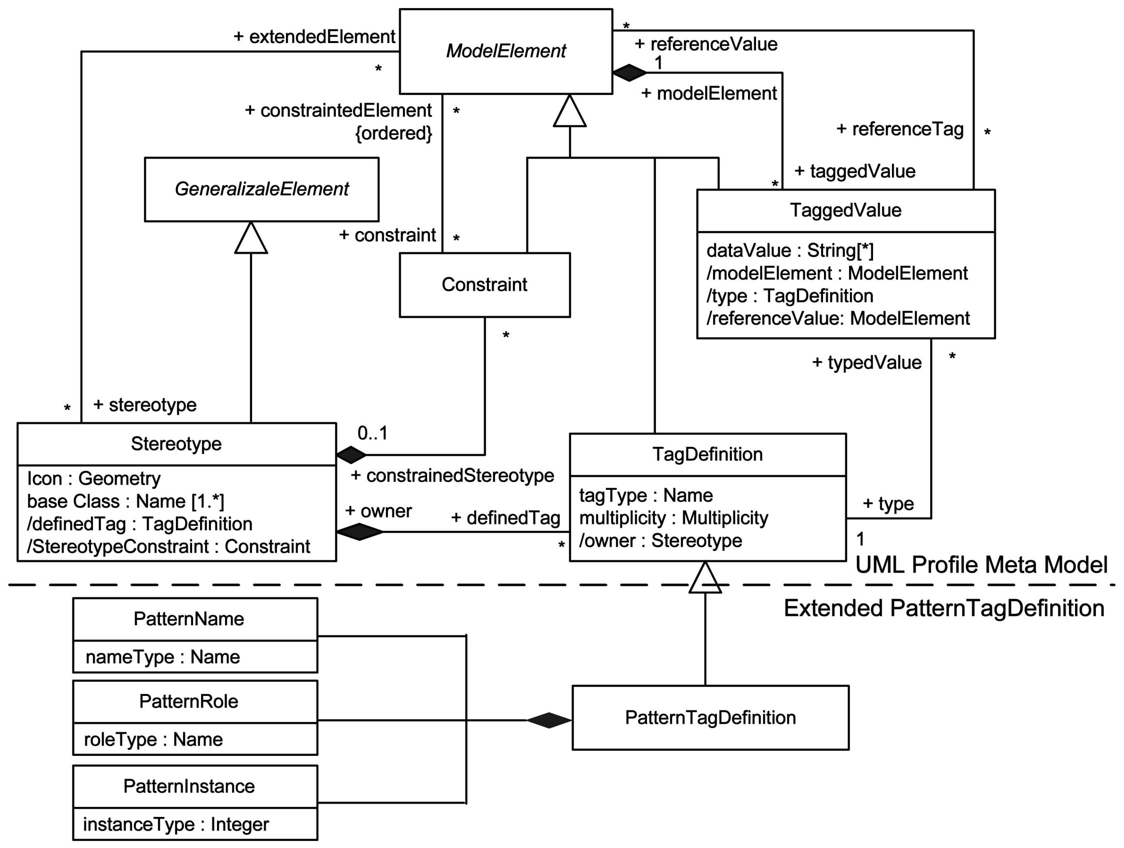


Fig. 6. UML profile metamodel and our extension.

To our knowledge, there has been no reactive user interface similar to ours for visualizing design patterns. Dynamic user interactive techniques have been used for information visualization. An example is the hybrid network visualizing software development communities [36]. Due to the large number of nodes representing the names of developers and edges representing their communications, the network connection is hidden by default. Relevant communication details such as e-mail messages are highlighted or displayed in popup windows.

Tightly integrated into the Eclipse IDE, SEXTANT [45] is a prototype for software exploration with graph-based visualization and navigation support. The user can hide irrelevant information during exploration. The visualization and navigation support is, however, not designed for dynamic visualization of design patterns in class diagrams.

4 Extending UML to Visualize Design Patterns

This section presents our approach for explicit visualization of design patterns in their applications and compositions through the same example used in Section 2. We introduce the UML extensions including new stereotypes, tagged values, and constraints, and present a general description of their semantics. We also discuss how the UML extensibility mechanisms have been applied in the definition of a UML profile for design patterns.

4.1 UML Profile Metamodel

UML can be considered as a multipurpose language with many notational constructs. Fig. 6 shows the UML metamodel that defines all legal UML specifications for our UML profile.[1] The UML provides extension mechanisms to allow users to model software systems if the current UML technique is not semantically sufficient to express the systems. These extension mechanisms are defined at the UML metamodel and model levels. They include stereotypes, tagged values, and constraints.

Stereotypes allow the definition of extensions to the UML vocabulary, denoted by <<*stereotype-name*>>. As shown in Fig. 6, the base class of a stereotype must match the metaclass of that model element such as Class, Attribute, and Operation. A *stereotype* groups tagged values and constraints under a meaningful name. When a stereotype is branded to a model element, the semantics of the tagged values and the constraints associated with the stereotype are attached to that model element implicitly. A number of possible uses of stereotypes have been proposed [4].

Tagged values extend model elements with new kinds of properties. They may be attached to a stereotype, and such an association will propagate to the model element to which the stereotype is branded. The format of a tagged value is a pair of name and an associated value, i.e., $\{\text{name} = \text{value}\}$. The tagged values attached to a stereotype must be compatible with the constraints of the stereotype's base class.

1. Our specifications are based on UML 1.4.

Constraints add new semantic restrictions to a model element. Typically, constraints are written in the Object Constraint Language (OCL) [50]. Constraints attached to a stereotype imply that all model elements branded by that stereotype must obey the semantic restrictions of the constraints. The constraints attached to a stereotyped model element must be compatible with the constraints of the stereotype and the base class of the model element.

A profile is a stereotyped package containing model elements that have been customized for a specific domain or purpose by extending the metamodel using stereotypes, tagged values, and constraints. A profile may specify model libraries on which it depends and the metamodel subset that it extends.

Fig. 6 shows the relationships among stereotype, constraint, tagged value, and model element. The stereotype, constraint, and tagged value can be attached to a model element and add corresponding semantics to restrict the model element. A stereotype may include a number of tagged values and constraints whose semantics is propagated from the stereotype to the model elements to which the stereotype is branded. In our approach, we propose a new tag definition called PatternTagDefinition, which is inherited from TagDefinition and consists of three parts, PatternName, PatternRole, and PatternInstance. PatternName has a type of Name, and so does PatternRole. PatternInstance has a type of integer. The part above the dashed line in Fig. 6 is from the UML metamodel, whereas the lower part is our new tag definition. There are several reasons for us to define a new kind of tag (PatternTagDefinition) instead of reusing the existing one (TagDefinition). First, we need to restrict the general definition of tags. This special type of tags is only used to define the role, pattern name, and instance, which are important for visualization. Second, for the usual TagDefinition, the name of taggedValue is clearly defined; the variation point is the value of taggedValue. For our new tag definition, the names of taggedValue themselves are changeable, which may be different from class to class. Third, introducing this new tag also facilitates our tool support.

4.2 A UML Profile for Design Patterns

We considered three options to extend the UML to explicitly visualize pattern-related information usually hidden in UML diagrams. The first option is to define three stereotypes: PatternClass, PatternAttribute, and PatternOperation, whose base classes are Class, Attribute, and Operation, respectively. There is one tagged value defined in each stereotype in the first option. The name of the tagged value is "pattern" and the dataValue 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 "instance" in the tuple indicates the instance number of the pattern the model element participates. It can be omitted if there is only one instance of the design pattern in the system design. The "role" in the tuple shows the role that a model element plays in the pattern.

The second option is also to define three stereotypes: PatternClass, PatternAttribute, and PatternOperation, whose base classes are Class, Attribute, and Operation, respectively. There is one tagged value defined for each stereotype. In contrast to the first option, the name of the tagged value has the format of "role@name[instance]" and the dataValue of the tagged value is either true or false. The meanings of fields "name," "instance," and "role" are the same as those in the first option. If the dataValue of a tagged value is true, the value can be omitted and only the name of the tagged value is shown in a diagram. If this tagged value is attached to a modeling element, it indicates that the modeling element participates in the "name" pattern with the role of "role." The value of "instance" distinguishes different instances of the same design pattern, which can be omitted if there is only one instance of the design pattern in the system design. The main difference between these two options is that the first option defines the name of the tagged value as "pattern" and the value of the tagged value as <name: string [instance: integer], role: string> whereas the second option defines the name of the tagged value as "role@name[instance]," and the dataValue of the tagged value is either true or false. The dataValue of a tagged value can be omitted when it is true, that is, $\{\mathrm{name} = \mathrm{true}\}$ can be shorthanded to {name}. In contrast, the name of a tagged value cannot be omitted.

Similarly to the previous two options, the third option also defines three stereotypes: PatternClass, PatternAttribute, and PatternOperation with base classes Class, Attribute, and Operation, respectively. In contrast to the other two options, each stereotype may have three tagged values. The first tagged value defines the tagType as PatternName and the dataValue storing the name of the pattern in which the model element participates. The second tagged value defines the tagType as PatternRole and the dataValue storing the role that the model element plays. The third tagged value defines the tagType as PatternInstance and the dataValue with the index value of a pattern instance. Although this option may clearly specify the PatternName, PatternRole, and PatternInstance with their values, it requires tagTypes in every tagged value that enlarge the specification. When a modeling element participates in multiple patterns, in addition, the designer must be careful not to mix up the PatternName, PatternRole, and PatternInstance tagged values of different patterns because the order of the tagged values of the same stereotype does not matter. For example, suppose a stereotype has the tagged values {PatternName = Adapter} {PatternRole = Adaptee} {PatternName = Bridge} {Pattern Role = Abstraction}. A simple disorder of these tagged values by a designer or a tool may cause trouble in understanding to which pattern a role belongs. Moreover, it is relatively difficult to implement this option in a tool due to the order issue.

There are several advantages of the second option. First, the other two options may result in larger specifications because the name of a tagged value cannot be omitted. Pattern-related information should be minimized in class diagrams for readability and scalability. Second, the first option is not supported by commonly used UML tools such as Rational Rose. Third, the third option has an issue with the order of the tagged values. Therefore, we choose the

TABLE 1
Stereotypes

Stereotype	Applies To	Definition
<<PatternClass>>	Class	Indicate that this class is a part of a design pattern
<<PatternAttribute>>	Attribute	Indicate that this attribute is a part of a design pattern
<<PatternOperation>>	Operation	Indicate that this operation is a part of a design pattern

second option to define the complete set of the UML profile, the stereotypes, the tagged values, the constraints, and the virtual metamodel (VMM).

We define the three stereotypes, PatternClass, PatternAttribute, and PatternOperation (see Table 1) to explicitly visualize design patterns in UML diagrams. The PatternClass stereotype is attached to a class that plays a role in a design pattern. Similarly, the PatternAttribute and PatternOperation stereotypes are attached to an attribute and an operation, respectively, which plays a certain role in a design pattern. Each stereotype also defines one tagged value, as shown in Table 2. These tagged values define the exact role of a class, an attribute, or an operation in a design pattern. These stereotypes, together with their tagged values and constraints, form a new UML profile for design patterns. Sections 4.3 and 4.4 provides detailed semantics of these stereotypes and of their tagged values and constraints.

4.3 Semantics

The PatternClass stereotype is defined to be applied to a class that plays a role in a design pattern. The particular role this class plays is defined in the tagged value in the format "role@name[instance]," where "name" specifies the name of the design pattern, "instance" specifies to which instance of the pattern the class belongs, and "role" specifies the role the class plays in the pattern. The "name," "instance," and "role" are instances of PatternName, PatternInstance, and PatternRole defined in Fig. 6, respectively. For instance, the BusinessDelegate class plays the role of Adaptee in the Adapter pattern in the example shown in Fig. 2. Then, the stereotype <<PatternClass{Adaptee@Adapter}>> is branded to the BusinessDelegate class, where the branded class participates in the Adapter pattern and plays the role of Adaptee in the pattern. The instance field is omitted because there is only one instance of the Adapter pattern in this design.

A class may simultaneously play different roles in different patterns. In this case, a new tagged value with the same format as "role@name[instance]" is branded to the class for each additional pattern it participates. For instance, the BusinessDelegate class also plays the role of BusinessDelegate in the Business Delegate pattern in the example shown in Fig. 2. Thus, the stereotype <<PatternClass{Business Delegate@BusinessDelegate}{Adaptee@Adapter}>> is branded to the BusinessDelagate class, where the branded class participates in the Business Delegate pattern with the role of BusinessDelegate and the Adapter pattern with the role of Adaptee.

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 = "role@name[instance]" and
tv.dataValue = Boolean),
```

where the base class of this stereotype is Class, and the stereotype has a tagged value with the name of "role@ name[instance]" and the value of true/false. The types of "name" and "role" are string, and the type of "instance" is integer.

The constraints of the PatternAttribute and PatternOperation stereotypes are defined similarly and are branded to an attribute and operation, respectively, when they play a role in a design pattern. These constraints are inherited from the corresponding stereotype and represent the properties of stereotypes. Section 4.4 defines the constraints restricted by design patterns. These constraints represent the properties of design patterns, instead of the properties of stereotypes as shown previously.

4.4 Constraints

Figs. 7 and 8 show the pattern-related constraints for stereotypes <<PatternClass>>, <<PatternAttribute>>, and <<PatternOperation>>, respectively.

In Fig. 7 (the constraints of <<PatternClass>>), the first and second constraints indicate that the name field and role field of the tagged values associated with <<PatternClass>> are mandatory, respectively. Thus, these fields cannot be empty. The third constraint indicates that the instance field of the tagged values associated with <<PatternClass>> can be empty if there is only one instance of a particular design pattern in a system design. If there are multiple instances of

TABLE 2
Tagged Values

Tagged Value	Applies To	Definition
role@name [instance]	<<PatternClass>>	Indicate that the attached class plays the role of *role* in the *instance* of a design pattern named *name*
role@name [instance]	<<PatternAttribute>>	Indicate that the attached attribute plays the role of *role* in the *instance* of a design pattern named *name*
role@name [instance]	<<PatternOperation>>	Indicate that the attached operation plays the role of *role* in the *instance* of a design pattern named *name*

```
1> self.taggedValue.dataValue.name -> notEmpty
2> self.taggedValue.name.role -> notEmpty
3> self.taggedValue.name.instance -> isEmpty or self.taggedValue -> exists (tv:taggedValue | tv.name.instance -> notEmpty)
4> self.taggedValue.name -> exists (v1, v2:name | v1.name = v2.name ) implies (v1.instance -> notEmpty and v2.instance ->
   notEmpty and v1.instance <> v2.instance)
```

Fig. 7. Constraints for stereotype PatternClass.

```
1> self.taggedValue->exists(tv:taggedValue,pc:PatternClass | tv.name.name = pc.taggedValue.name.name)
2> self.taggedValue->size <= PatternClass.taggedValue.dataValue ->size
3> self.taggedValue.name.role -> notEmpty
4> self.taggedValue.name.name -> isEmpty implies PatternClass.taggedValue.dataValue ->size = 1
5> self.taggedValue.name.name -> isEmpty implies self.taggedValue.name.name = PatternClass.taggedValue. name.name
6> PatternClass.taggedValue.name.instance -> isEmpty implies self.taggedValue.name.instance -> isEmpty
7> (self.taggedValue.name.instance -> isEmpty and PatternClass.taggedValue.name.instance -> notEmpty) implies
   self.taggedValue.name.instance = PatternClass.taggedValue.name.instance
```

Fig. 8. Constraints for stereotypes PatternAttribute and PatternOperation.

a certain design pattern, however, the instance field cannot be empty, as defined in the fourth constraint.

In the constraints of <<PatternAttribute>> and <<PatternOperation>> (Fig. 8), the first constraint indicates that all pattern names found in the <<PatternAttribute>> or <<PatternOperation>> stereotype should also be in the <<PatternClass>> stereotype. If an attribute participates in a particular design pattern, its class should also participate in the same design pattern. The second constraint shows that the number of patterns in the <<PatternAttribute>> or <<PatternOperation>> stereotypes must be less than or equal to that of <<PatternClass>>. It is often the case that a class plays multiple roles in several patterns. However, one attribute may not participate in all patterns in which its class participates. Sometimes, an attribute of a class plays roles in some patterns; other attributes of the same class may play roles in other patterns. The third constraint defines that the role field of the tagged values associated with <<PatternAttribute>> or <<PatternOperation>> is mandatory. The fourth constraint states that the name field of the tagged values associated with <<PatternAttribute>> or <<PatternOperation>> can be omitted if its class only participates in one design pattern. If the name field of <<PatternAttribute>> or <<PatternOperation>> is omitted, <<PatternAttribute>> or <<PatternOperation>> has the same name as <<PatternClass>>, which is stated in the fifth constraint. The instance field of the tagged values associated with <<PatternAttribute>> or <<PatternOperation>> can be empty if the instance field of <<PatternClass>> is empty as stated in the sixth constraint. If the instance field is empty, <<PatternAttribute>> or <<PatternOperation>> has the same instance as <<PatternClass>>, which is defined in the seventh constraint.

4.5 Virtual Metamodel

A virtual metamodel (VMM) is the UML expression of a formal model with a set of UML extensions. The VMM for the newly defined extensions is represented as a set of class diagrams. A VMM can graphically represent the relationship among the newly defined elements (PatternClass, PatternOperation, and PatternAttribute) and those defined by the UML specification (Class, Operation, and Attribute), which gives a clear picture of the relationships between the newly defined elements and those in the UML. It can also define the relationships between the new stereotypes and tagged values.

The VMM represents a Stereotype as a Class stereotyped <<stereotype>> (as shown in Fig. 9). The VMM represents a tagged value 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 the "<" and ">" symbols, indicating a Boolean. The multiplicity ([1..*]) following the Attribute name indicates that the tagged value may have one or more values.

After applying the new UML extensions to the example design shown in Fig. 2, the new class diagram of the system design with pattern-related information represented in the corresponding stereotypes is shown in Fig. 10. From this diagram, we can identify four design patterns and their participants. For example, from the stereotype branded to the ServiceLocator class, that is, <<PatternClass{LookupService@ServiceLocator [1]}{Singleton@Singleton [1]}>>, we may conclude that ServiceLocator participates in two design patterns, the Service Locator and Singleton patterns. It plays the role of LookupService in the first instance of the Service Locator pattern and the role of Singleton in the first instance of the Singleton pattern. There is only one instance of the Service Locator and Singleton pattern.

5 On-Demand Visualization

As presented in the previous section, we extend the UML with a profile. We attach newly defined stereotypes and

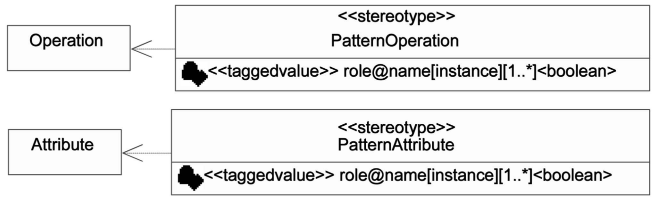
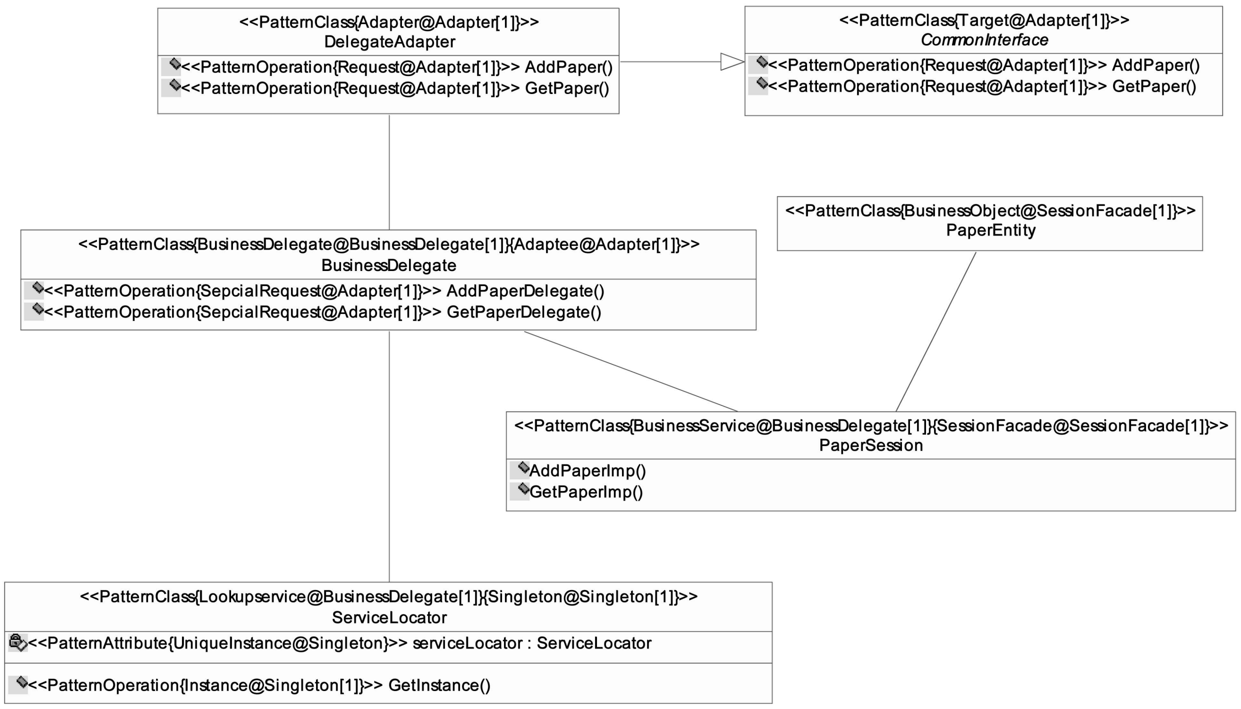

Class
<<stereotype>>
PatternClass
<<taggedvalue>> role@name[instance][1..*]<boolean>
Operation
<<stereotype>>
PatternOperation
<<taggedvalue>> role@name[instance][1..*]<boolean>
Attribute
<<stereotype>>
PatternAttribute
<<taggedvalue>> role@name[instance][1..*]<boolean>

Fig. 9. Virtual metamodel.



<<PatternClass{Adapter@Adapter[1]}>>
DelegateAdapter
<<PatternOperation{Request@Adapter[1]}>> AddPaper()
<<PatternOperation{Request@Adapter[1]}>> GetPaper()
<<PatternClass{Target@Adapter[1]}>>
CommonInterface
<<PatternOperation{Request@Adapter[1]}>> AddPaper()
<<PatternOperation{Request@Adapter[1]}>> GetPaper()
<<PatternClass{BusinessObject@SessionFacade[1]}>>
PaperEntity
<<PatternClass{BusinessDelegate@BusinessDelegate[1]}{Adaptee@Adapter[1]}>>
BusinessDelegate
<<PatternOperation{SepcialRequest@Adapter[1]}>> AddPaperDelegate()
<<PatternOperation{SepcialRequest@Adapter[1]}>> GetPaperDelegate()
<<PatternClass{BusinessService@BusinessDelegate[1]}{SessionFacade@SessionFacade[1]}>>
PaperSession
AddPaperImp()
GetPaperImp()
<<PatternClass{Lookupservice@BusinessDelegate[1]}{Singleton@Singleton[1]}>>
ServiceLocator
<<PatternAttribute{UniqueInstance@Singleton}>> serviceLocator : ServiceLocator
<<PatternOperation{Instance@Singleton[1]}>> GetInstance()

Fig. 10. Composition of Business Delegate, Session Facade, and Adapter modeled with the new UML profile.

tagged values to UML diagrams to include pattern-related information. The additional pattern-related information does not scale well with large applications as shown in Fig. 10. Even though the system is not very big, we can see that attaching the pattern-related information onto a class diagram dramatically increases the size of the class diagram. Like all current approaches, our approach attaches static information (notations) to the UML diagrams. Thus, no matter how careful the static information or notations are selected, the scalability problem would eventually arise because additional notations are needed in the UML diagrams. In addition, most of the static approaches tangle pattern-related information with the class structure, making both pattern and class harder to see. Although some approaches, such as [31], propose to separate pattern-related information from the class structure into different diagrams, the connections, consistencies and traceability among these diagrams can be challenging. Furthermore, it is not easy for the user to zoom in and concentrate on a particular part of a diagram. We propose on-demand visualization techniques based on coloring and mouse movement to solve these problems. We have developed a tool, called VisDP [14], which can hide/show pattern-related information on demand. In this way, we can dynamically, instead of statically, show the stereotypes with pattern-related information. A design pattern typically contains both structural and behavioral aspects, which are modeled in UML class and collaboration diagrams, respectively. In the following, we introduce our dynamic visualization techniques from structural and behavioral aspects.

5.1 Structural Visualization

When the pattern-related information is hidden, our diagrams are just like the ordinary UML diagrams in VisDP. For example, Fig. 11 shows a screen shot of VisDP with the UML diagram of the example described in Section 2. It is a normal UML diagram with all pattern-related information hidden.

When the pattern-related information is shown with different colors and on-demand information, the user can identify the design patterns in which a class (operation/attribute) participates and the roles it plays. By using our techniques and tool with a UML diagram, the user can move her cursor onto the modeling element (for example,

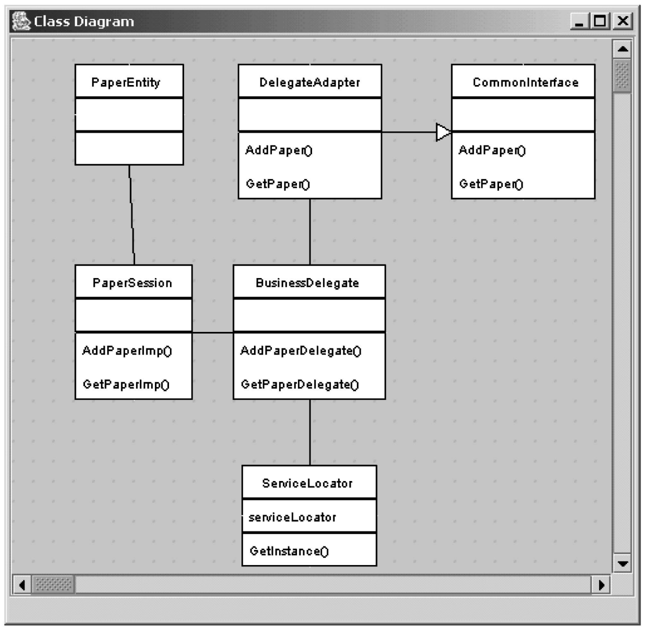
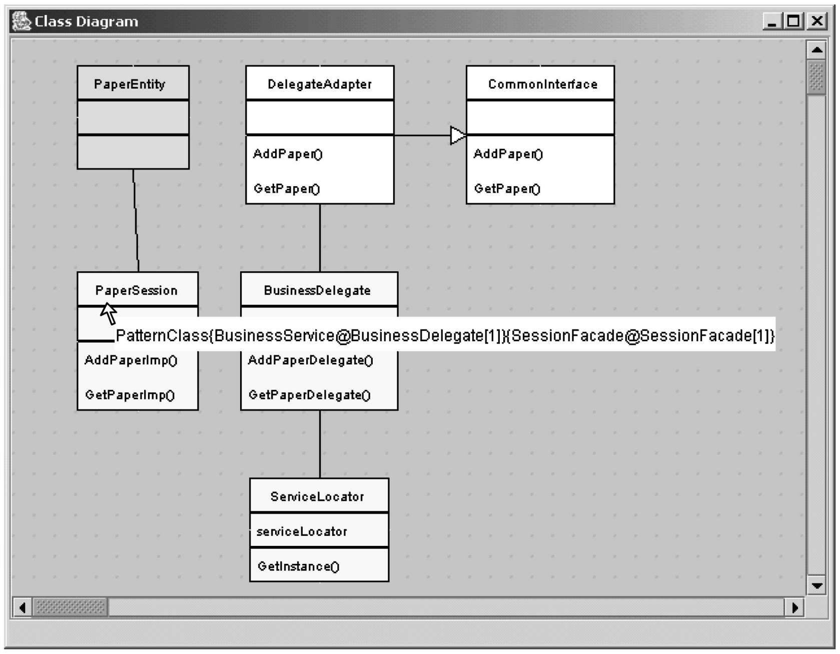


Fig. 11. Composition of Business Delegate, Session Facade, and Adapter with pattern information hidden.

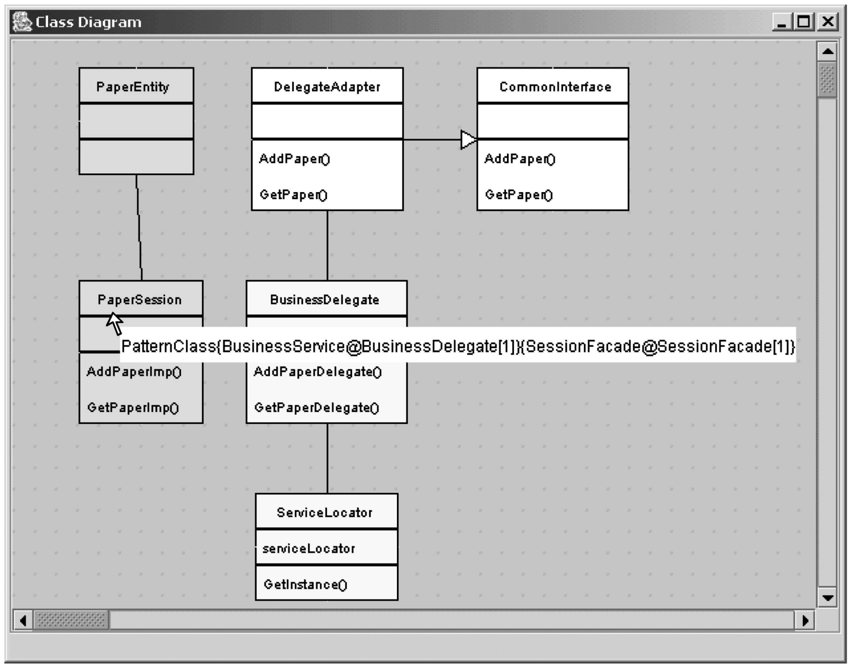


PatternClass{BusinessService@BusinessDelegate[1]}{SessionFacade@SessionFacade[1]}

Fig. 12. BusinessDelegate class diagram with pattern information shown.

class, operation, and attribute) in question. All the classes that participate in the same pattern under the cursor are changed to the same color. If the class under the cursor participates in more than one design patterns, different colors are used to distinguish the patterns. All pattern-related information encoded in the stereotypes and tagged values of a modeling element is shown when the user moves the cursor over that modeling element. If the cursor is over a class that participates in more than one pattern, different colors are displayed in alternation on this class. When the user's cursor moves out, all pattern-related information is gone, and no color is shown. The diagram returns back to normal.

Figs. 12 and 13, for instance, demonstrate the scenario where a software designer moves her cursor over the PaperSession class in the UML diagram in Fig. 11. Two colors are displayed because this class participates in both the Business Delegate pattern and the Session Facade pattern. All the classes participating in the Business Delegate pattern are changed to one color (yellow, shown as light



PatternClass{BusinessService@BusinessDelegate[1]}{SessionFacade@SessionFacade[1]}

Fig. 13. BusinessDelegate class diagram with pattern information shown.

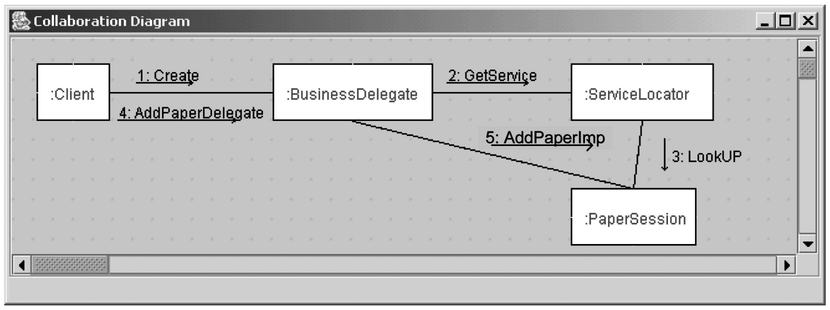


Fig. 14. Collaboration diagram with no pattern-related information.

gray in black-and-white print), whereas the class participating in the Session Facade pattern is changed to another color (green, shown as dark gray in black-and-white print). The color of the overlapping class(es) (PaperSession) is blinking between yellow (for Business Delegate pattern) and green (for Session Facade pattern), as illustrated by two screen shots at two points in time in Figs. 12 and 13, respectively. Since the user's cursor is over the class name compartment of the PaperSession class, VisDP shows a text box containing detailed pattern-related information described in terms of the stereotype and tagged values of this class. Through the stereotype and tagged values, therefore, the designer is able to identify this class participating in both the Business Delegate and the Session Facade patterns. It plays the role of "Business Service" in the Business Delegate pattern and the role of "Session Facade" in the Session Facade pattern. Both patterns in which the BusinessDelegate class participates are the first instances of such patterns. Suppose the designer moves the cursor down to the operation compartment of the same class, the stereotype and tagged values of the operation AddPaperDelegate() is displayed. All colors stay the same. When the user's cursor moves out, all pattern-related information and colors disappear. The diagram returns to normal, as shown in Fig. 11.

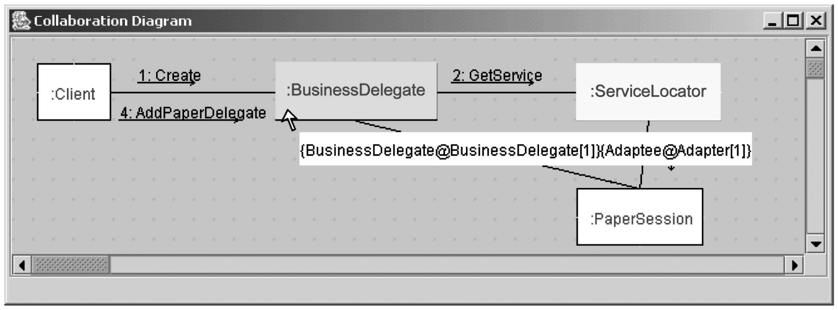
5.2 Behavioral Visualization

A collaboration diagram describes how groups of objects collaborate in some behavior. It usually consists of a set of objects and a set of messages transmitted from one object to another. In collaboration diagrams, an object of a certain class may participate in one or more design patterns. This information is also lost in normal collaboration diagrams. VisDP is able to visualize pattern-related information on demand in UML collaboration diagrams. Similar to class diagrams, the user can move the cursor over any objects in the collaboration diagrams. Different colors are displayed to distinguish different patterns. Detailed pattern-related information is also shown in the text box under the cursor.

As an example, Fig. 14 is the collaboration diagram describing the behavior of the user adding papers through the system discussed in Section 2. When the user adds papers into the system, an object of the BusinessDelegate class is first created. It asks the object of ServiceLocator for the business object (PaperSession in this case) by invoking the GetService method defined in ServiceLocator, which looks up the business object (PaperSession) and returns it to BusinessDelegate. The user can then invoke the AddPaper-Delegate method in BusinessDelegate, which delegates the invocation to AddPaperImp of PaperSession. Fig. 15 shows the scenario where the user's cursor is over the object of the BusinessDelegate class, which participates in both the Business Delegate and Adapter patterns. Similar to the class diagrams, two colors (yellow and green) are displayed and blinking in the overlapping part and the detailed pattern-related information is displayed in a text box in terms of stereotype and tagged values. When the cursor moves out, all colors and pattern-related information disappear. If the users want to know which elements play the other roles of, for example, the BusinessDelegate pattern instance, they can simply move the mouse cursor to other classes/objects that are highlighted with the same color.

5.3 A Web Service

We have implemented the aforementioned techniques as a Web service [7], [55] accessible both from a browser and from a standalone application. The overall system architecture of VisDP includes the following processes: registering the Web service, generating an XML file, invoking the service, and generating new UML diagrams. On the server side, we deploy VisDP as a Web service that is registered in the Web service engine so that it is ready to serve end users.



{BusinessDelegate@BusinessDelegate[1]}{Adaptee@Adapter[1]}

Fig. 15. Collaboration diagram with pattern related information.

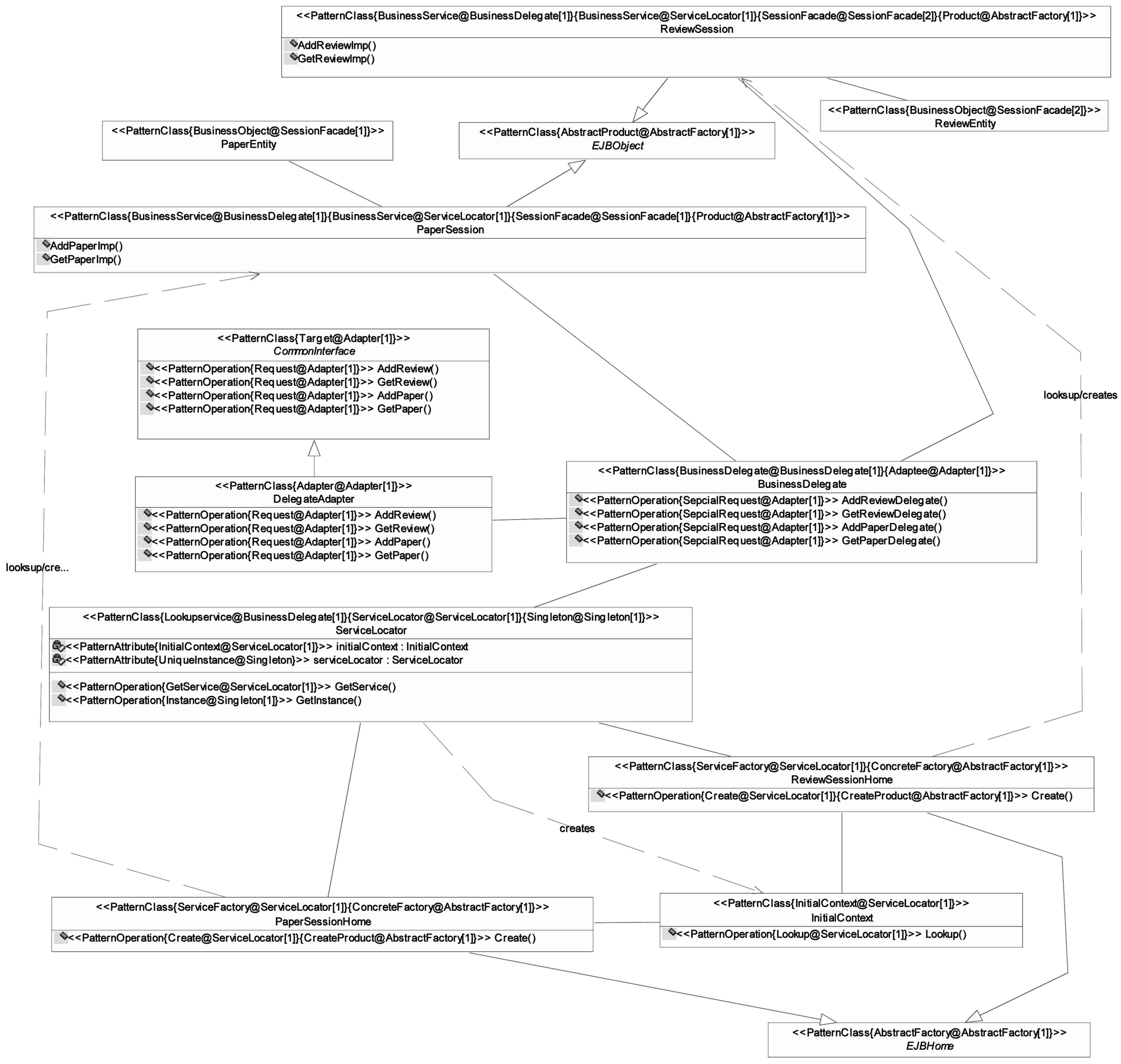

<<PatternClass{BusinessService@BusinessDelegate[1]}{BusinessService@ServiceLocator[1]}{SessionFacade@SessionFacade[2]}{Product@AbstractFactory[1]}>>
ReviewSession
AddReviewImp()
GetReviewImp()
<<PatternClass{BusinessObject@SessionFacade[2]}>>
ReviewEntity
<<PatternClass{BusinessObject@SessionFacade[1]}>>
PaperEntity
<<PatternClass{AbstractProduct@AbstractFactory[1]}>>
EJBObject
<<PatternClass{BusinessService@BusinessDelegate[1]}{BusinessService@ServiceLocator[1]}{SessionFacade@SessionFacade[1]}{Product@AbstractFactory[1]}>>
PaperSession
AddPaperImp()
GetPaperImp()
<<PatternClass{Target@Adapter[1]}>>
CommonInterface
<<PatternOperation{Request@Adapter[1]}>> AddReview()
<<PatternOperation{Request@Adapter[1]}>> GetReview()
<<PatternOperation{Request@Adapter[1]}>> AddPaper()
<<PatternOperation{Request@Adapter[1]}>> GetPaper()
looksup/creates
<<PatternClass{BusinessDelegate@BusinessDelegate[1]}{Adaptee@Adapter[1]}>>
BusinessDelegate
<<PatternOperation{SepcialRequest@Adapter[1]}>> AddReviewDelegate()
<<PatternOperation{SepcialRequest@Adapter[1]}>> GetReviewDelegate()
<<PatternOperation{SepcialRequest@Adapter[1]}>> AddPaperDelegate()
<<PatternOperation{SepcialRequest@Adapter[1]}>> GetPaperDelegate()
<<PatternClass{Adapter@Adapter[1]}>>
DelegateAdapter
<<PatternOperation{Request@Adapter[1]}>> AddReview()
<<PatternOperation{Request@Adapter[1]}>> GetReview()
<<PatternOperation{Request@Adapter[1]}>> AddPaper()
<<PatternOperation{Request@Adapter[1]}>> GetPaper()
looksup/cre...
<<PatternClass{Lookupservice@BusinessDelegate[1]}{ServiceLocator@ServiceLocator[1]}{Singleton@Singleton[1]}>>
ServiceLocator
<<PatternAttribute{InitialContext@ServiceLocator[1]}>> initialContext : InitialContext
<<PatternAttribute{UniqueInstance@Singleton}>> serviceLocator : ServiceLocator
<<PatternOperation{GetService@ServiceLocator[1]}>> GetService()
<<PatternOperation{Instance@Singleton[1]}>> GetInstance()
<<PatternClass{ServiceFactory@ServiceLocator[1]}{ConcreteFactory@AbstractFactory[1]}>>
ReviewSessionHome
<<PatternOperation{Create@ServiceLocator[1]}{CreateProduct@AbstractFactory[1]}>> Create()
creates
<<PatternClass{ServiceFactory@ServiceLocator[1]}{ConcreteFactory@AbstractFactory[1]}>>
PaperSessionHome
<<PatternOperation{Create@ServiceLocator[1]}{CreateProduct@AbstractFactory[1]}>> Create()
<<PatternClass{InitialContext@ServiceLocator[1]}>>
InitialContext
<<PatternOperation{Lookup@ServiceLocator[1]}>> Lookup()
<<PatternClass{AbstractFactory@AbstractFactory[1]}>>
EJBHome

Fig. 16. Conference management system.

On the client side, the user may use any common UML tools, such as Rational Rose, to draw diagrams with stereotypes and tagged values representing pattern-related information as introduced in Section 4. When the user wants to use the VisDP Web service to visualize pattern-related information encapsulated in the diagrams, she may use an appropriate plug-in (for example, UniSys XMI [54] for Rational Rose) to transform a UML diagram into an XML file. The JavaServer Pages (JSP) page [54] of VisDP takes the XML file as an input and returns the user a Java applet with a new UML diagram that displays design patterns on demand as shown previously. There are several advantages of the architecture of VisDP. First, it works with current UML tools, not only Rational Rose, as long as a corresponding plug-in can generate the XML files of the UML diagrams. Second, new services can be provided, for example, for visualizing UML diagrams other than class and collaboration diagrams that have been visualized in VisDP. Third, our service can be used anywhere and anytime [54].

5.4 A Conference Management System

We applied our tool on the partial design of a management system for technical conferences and workshops shown in Fig. 1. Fig. 16 depicts the resulting diagram by statically attaching all pattern-related information in the original UML diagram (Fig. 1). Class BusinessDelegate, which plays the role of BusinessDelegate in the Business Delegate pattern and the role of Adaptee in the Adapter pattern, is used to provide control and protection for the business service, PaperSession and ReviewSession. Class Service-Locator, which plays the role of LookupService in the Business Delegate pattern and the role of ServiceLocator in the Service Locator pattern, provides service lookup and encapsulates the implementation details of such a lookup. To ensure that only one instance of ServiceLocator exists, the Singleton pattern is used. Class InitialContext plays the role of InitialContext in the Service Locator pattern. It provides the InitialContext object from the ServiceLocator class. The PaperSessionHome and ReviewSessionHome classes play the role of ServiceFactory in the Service Locator pattern. They are responsible for the life-cycle management (see the create/lookup dependencies represented by the dashed line in Fig. 1) of the business objects, PaperSession, and ReviewSession, respectively. Classes PaperSession and ReviewSession, which play the role of BusinessService in the Business Delegate pattern, the role of BusinessService in the Service Locator pattern, and the role of SessionFacade in the Session Facade pattern, manage the relationships and interactions among BusinessObject, PaperEntity, and ReviewEntity. Classes PaperEntity and ReviewEntity play the roles of BusinessObject in the Session Facade pattern. The PaperEntity class is used for authors to submit papers and for all users to view the papers that they are entitled to access. The ReviewEntity class allows the reviewers to write the reviews for papers and the authors to check their reviews. The DelegateAdapter class, which plays the role of Adapter in the Adapter pattern, adapts the business functions provided by BusinessDelegate. The CommonInterface class, which plays the role of Target in the Adapter pattern, exposes a common interface to any third party. Classes EJBObject, PaperSession, ReviewSession, PaperSessionHome, ReviewSessionHome, and EJBHome participate in the Abstract Factory pattern. The EJBObject and EJBHome classes play the roles of abstract product and abstract factory, respectively. The PaperSession and ReviewSession classes play the roles of concrete product, whereas the PaperSessionHome and ReviewSessionHome classes play the roles of concrete factory.

Fig. 17 shows some screen shots of the class diagrams generated by VisDP. It shows the structural information of the conference management system. When the user moves the cursor over the ServiceLocator class, the pattern-related information is shown on the class diagram (Fig. 17). The stereotype <<PatternClass{LookupService@BusinessDelegate}{ServiceLocator@ServiceLocator}{Singleton@Singleton}>> is displayed in a text box, indicating that the ServiceLocator class participates in the Business Delegate, Service Locator, and Singleton patterns, and plays the roles of LookupService, ServiceLocator, and Singleton, respectively. The ServiceLocator class is displayed in three alternating colors because it is the overlapping part of three design patterns, Business Delegate, Service Locator, and Singleton. Similarly, the PaperSession and ReviewSession classes are the overlapping part of two design patterns: Business Delegate and Service Locator. They are displayed in two alternative colors. Fig. 17 illustrates three screen shots capturing three moments where the user moves the cursor over the ServiceLocator class. The PaperSession, ReviewSession, BusinessDelegate, and ServiceLocator classes participate in the Business Delegate pattern displayed in yellow (shown as light gray in black-and-white print). The PaperSession, PaperSessionHome, ReviewSession, ReviewSessionHome, ServiceLocator, and InitialContext classes participate in the Service Locator pattern displayed in green (shown as dark gray in black-and-white print). The ServiceLocator class itself participates in the Singleton pattern displayed in brown (shown as gray in black-and-white print).

6 Complexity Studies

As discussed in Section 3, current approaches on visualizing design patterns can be categorized into two kinds, UML-based approaches [20], [31], [48] and non-UML-based approaches [16], [33], [42]. The UML-based approaches can be further divided into single diagram [48] and multidiagram [20], [31]. The multidiagram approaches tend to use different diagrams to represent design patterns at type level and instance level, respectively, whereas the single-diagram approaches represent both the type and instance levels of design patterns in a single diagram. Our approaches can be considered single-diagram approaches.

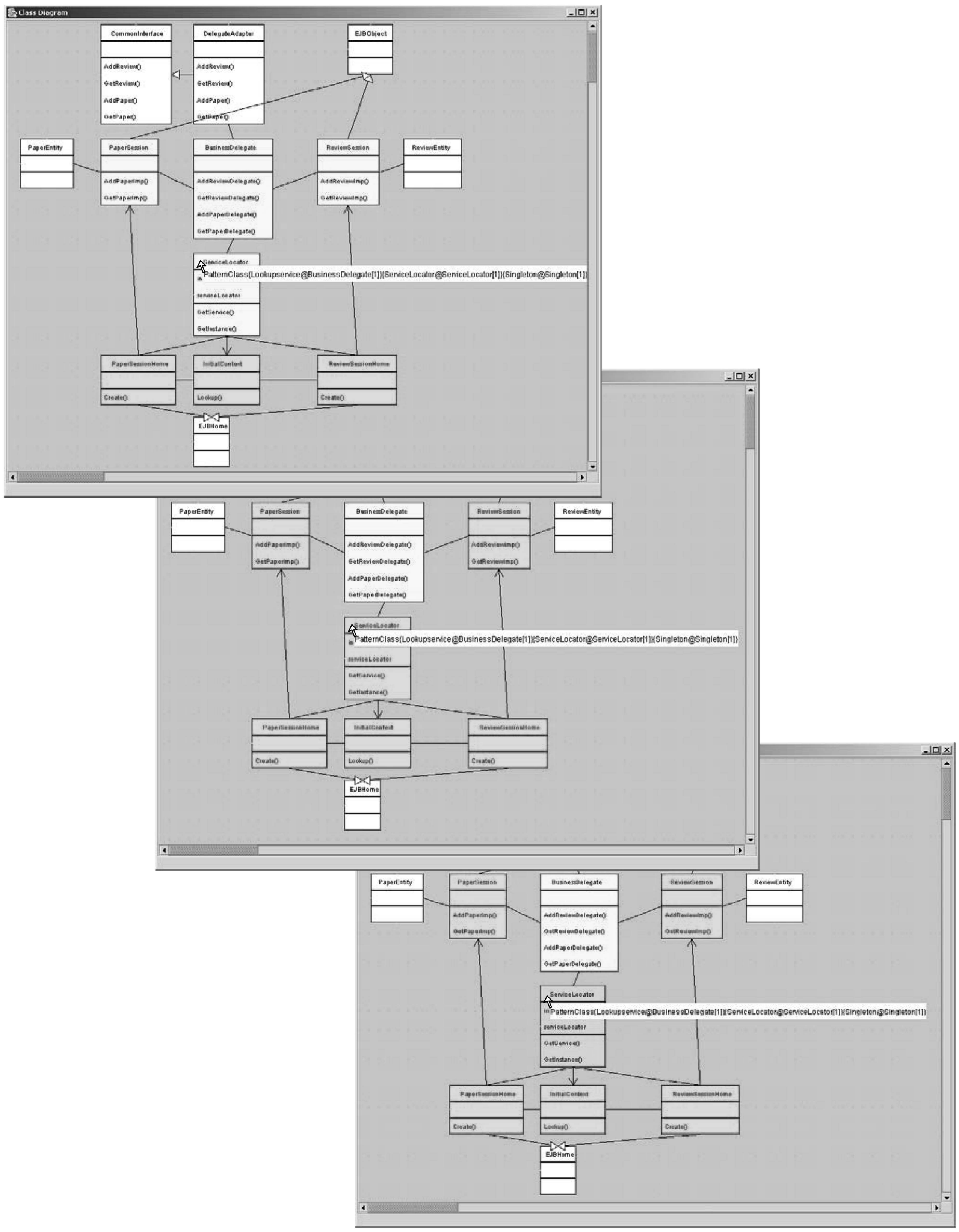
This section reports a comparative study on the graph complexity of different UML-based single-diagram approaches. In this study, we do not include the UML-based multidiagram and non-UML-based approaches for the following reasons: First, there are often repetitive information and interconnections between different diagrams in multidiagram approaches, making them more complex by nature. Second, some of these approaches may have other goals, such as specification, generation, and reconstruction, with additional modeling elements. Third, some of the graph complexity methods that we use are not suitable for analyzing these approaches. We do not include UML behavioral diagrams such as collaboration diagrams in our study because most of other approaches do not deal with the behavioral aspect.

In this complexity study, we use the example in Section 5.4 and draw five diagrams:

1. original UML,
2. UML with stereotype,
3. UML with stereotype on demand,
4. UML collaboration annotation, and
5. pattern:role annotation.

The first diagram is the original UML class diagram without any pattern-related information (see Fig. 1). The second and third diagrams are based on our static (Fig. 16) and dynamic (Fig. 17) approaches, respectively. The fourth diagram uses UML collaboration annotation [5] to represent pattern-related information. The last diagram uses pattern:role annotation [48]. As shown in Fig. 18, we consider several metrics to analyze the graph complexity of these five diagrams: number of edges and nodes, number of characters, number of tokens, McCabe metric [34], diagram class complexity [38], and graphic token count [38].

We first count the number of edges and nodes of all five diagrams, as shown in Fig. 18a. The results show that our approaches (the second and third diagrams) produce similar number of edges and nodes as the original UML class diagram, whereas the fourth and fifth diagrams have significantly more edges and nodes, respectively, than the original UML diagram. This is not surprising because the UML collaboration annotation uses additional edges and nodes to represent pattern-related information. Suppose



PatternClass(Lookupservice@BusinessDelegate[1])(ServiceLocator@ServiceLocator[1])(Singleton@Singleton[1])

Fig. 17. Conference management system with pattern information shown.

there are n patterns and each pattern has m_i roles, $i = 1 \ldots n$. Then, there are n additional nodes and $\sum_{i=1}^{n} m_i$ additional edges in the UML collaboration annotation. Only the number of nodes of the pattern:role annotation increases because it only uses additional nodes to represent pattern-related information. In contrast, our dynamic approach only adds, at most, one node at any point in time, and our static approach does not add edges and nodes in representing

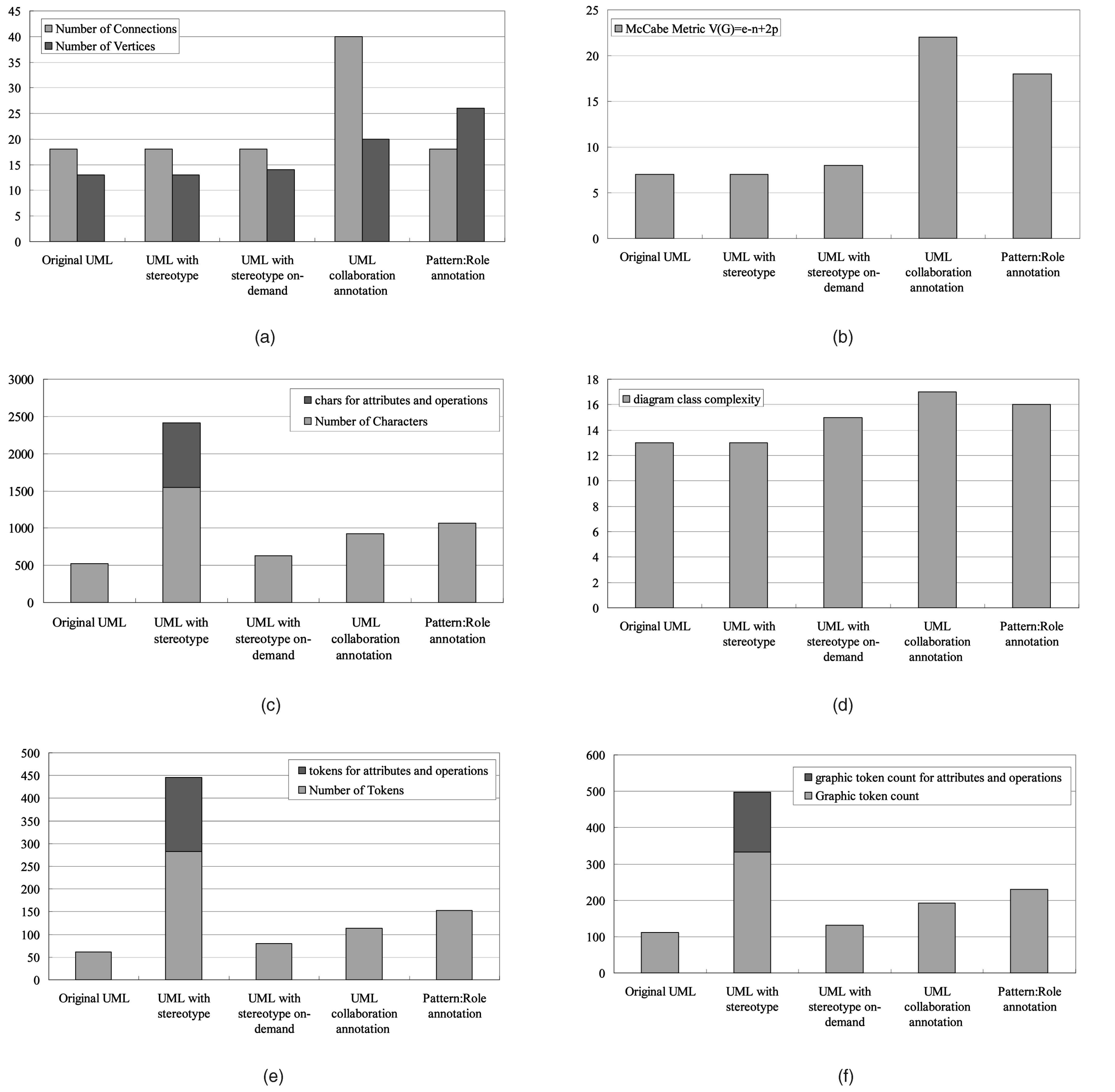


Fig. 18. Graph complexity analysis. (a) Number of edges and nodes. (b) Number of characters. (c) Number of tokens (d) McCabe metric. (e) Diagram class complexity. (f) Graphic token count.

pattern-related information. Instead, we use stereotypes that may result in more textual information such as characters and tokens. This leads us to study the number of characters and tokens in Figs. 18b and 18c, respectively. The tokens here refer to the words separated by spaces or delimiters. A pair of delimiters such as () and {} are counted as one token. For our dynamic diagrams, we count the maximum number of visible tokens/characters at any point in time. The UML with stereotype diagram appears to have significantly more characters and tokens than all other diagrams. The main reason is that we use stereotypes to represent not only the role each class plays, but also the role each operation/attribute plays in a pattern. The bar representing the UML with stereotype approach in Fig. 18b or Fig 18c includes two parts. The upper part displays the number of characters or tokens in the stereotypes representing the role of each operation/attribute, whereas the lower part depicts that of each class. In order to follow UML stereotype syntax, there are some additional characters and tokens such as stereotype name (for example, PatternClass and PatternOperation) and delimiters (for example, {} and []) in the UML with stereotype diagram, which still results in more characters and tokens than other approaches.

McCabe metric [34] is a graphic metric for cyclomatic complexity. It can be defined to compute the complexity of a graph in the following way: For a graph G with n nodes, e edges, and p connected components, $v(G) = e - n + 2p$. McCabe metric tends to compute the graphic complexity based on the numbers of the edges and nodes. We calculate the McCabe metric of all five diagrams. Fig. 18d shows that both the UML collaboration annotation and the pattern:role annotation render the diagrams with significantly higher McCabe complexity. This result converges with the one of the number of edges and nodes in Fig. 18a.

The diagram class metric characterizes the complexity of the class of diagrams being created. In this metric, not only edges and nodes are considered, but also labels are included. In contrast to previous metrics, this one takes into account the types of edges, nodes, and labels, instead of the numbers of them. An edge of different type from other edges uses some visual attribute or appendage such as line weight, fill pattern (dotted versus dashed), arrowhead type, arrow tail type, or color. A node is of different graphic type by means of some visual attribute such as shape, color, fill pattern, or size. The type of a label differs from others through graphic attribute such as font, font weight, size, or capitalization. The diagram class complexity can be computed as

$$\begin{aligned}&\text{Diagram Class Complexity} =\\&\qquad \text{number of node types} + \text{number of edge types}\\&\qquad + \text{number of label types.}\end{aligned}$$

We compute the diagram class complexity of all five diagrams with the result shown in Fig. 18e. Because all the approaches add only a few new types of edges, nodes, and labels, their diagrams are not significantly more complex than the original UML diagram. Only the diagram with UML collaboration annotation is slightly more complex than all other diagrams in terms of diagram class metric.

The graphic token count is a measure of graph complexity by counting the total number of all the nodes, edges, and labels in a diagram. If a node contains another node, there is a containment relationship. Thus, an implicit edge is counted for such containment relationship. If a node is directly adjoined to another node, there is an adjoinment relationship causing an implicit edge to be counted. Labels are text, whose count is done according to the rules of textual token counting. The graphic token count complexity can be computed:

$$\begin{aligned}&\text{Graphic token count} =\\&\text{number of nodes} + \text{number of edges} + \text{textual token count} +\\&\text{number of containment} + \text{number of adjoinments.}\end{aligned}$$

This metric considers labels, as well as nodes and edges, and treats a textual token with the same complexity as a node or edge. Fig. 18f shows the complexity of all five diagrams in terms of this metric. The UML with stereotype appears to be significantly more complex than other approaches. As discussed previously, the main reason is that we use stereotypes to represent operation/attribute roles, as well as class roles in a pattern, which increases the complexity of the diagram. If we remove the stereotypes of operation/attribute roles as shown in the upper part of the bar of UML with stereotype, then the remaining becomes closer to other approaches.

In summary, our UML with stereotype approach is less complex than other approaches in the complexity metrics if not considering the number of characters or tokens. Overall, our dynamic approach (UML with stereotype on demand) is less complex than or equal to other approaches in all graphic complexity metrics discussed previously. Our measurement of the complexity metrics are mostly based on the number of nodes and edges, which belong to the category of geometrical complexity [10]. According to [10], the simpler a diagram geometrically, the easier it can be understood.

7 Case Study

In this section, we study the Java.awt package from JDK version 1.4. It consists of 485 classes stored in 345 files. The total number of classes includes inner and anonymous classes in the package. The total number of lines of the code is 142.8 KLOC. The main goal of this case study is to evaluate the scalability of our approach and tool in the context of a widely used real-world application. To evaluate the scalability, we use existing tools such as Rational Rose to recover the design of the Java.awt package from its source code. A snapshot of the reverse-engineering result is shown in Fig. 19. Due to the large size of the class diagram, only part of the design of Java.awt is shown in the figure. Other parts of the design may be viewed by moving the horizontal and vertical scroll bars. Inner and anonymous classes in the package are not shown.

Based on the recent work on reverse-engineering design patterns from Java.awt, in particular, the result in [39], we manually add the stereotypes of pattern-related information in the class diagram of Java.awt. Niere et al. [39] have studied the Java.awt package and identified four design patterns, one instance of the Strategy pattern, one instance of the Composite pattern, and two instances of the Bridge pattern. In addition to their discovery using their tool, we manually identified some more design patterns in the Java.awt package as shown in Table 3. The first four design pattern instances, Strategy[1], Composite[1], Bridge[1], and Bridge[2], are discovered automatically in [39], whereas the last three pattern instances, Bridge[3], Adapter[1], and Adapter[2], are through our manual discovery. The left column in Table 3 lists the names of the design patterns and their corresponding instance indices in the design. The instance indices are put into the angle brackets following the pattern names. The middle column in Table 3 lists the roles in the corresponding patterns in the left column. The right column lists the Java.awt classes that play the corresponding roles in the middle column of the corresponding patterns in the left column. For instance, the first instance of the Bridge pattern in Table 3 has three roles: Abstraction, Implementor, and ConcreteImplementor. The Container class from Java.awt plays the role of Abstraction. The LayoutManager class plays the role of Implementor, whereas the FlowLayout and GridLayout classes play the role of ComcreteImplementor in the Bridge pattern.

We manually add all the information related to these seven pattern instances in the class diagram of Java.awt based on our UML profile. We then use the UniSys plug-in [54] to generate the XMI file as the input to our VisDP tool. Our tool returns a Java applet, as shown in Fig. 20, which is

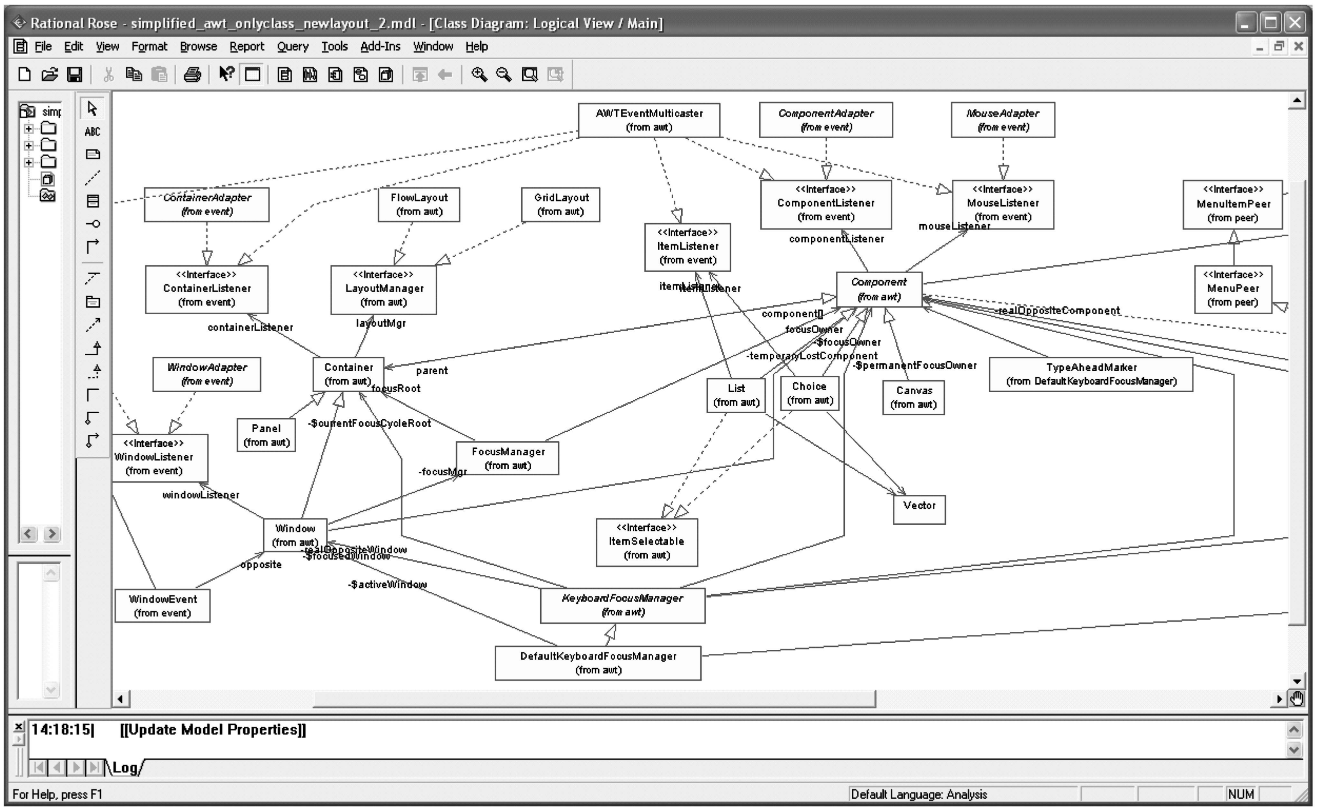

Rational Rose - simplified_awt_onlyclass_newlayout_2.mdl - [Class Diagram: Logical View / Main]
File Edit View Format Browse Report Query Tools Add-Ins Window Help
AWTEventMulticaster (from awt)
ComponentAdapter (from event)
MouseAdapter (from event)
ContainerAdapter (from event)
FlowLayout (from awt)
GridLayout (from awt)
<<Interface>> ItemListener (from event)
<<Interface>> ComponentListener (from event)
<<Interface>> MouseListener (from event)
<<Interface>> MenuItemPeer (from peer)
<<Interface>> ContainerListener (from event)
<<Interface>> LayoutManager (from awt)
Component (from awt)
<<Interface>> MenuPeer (from peer)
WindowAdapter (from event)
Container (from awt)
TypeAheadMarker (from DefaultKeyboardFocusManager)
List (from awt)
Choice (from awt)
Canvas (from awt)
Panel (from awt)
<<Interface>> WindowListener (from event)
FocusManager (from awt)
Vector
Window (from awt)
<<Interface>> ItemSelectable (from awt)
WindowEvent (from event)
KeyboardFocusManager (from awt)
DefaultKeyboardFocusManager (from awt)
14:18:15| [[Update Model Properties]]
Log
For Help, press F1
Default Language: Analysis
NUM

Fig. 19. A snapshot of Java.awt package.

a screen shot of a part of Java.awt class diagram. The user may use the horizontal and vertical scroll bars to view other part of the class diagram.

Fig. 20 shows a snapshot of our tool when the user moves the mouse cursor on top of the Component class from the Java.awt package. The Component class participates in three design patterns, Bridge, Strategy, and Composite. The stereotype <<PatternClass{Abstraction@ Bridge[2]}{Strategy@Strategy[1]}{Component@Composite [1]} shown in the text bar describes that the Component class plays the role of Abstraction in the second instance of the Bridge pattern, the role of Strategy in the first instance of the Strategy pattern, and the role of Component in the first instance of the Composite pattern. The Component class displays three colors in alternation because it is overlapped by the three pattern instances. For brevity, we show only one snapshot, instead of three.

TABLE 3
Design Patterns Applied in Java.awt

Pattern[instance]	Roles in Patterns	Participating classes
Strategy[1]	Strategy	Component
	ConcreteStrategy	Container Canvas Choice List
Composite[1]	Component	Component
	Composite	Container
Bridge[1]	Abstraction	Container
	Implementor	LayoutManager
	ConcreteImplementor	FlowLayout GridLayout
Bridge[2]	Abstraction	Component
	Implementor	ComponentPeer
Bridge[3]	Abstraction	Container
	Implementor	ContainerPeer
Adapter[1]	Target	ComponentListener
	Adapter	ComponentAdapter
Adapter[2]	Target	MouseListener
	Adapter	MouseAdapter

8 Conclusions and Future Work

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 are lost [11], [13], [15], [48]. Without explicitly representing pattern-related information, designers are forced to communicate at the class and object level, instead of the pattern level. The design decisions and trade-offs captured in the pattern are also lost. The designers are not able to trace design patterns into the system design. In this paper, we have introduced a static and a dynamic technique for explicit visualization of design patterns in system designs. We present a UML profile for design patterns to attach pattern-related information through stereotypes, tagged values, and constraints in UML diagrams. Based on this static technique, we have developed a tool for dynamically displaying pattern-related information. Our approach allows the user to identify design patterns by moving the mouse cursor and viewing color changes in UML diagrams. Additional pattern-related information can

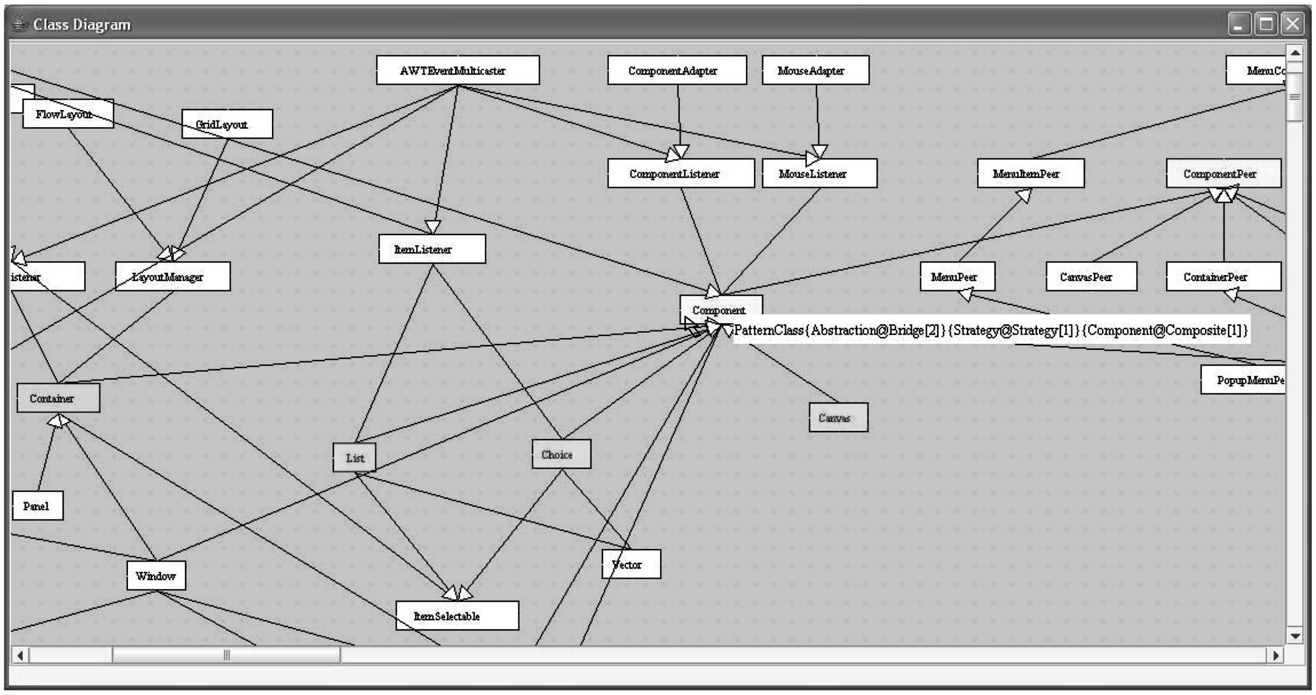


Fig. 20. A screen shot of VisDP.

be dynamically displayed based on the current cursor location. Furthermore, our tool is deployed as a Web service so that anyone can use it online by accessing our tool Web site [54] and providing, as an input, an XMI file generated by the plug-ins of a UML tool such as Rational Rose [53] or ArgoUML [51]. In this way, we present a service-oriented architecture that not only allows our service to work with current UML tools but also allows new services to be developed. We also use a case study to demonstrate and evaluate our approach and perform a comparative study to show the complexity of our approach. In summary, we have achieved our goals presented in Section 2. To the best of our knowledge, we are the first to define a UML profile for visualizing design patterns and the first to provide dynamic techniques for visualizing pattern-related information in software system designs.

We are converting our tool as a plug-in of current UML tools such as Rational Rose. Therefore, the user can not only use the tool online but also download and use it as a plug-in of a UML tool. Most of the current tools for design pattern detections from source code only provide the number of instances of each detected pattern. There is generally no information about where these detected patterns are in the context of the system design. We are working on the techniques to automatically discover pattern instances from source code [12] and add pattern-related information as stereotypes directly into the XMI file. Thus, we can use VisDP to visualize the detected patterns in system designs.

Based on our techniques and tool presented in this paper, we are also working on explicitly tracing and visualizing the evolutions of design patterns. As discussed in [2], many design patterns document some ways for their future evolution and change. Design for change [21] is one of the important goals of design patterns. We plan to extend our techniques and tool to help the designers to manage the changes and evolutions of design patterns.

Using the same color scheme for the class and collaboration diagrams, our approach is limited to consistence checking between the class and collaboration diagrams, as well as automated generation of the class and collaboration diagrams of an application from those of a design pattern. We plan to work on the consistence checking and automated generation of application diagrams from pattern diagrams. VisDP is developed based on the UML and XMI standards. Thus, it can work with other techniques and tools following the same standards. However, it does not work well with software design and design patterns represented by other incompatible techniques and standards. VisDP is constrained by the standards we follow. Nevertheless, the ideas of our approach can be applied to develop other tools to visualize design pattern instances in software designs described by other standards.

We plan to evaluate the usefulness of our techniques and tool to software engineers by conducting controlled experiments. Through these experiments, we can investigate whether our tool provides information relevant and also useful in contents, as well as format to software engineers. This usability study may also be the source of future improvement of our techniques and tool, for example, including the improvement on the layout of class diagrams. The layout of VisDP will be improved in three aspects: First, reflecting the original layout such as that of Rational Rose as much as possible so that the user's mental model is maintained. Second, developing effective orthogonal drawing with bended edges by adapting some existing algorithms [9], [46]. Third, introducing hierarchical and clustered views so that the final layout is scalable and can be easily navigated.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive and insightful comments that have helped them to improve the presentation significantly.

References

[1] P. Alencar, D. Cowan, and C. Lucena, "A Formal Approach to Architectural Design Patterns," *Proc. Third Int'l Symp. Formal Methods Europe,* pp. 576-594, 1996.

[2] P. Alencar, D. Cowan, J. Dong, and C. Lucena, "A Pattern-Based Approach to Structural Design Composition," *Proc. IEEE 23rd Ann. Int'l Computer Software and Applications Conf. (COMPSAC '99),* pp. 160-165, Oct. 1999.

[3] D. Alur, J. Crupi, and D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies.* Sun Microsystems, 2001.

[4] S. Berner, M. Glinz, and S. Joos, "A Classification of Stereotypes for Object-Oriented Modeling Languages," *Proc. Second Int'l Conf. Unified Modeling Language (UML '99),* pp. 249-264, Oct. 1999.

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

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.

[7] E. Cerami, *Web Services Essentials.* O'Reilly and Assoc., 2002.

[8] J.O. Coplien and D.C. Schmidt, *Pattern Languages of Program Design.* Addison-Wesley, 1995.

[9] G. Di Battista, P. Eades, R. Tamassia, and I.G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, 1999.

[10] C. Ding and P. Mateti, "A Framework for the Automated Drawing of Data Structure Diagrams," *IEEE Trans. Software Eng.,* vol. 16, no. 5, May 1990.

[11] J. Dong, "Adding Pattern Related Information in Structural and Behavioral Diagrams," *Information and Software Technology (IST),* vol. 46, no. 5, pp. 293-300, Apr. 2004.

[12] J. Dong, D.S. Lad, and Y. Zhao, "DP-Miner: Design Pattern Discovery Using Matrix," *Proc. 14th Ann. IEEE Int'l Conf. Eng. Computer-Based Systems (ECBS '07),* pp. 371-380, Mar. 2007.

[13] J. Dong and S. Yang, "Visualizing Design Patterns with a UML Profile," *Proc. IEEE Symp. Human Centric Computing Language and Environments,* pp. 123-125, Oct. 2003.

[14] J. Dong, S. Yang, and K. Zhang, "VisDP: A Web Service for Visualizing Design Patterns on Demand," *Proc. IEEE Int'l Conf. Information Technology Coding and Computing (ITCC '05),* pp. 385-391, Apr. 2005.

[15] J. Dong and K. Zhang, *Design Pattern Compositions in UML. Software Visualization—From Theory to Practice.* Kluwer Academic, pp. 287-308, 2003.

[16] A.H. Eden, J. Gil, and A. Yehudai, "Precise Specification and Automatic Application of Design Patterns," *Proc. 12th IEEE Int'l Automated Software Eng. Conf.,* pp. 143-152, Nov. 1997.

[17] G. Florijn, M. Meijers, and P. van Winsen, "Tool Support for Object-Oriented Patterns," *Proc. European Conf. Object-Oriented Programming,* pp. 472-495, 1997.

[18] M. Fontoura, W. Pree, and B. Rumpe, "UML-F: A Modeling Language for Object-Oriented Frameworks," *Proc. European Conf. Object-Oriented Programming (ECOOP '00),* pp. 63-82, July 2000.

[19] M. Fowler, *Analysis Patterns: Reusable Object Models.* Addison-Wesley, 1997.

[20] R.B. France, D. Kim, S. Ghosh, and E. Song, "A UML-Based Pattern Specification Technique," *IEEE Trans. Software Eng.,* vol. 30, no. 3, Mar. 2004.

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

[22] G.C. Gannod and B.H.C. Cheng, "A Framework for Classifying and Comparing Software Reverse Engineering and Design Recovery Techniques," *Proc. Working Conf. Reverse Eng.,* F. Balmas, M. Blaha, and S. Rugaber, eds., pp. 77-88, 1999.

[23] Y.-G. Gueheneuc, K. Mens, and R. Wuyts, "A Comparative Framework for Design Recovery Tools," *Proc. Conf. Software Maintenance and Reeng. (CSMR '06),* 2006.

[24] Y. Gueheneuc, H. Sahraoui, and F. Zaidi, "Fingerprinting Design Patterns," *Proc. 11th Working Conf. Reverse Eng. (WCRE '04),* 2004.

[25] J.H. Hausmann, R. Heckel, and S. Sauer, "Towards Dynamic Meta Modeling of UML Extensions: An Extensible Semantics for UML Sequence Diagrams," *Proc. 2001 IEEE Symp. Human-Centric Computing Languages and Environments,* pp. 80-87, Sept. 2001.

[26] D. Heuzeroth, T. Holl, and W. Löwe, "Combining Static and Dynamic Analyses to Detect Interaction Patterns," *Proc. Sixth Int'l Conf. Integrated Design and Process Technology (IDPT '02),* June 2002.

[27] M.M. Kande and A. Strohmeier, "Towards a UML Profile for Software Architecture Descriptions," *Proc. Unified Modeling Language,* pp. 513-527, 2000.

[28] R. Keller, R. Schauer, S. Robitalille, and P. Page, "Pattern-Based Reverse-Engineering of Design Components," *Proc. 21st Int'l Conf. Software Eng.,* pp. 226-235, May 1999.

[29] C. Kramer and L. Prechelt, "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software," *Proc. Working Conf. Reverse Eng.,* Nov. 1996.

[30] S. Lahtinen and J. Peltonen, "Enhancing Usability of UML CASE-Tools with Speech Recognition," *Proc. 2003 IEEE Symp. Human-Centric Computing Languages and Environments,* pp. 227-235, Oct. 2003.

[31] A. Lauder and S. Kent, "Precise Visual Specification of Design Patterns," *Proc. European Conf. Object-Oriented Programming,* pp. 114-134, 1998.

[32] A. LeGuennec, G. Sunye, and J. Jezequel, "Precise Modeling of Design Patterns," *Proc. Third Int'l Conf. Unified Modeling Language,* pp. 482-496, Oct. 2000.

[33] D. Mapdlsden, J. Hosking, and J. Grundy, "Design Pattern Modelling and Instantiation Using DPML," *Proc. 40th Int'l Conf. Object-Oriented Languages and Systems (TOOLS Pacific '02),* 2002.

[34] T. McCabe, "A Software Complexity Measure," *IEEE Trans. Software Eng.,* vol. 2, no. 6, pp. 308-320, Dec. 1976.

[35] N. Medvidovic, D.S. Rosenblum, D.F. Redmiles, and J.E. Robbins, "Modeling Software Architectures in the Unified Modeling Language," *ACM Trans. Software Eng. and Methodology,* vol. 11, no. 1, pp. 2-57, Jan. 2002.

[36] Y. Medynskiy, N. Ducheneaut, and A. Farahat, "Using Hybrid Networks for the Analysis of Online Software Development Communities," *Proc. ACM Int'l Conf. Human Factors in Computing Systems (CHI '06),* pp. 513-516, Apr. 2006.

[37] T. Mikkonen, "Formalizing Design Pattern," *Proc. 20th Int'l Conf. Software Eng.,* pp. 115-124, 1998.

[38] J.V. Nickerson, "Visual Programming," PhD dissertation, New York Univ., 1994.

[39] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, and J. Welsh, "Towards Pattern-Based Design Recovery," *Proc. 24th Int'l Conf. Software Eng.,* pp. 338-348, 2002.

[40] J. Peltonen and P. Selonen, "Processing UML Models with Visual Scripts," *Proc. 2001 IEEE Symp. Human-Centric Computing Languages and Environments,* pp. 264-271, Sept. 2001.

[41] W. Pree, *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1995.

[42] S.P. Reiss, "Working with Patterns and Codes," *Proc. 33rd Hawaii Int'l Conf. System Sciences,* 2000.

[43] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[44] S. Sauer and G. Engels, "UML-Based Behavior Specification of Interactive Multimedia Applications," *Proc. 2001 IEEE Symp. Human-Centric Computing Languages and Environments,* pp. 248-255, Sept. 2001.

[45] T. Schäfer, M. Eichberg, M. Haupt, and M. Menzini, "The SEXTANT Software Exploration Tool," *IEEE Trans. Software Visualization,* vol. 32, no. 9, pp. 753-768, Sept. 2006.

[46] J.M. Six, "Vistool: A Tool for Visualizing Graphs," PhD dissertation, Univ. of Texas at Dallas, Oct. 2000.

[47] G. Sunye, A. LeGuennec, and J. Jezequel, "Design Patterns Application in UML," *Proc. European Conf. Object-Oriented Programming,* pp. 44-62, 2000.

[48] J. Vlissides, "Notation, Notation, Notation," *C++ Report,* Apr. 1998.

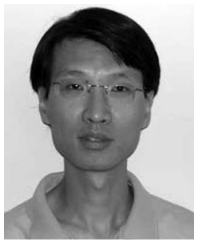
[49] G. Wagner, "A UML Profile for Agent-Oriented Modeling," *Proc. Third Int'l Workshop Agent-Oriented Software Eng.,* July 2002.

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

[51] ArgoUML, http://argouml.tigris.org/, 2006.

[52] Object Management Group, *Unified Modeling Language Specification Version 1.4,* http://www.omg.org, 2001.

[53] IBM Rational Rose, http://www.ibm.com/software/rational/, 2006.
[54] VisDP, http://www.utdallas.edu/~jdong/VisDP, 2007.
[55] "Web Services Architecture Requirements," W3C Working Draft 14, http://www.w3.org, Nov. 2002.



Jing Dong received the BS degree in computer science from Peking University in 1992 and the MMath and PhD degrees in computer science from the University of Waterloo, Canada, in 1997 and 2002, respectively. He is an assistant professor in the Department of Computer Science at the University of Texas at Dallas. His research and teaching interests include formal and automated methods for software engineering, software modeling and design, service-oriented architecture, and visualization. He is a member of the IEEE, IEEE Computer Society, and the ACM.

Sheng Yang received the BE degree from Tsinghua University in 1994 and the MS and PhD degrees in computer science from the University of Texas at Dallas in 2001 and 2006, respectively. His research interests include automated software engineering methods, model-driven architecture, software evolution and analysis, and software engineering tools and environment. He is an architect/lead engineer in a cutting-edge mobile entertainment company in Silicone Valley. His role in the company is to lead the architecture and design of the entire platform. He is also remotely managing a team of software engineers in China. He is a member of the IEEE.

Kang Zhang received the BEng degree in computer engineering from the University of Electronic Science and Technology, China, in 1982 and the PhD degree from the University of Brighton, UK, in 1990. He is a professor of computer science and the director of the Visual Computing Lab at the University of Texas at Dallas. He previously held academic positions in China, the UK, and Australia. His research interests include visual languages, information visualization, and their applications in software engineering and Web engineering. He has had more than 150 publications in these areas (see http://www.utdallas.edu/~kzhang). He is a senior member of the IEEE.