

Representing the Applications and Compositions of Design Patterns in UML

Jing Dong
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083
jdong@utdallas.edu

ABSTRACT

Design patterns capture the distilled experience of expert designers. The compositions of design patterns may reuse design experience and solve a set of problems. Design patterns and their compositions are usually modeled using UML. When a design pattern is applied or composed with other patterns, the pattern-related information may be lost because UML does not track this information. Thus, it is hard for a designer to identify a design pattern when it is applied or composed. The benefits of design patterns are compromised because the designers cannot communicate with each other in terms of the design patterns they use when the design patterns are applied or composed. In this paper, we present notations to explicitly represent each pattern in the applications and compositions of design patterns. The notations allow us to maintain pattern-related information. Thus, a design pattern is identifiable and traceable from its application and composition with others.

Keywords

Design patterns, UML, Notations, Extensions

1. INTRODUCTION

Design patterns [4] document good solutions to recurring problems in a particular context. A design pattern systematically names, explains, and evaluates important and recurring design. The composition of design patterns [8, 1] enable a higher level of reuse than individual design patterns and objects. The modeling and representation of design patterns and their compositions are usually based on object-oriented modeling techniques that use graphical notations such as the Unified Modeling Language (UML) [5].

Notations are important for conveying concepts and information. Most notations can make complex concepts easy to understand and grasp. Good notations and correct uses of them can result in significant gains in terms of precision, expressiveness, unambiguity, succinctness, simplicity, clarity and compactness [9]. There are several different kinds of notations for modeling object-oriented design, such as diagrammatic notations and textual notations. Textual

notations can be further divided into formal text, such as logic-based, algebra-based, process-based notations, and informal text (natural languages). A diagrammatic notation can be beneficial in many ways. First, it can convey complex concepts and models, such as object-oriented design. Second, it can help people grasp large amount of information more quickly than straight text. Third, as well as being easy to understand, it is normally easier to learn drawing diagrams than writing text because diagrams are more concrete and intuitive than text written in formal or informal languages.

However, there are yet shortcomings of graphical notations. The flip side is that graphical notations are sometimes imprecise, ambiguous, unclear and not expressive. For example, the UML notations lack the express power for the intuition and essence of design patterns and the hot-spot of frameworks. Although improvements [6] and extensions [3] to UML provide solutions to some particular problems related to the difficulties of modeling non-determinism in UML, losing pattern-related information after the applications and compositions of design patterns remains a problem of UML. The modeling elements, such as classes, operations, and attributes, in each design pattern usually play some roles that are manifested by their names. The application of a design pattern may change the names of its classes, operations, and attributes to the terms in the application domain. Thus, the role information of the pattern is lost. It is not obvious which modeling elements participate in this pattern. As a result, the designer cannot communicate with others about a system design in terms of design patterns used. The benefits of using design patterns are compromised. To retain the pattern-related information even after the pattern is applied or composed with other patterns, we propose some new notations that extend UML. In our notations, pattern-related information is explicit so that a design pattern can be easily identified when it is applied and composed. The notations are also scalable to large designs containing a number of design patterns as shown in Section 5.

In the next section, we illustrate the problem of inexplicit pattern representations. In Section 3, we present the current solutions to this problem and discuss the shortcomings of these solutions. In Section 4, we introduce several extensions to UML to solve this problem. In Section 5, we describe a case study to show that these extensions overcome the shortcomings of the previous solutions.

2. ILLUSTRATION OF THE PROBLEM

Consider the application of the Composite pattern [4] to design a hierarchical Web application containing a number of Web pages. Using hyperlinks, the user can group Web pages to form larger pages, which in turn can be grouped to form still larger pages. Each page may contain text, pictures, hyperlinks, etc. Figure 1.a depicts the class diagram of the Composite pattern. Figure 1.b shows an

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC 2003 Melbourne, Florida USA

Copyright 2003 ACM 1-58113-624-2/03/03 ...\$5.00.

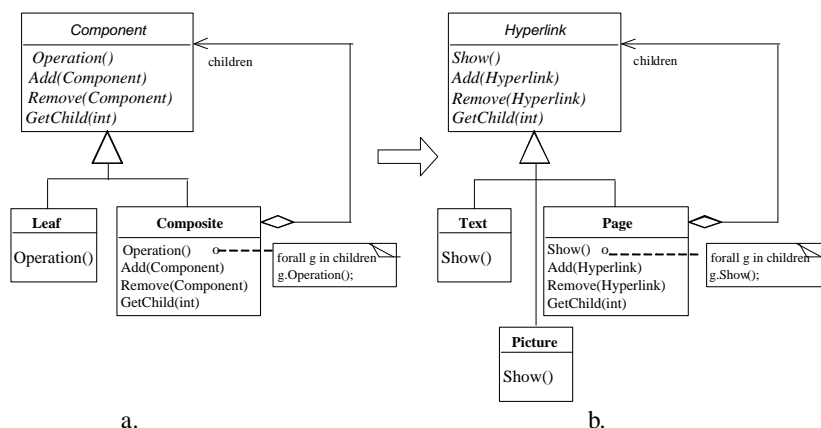


Figure 1: The Composite Pattern and Its Application

application of the Composite pattern in this Web application. Although it is unusual to present a pattern with its application side by side, we chose to show them together because it is easier to identify the participants of the Composite pattern by the roles they play with Figure 1.a. For example, it is obvious to identify which class plays the role of *Component*, *Composite* or *Leaf* in Figure 1.b if the two class diagrams are put in adjacent as shown in Figure 1. In normal applications, a design pattern can have many isomorphism instances not accompanied by the original pattern. Therefore, design patterns are buried under the complex design and require sophisticated reverse-engineering methods and tools to discover them. Without pattern-related information, it is hard for the designer to communicate the decisions and tradeoffs behind a design. Thus, the benefits of using design patterns are compromised because the design is not traceable to the original design pattern. It becomes even worse when the application of a design pattern is composed with those of other patterns. Figure 2, for instance, presents a design of system sort (see Section 5 for details) that contains the applications of five design patterns: Strategy, Bridge, Adapter, Iterator, and Factory Method [4]. It is not obvious which patterns each class participates in Figure 2.

Since the solutions to the problem of the compositions of design patterns also solve the problem of the applications of them, we will concentrate on the solutions to the composition of design patterns, using an example of composing the applications of the Composite pattern and the Decorator pattern [4]. In this work, we do not make the distinction between a composite design pattern¹ and an arbitrary pattern composition. In the following discussions, we use a composition² of the Composite pattern [4] and the Decorator pattern [4] to illustrate our notations for representing pattern compositions.

3. RELATED WORK

In this section, we discuss some previous methods for explicitly representing individual design pattern in a composition of patterns. We show the pros and cons of these methods and argue that they do not satisfy our expectation.

Venn Diagram-Style Pattern Annotation. The first notation

¹A composite design pattern is defined as a composition of design patterns in which the resulting composition is also considered to be a design pattern [8]. A composite design pattern can be seen as a special kind of design pattern compositions.

²This composition is actually a composite design pattern, called the Navigational Contexts pattern in [7].

for identifying patterns in a design diagram is based on Venn diagrams [9]. Figure 3 shows the Composite pattern and the Decorator pattern manifested themselves in their composition. It shows that the Component, Composite and Content classes participate in the Composite pattern, while Component, Content, Context, ConcreteContextA and ConcreteContextB are participants in the Decorator pattern. This notation works fine with a small number of patterns per class. When a class participates in more and more patterns, the overlapping regions, where the class resides, may become hard to distinguish, especially when different gray levels need to be selected to represent different patterns.

Besides the scalability problem, shading in a diagram has the problem that it may not print well on paper by different printers, nor does it reproduce well for faxes and scans. Printers with different quality may render distinct results when printing dissimilar gray levels. Another shortcoming of this notation is that it is not explicit what participant roles a modeling element, such as a class, plays. We not only need to identify each pattern in a design diagram, but also want to show the particular roles each modeling element plays.

Dotted Bounding Pattern Annotation. To prevent the shading problem, we propose a variation of the previous notation that replaces shadings simply by dashed lines. This change solves the problem caused by shading. It, yet, remains hard to identify precisely the roles a modeling element, e.g. a class, a method or an attribute plays. The scalability problem also remains since there can be many dashed lines clashing in the overlapping regions.

UML Collaboration Notation. To address the difficulty of explicit identification of the participant roles a class plays, an alternative notation is provided in UML, called the parameterized collaboration diagrams [5]. This notation can depict design pattern structure by representing patterns and their participants in a class diagram as shown in Figure 4. Dashed ellipses with pattern names inside are used to represent patterns. Dashed lines labeled with participant names are used to associate the patterns with their participating classes. While this notation improves over the previous two notations with the explicit representations of pattern participants, it raises other problems. The dashed lines appear cluttering the presentation. The pattern information is mixed with the class structure, making both hard to distinguish. Moreover, not only a class may play some roles in a design pattern, but also an operation (or attribute) may play some roles. This notation fails to represent the roles an operation (attribute) plays in a design pattern.

Pattern:Role Annotations. To improve the diagrammatic presentation by removing the cluttering dashed lines, Gamma has de-

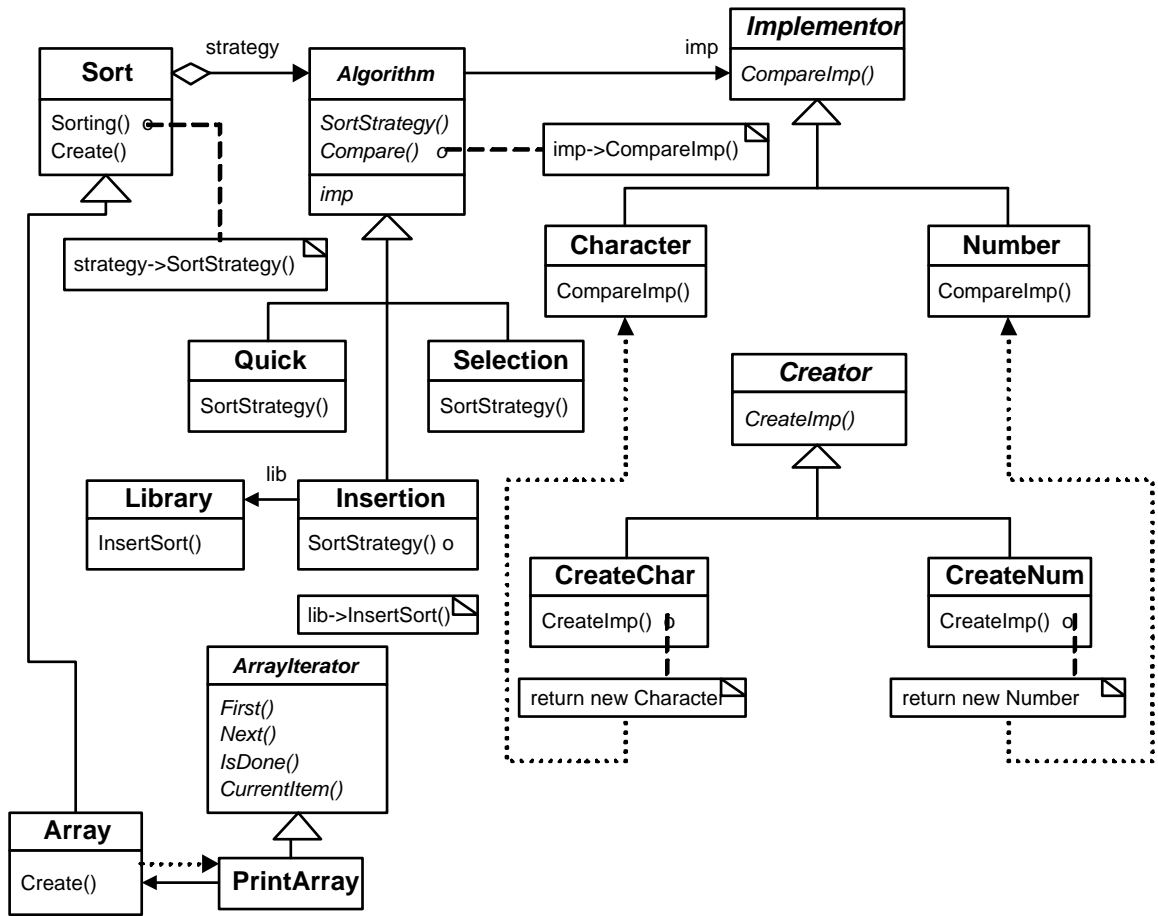


Figure 2: A Design of System Sort

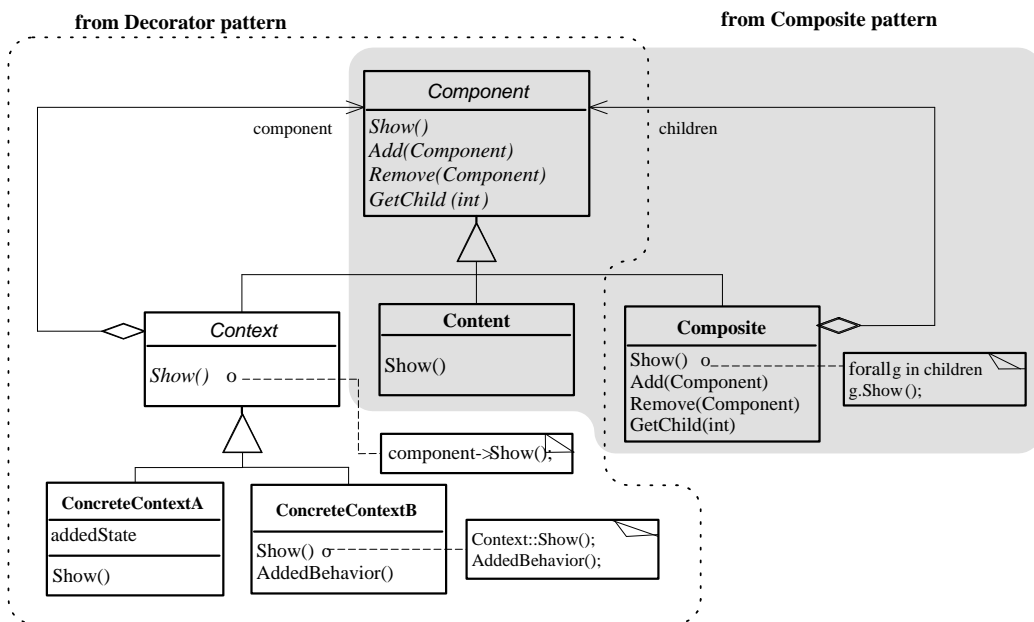


Figure 3: Venn Diagram-Style Pattern Annotation

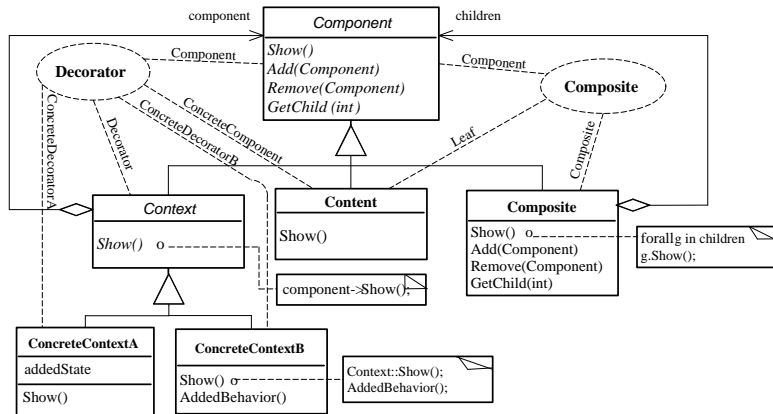


Figure 4: UML Collaboration Notation

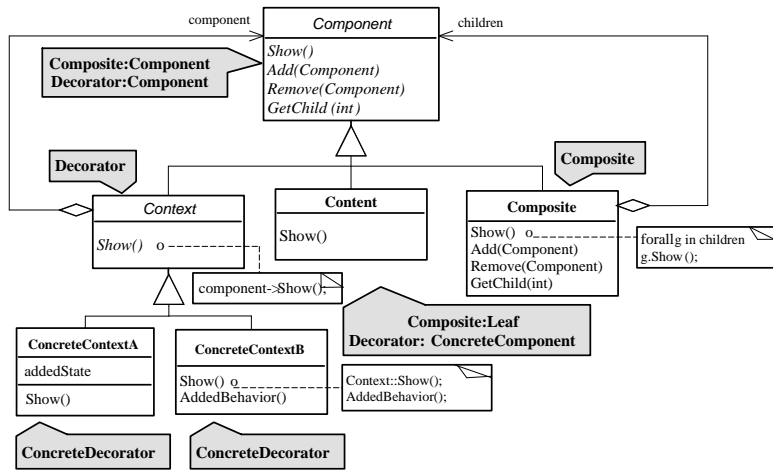


Figure 5: Pattern:Role Annotations

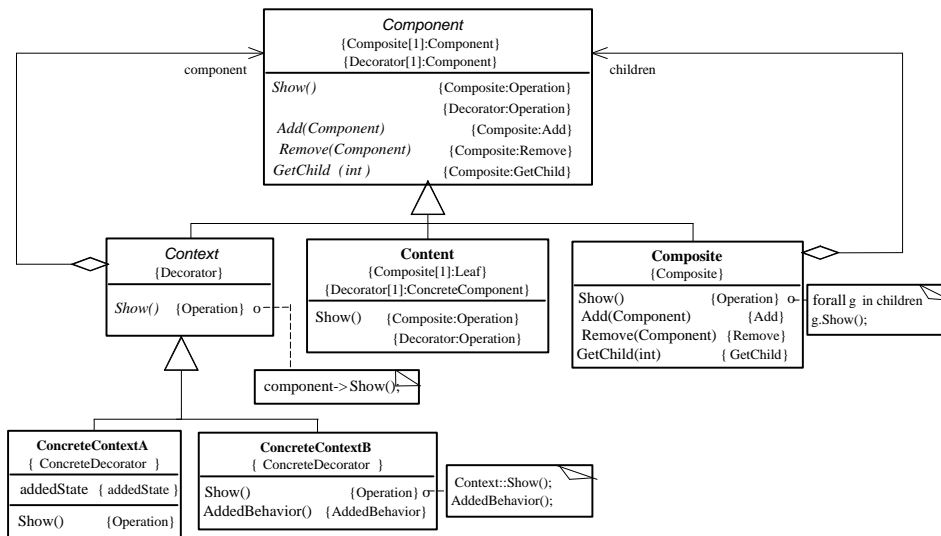


Figure 6: Tagged Pattern Annotation

fined a graphical notation, called “pattern:role” annotations documented in [9]. 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 simplification. Figure 5 depicts 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 according to [9]. Unfortunately, the problems related shading arise again as the first notation. The gray backgrounds do not fax and scan well. 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. Similarly to the previous notation, this notation fails to represent the roles an operation (attribute) plays in a design pattern. If there are different instances of a pattern, furthermore, this notation cannot distinguish in which instance of the pattern a modeling element participates.

4. UML EXTENSIONS

Although the Venn diagram-style notation and the dotted bounding notation can show which classes participate a pattern in a design diagram, these notations cannot explicitly represent the role that each class plays in the pattern. The UML collaboration notation and the “pattern:role” notation improve the expressive power by explicitly representing the role that each class plays in the pattern. However, not only a class may play certain role in a pattern, but also an operation (or an attribute) may play certain role in the pattern. None of these notations can represent the information that an operation or an attribute participates and the roles it plays in a pattern. Further, they cannot distinguish different instances of the same pattern. Explicitly representing operation and attribute roles in a pattern is important because many patterns are based on polymorphism, delegation and aggregation, which are often presented based on the relationships among operations and attributes. Explicit representation of the key operations and attributes can not only help on the application (instantiation) of a pattern because the pattern impose some restrictions through the relationships among operations and attributes, but also assist on the traceability of a pattern since it allows us to trace back to the design pattern from a complex design diagram. To explicitly represent a pattern in the composition of patterns, we present new graphic notations (extensions to UML) to visually represent each individual pattern within an aggregate of patterns. In this way, each individual pattern is explicit in design documentation so that it can be identified easily. These extensions overcome the shortcomings of previous notations.

In order to represent explicitly the roles of each class, operation, and attribute in a pattern, we propose a new notation that is an extension to UML. The extension is defined mainly by applying the UML built-in extensibility mechanisms. This extension forms a basis for a new UML profile [2], especially useful for representing patterns and their instances and participants. UML provides three language extension mechanisms: stereotypes, tagged values, and constraints. Tagged values are used to extend the properties of a modeling element with a certain kind of information. A tagged value is basically a pair consisting of a name (the tag) and the associated value, written as “{tag=value}”. Both tag and value are usually strings only, although the value may have a special interpretation, such as numbers or the Boolean values. In the case of tags with Boolean values, UML allows us to write “{tag}” as a shortcut for “{tag=TRUE}”.

Our new notation is called “tagged pattern annotation”. The idea is that, for each class, we create new tagged values that are used to hold pattern and/or instance(s) and/or participant name(s) asso-

ciated with this given class and its operations and attributes. A tag has the format of “pattern[instance]:role”. If a class is tagged with `Composite[1]:Component`, for example, the class plays a role of `Component` in the first instance of the `Composite` pattern. If it will not cause any ambiguity, only the participant name is shown for simplification. Figure 6 displays the diagram based on our notation, where the `Component` and the `Content` classes are the overlapping part of the composition of the `Decorator` pattern and the `Composite` pattern. With tagged values, the roles that these two classes play in each pattern are shown. In addition, the operations and attributes are attached with tagged values showing the roles they play in each pattern. The limitation of our notation is that the pattern-related information is not as noticeable as the “pattern:role” notation with shading, which is a trade-off³. For a small number of patterns, this new notation can combine with the dotted bounding notation (see Section 3) by bounding each pattern with dashed circles so that the pattern boundaries are explicitly depicted as shown in Figure 8.

Besides the two improvements to make pattern-related information explicit (see Figure 7) and pattern boundaries explicit (see Figure 8), we propose another improvement to extend UML by adding a new compartment in each class in the class diagram. This new compartment of each class is used to hold pattern-related information. Consequently, the pattern and/or instance(s) and/or participant name(s) associated with a class are put into the new compartment of this given class. In this way, pattern-related information is treated as first-class members in the same way as attributes and operations of a class, as shown in Figure 9, where the roles each class plays are displayed in a separate compartment. Note that this notation is asymmetry between classes and attributes/operations, since pattern-related information of attributes/operations must remain in the already existing compartments.

5. CASE STUDY

As an illustration of our extensions to UML, we briefly describe a case study. The case study deals with a general-purpose system sort, which sorts lines of text from standard input and writes the results to standard output. A line is a sequence of characters terminated by a newline. The size of sort files is limited within the main memory. Different sort algorithms, e.g. quick sort and insertion sort, can be chosen at run-time or configured before the system is running. The result will be printed in the order specified by the user. The design of this application (see Figure 2) contains five design patterns: Adapter, Bridge, Factory Method, Iterator, and Strategy.

To address the requirement allowing an interchangeable sorting algorithm, the Strategy pattern was used to encapsulate the different sorting algorithms, e.g., quick sort, insert sort, selection sort, and etc. In this case, we only deal with comparison-based algorithms. Therefore, all algorithms need a function to compare pairs of elements which can be characters, numbers, file folders, etc. The Bridge pattern captures this abstraction since it decouples the abstraction (comparison) from its implementation (character comparison and number comparison) so that they can vary independently. The Factory Method pattern defines an interface for creating objects, but lets subclasses (*CreateChar* and *CreateNum*) decide which class to instantiate. The Iterator pattern is used to print all sorted elements without exposing its underlying representation. Suppose we have a library containing functions as, for example, the

³If we do not need to worry about the shading problem because, for example, everyone has good quality fax machines, scanners, and printers, we can still shade the pattern-related stereotypes so that the pattern-related information appears to occupy a different plane as shown in Figure 7.

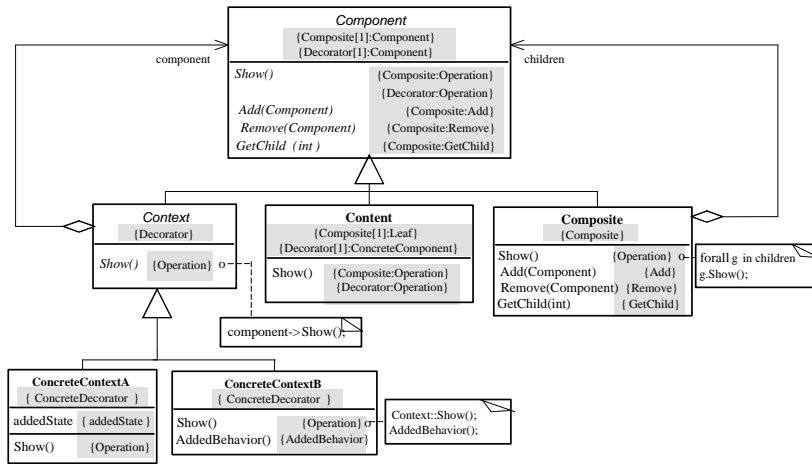


Figure 7: Tagged Pattern Annotation with Shading

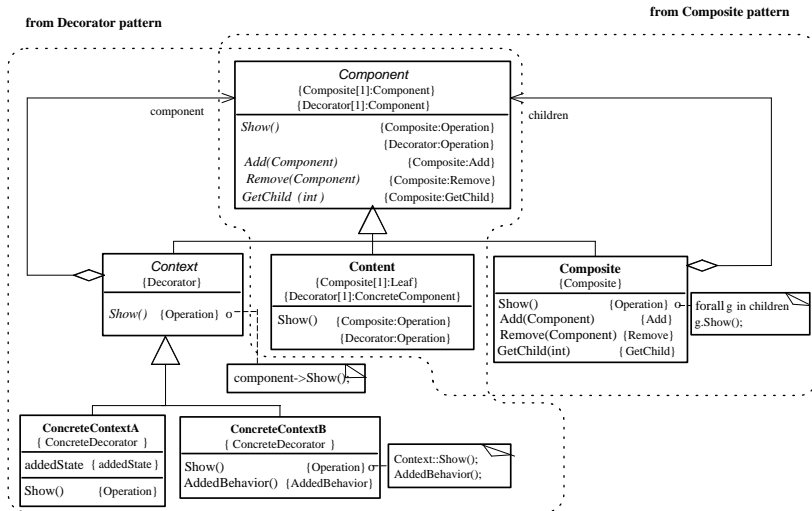


Figure 8: Tagged Pattern Annotation with Bounding

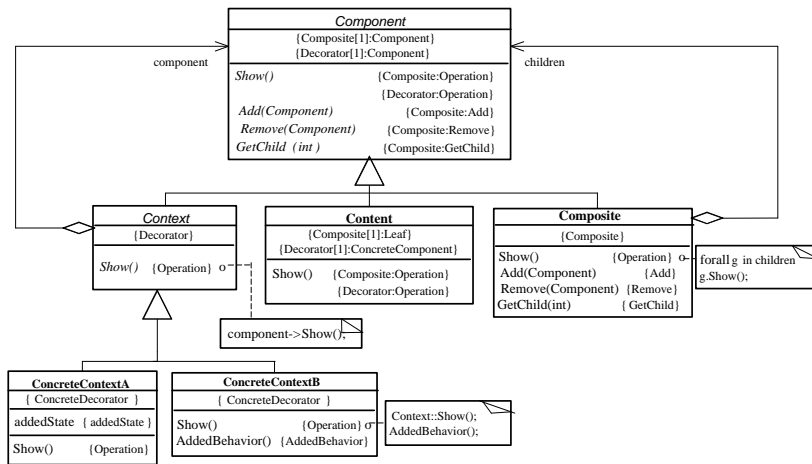


Figure 9: Tagged Pattern Annotation with New Compartments

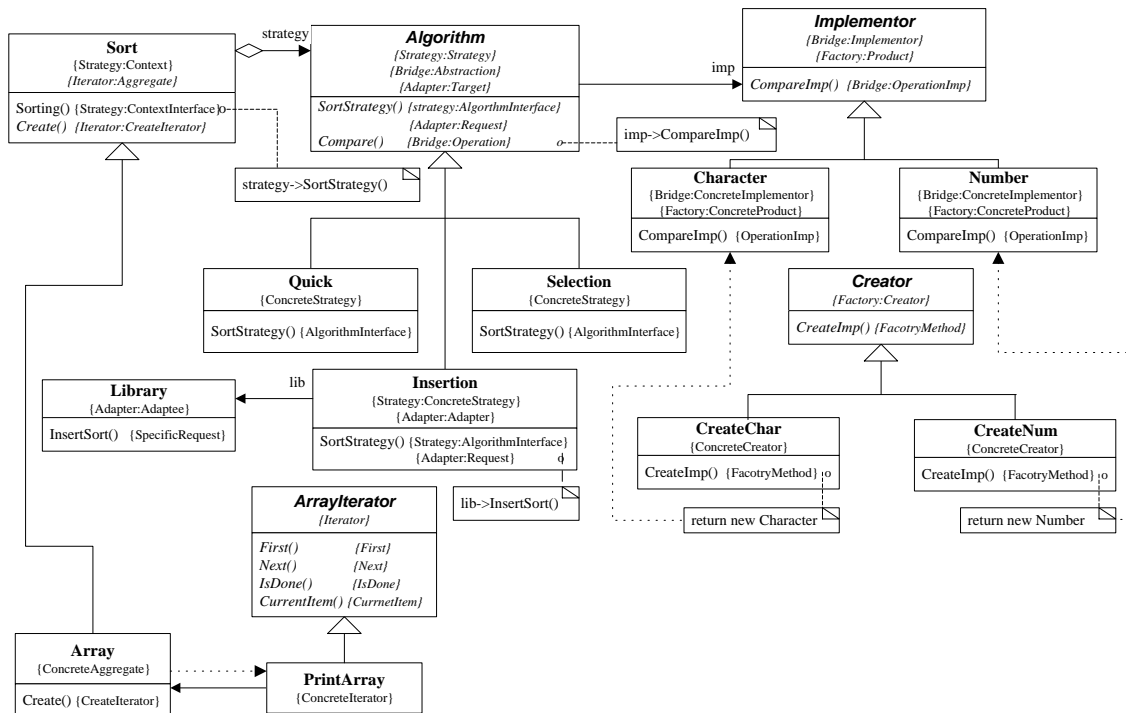


Figure 10: A Design of System Sort in Tagged Pattern Notation

insertion sort, we can reuse some functions required in this design. Since the interface of the insertion sort method may not be compatible with the interface of *SortStrategy* method in the *Algorithm* class, we can use the Adapter pattern to adapt the interface.

Figure 10 presents the design of system sort in tagged pattern notation⁴. With the additional information presented in the figure, we can identify different roles each class (method or attribute) plays in the design. Thus, the design is traceable and the pattern information is not lost as in Figure 2. For example, the *Algorithm* class plays three roles in three patterns: as a *Strategy* in the Strategy pattern, as an *Abstraction* in the Bridge pattern, as a *Target* in the Adapter pattern. The *SortStrategy* method plays two roles in two patterns: as an *AlgorithmInterface* in the Strategy pattern and as a *Request* in the Adapter pattern.

6. CONCLUSIONS

In this paper, we introduced some new notations that extend UML to explicitly visualize design patterns. It is important for designers to describe explicitly patterns in a design diagram because the goals of design patterns are to reuse design experience, to improve communication within and across software development teams, to capture explicitly the design decisions made by designers, and to record design tradeoffs and design alternatives in different applications. The application of a design pattern may change the names of classes, operations, and attributes participating in this pattern to the terms of the application domain. Thus, the roles that the classes, operations, and attributes play in this pattern have lost. This pattern-related information is important to accomplish the goals of design pattern. Without explicitly representing this information, the designers are force to communicate at the class and

⁴Since only one instance of each pattern is applied, the instance part is omitted. The representations with shading, bounding and new compartment can be shown similarly, which are also omitted.

object level, instead of the pattern level. The design decisions and tradeoffs captured in the pattern are lost too. Therefore, the notations provided in this paper help on the explicit representation of design patterns and accomplishing the goals of design patterns.

7. REFERENCES

- [1] J. Dong, P. Alencar, and D. Cowan. Ensuring Structure and Behavior Correctness in Design Composition. *Proceedings of the 7th Annual IEEE International Conference on Engineering of Computer Based Systems, UK*, pages 279–287, April 2000.
- [2] D. D’Souza, A. Sane, and A. Birchenough. First Class Extensibility for UML – Packaging of Profiles, Stereotypes, Patterns. *Proceedings of the 2nd International Conference on UML, Springer-Verlag*, pages 265–277, October 1999.
- [3] M. Fontoura, W. Pree, and B. Rumpe. UML-F: A Modeling Language for Object-Oriented Frameworks. *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*, pages 63–82, July 2000.
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.
- [5] Object Management Group. *Unified Modeling Language Specification, Version 1.4*. <http://www.omg.org>, 2001.
- [6] A. Lauder and S. Kent. Precise Visual Specification of Design Patterns. *Proceedings of European Conference on Object-Oriented Programming*, pages 114–134, July 1998.
- [7] G. Rossi, D. Schwabe, and A. Garrido. Design Reuse in Hypermedia Applications Development. *Proceedings of ACM International Conference on Hypertext*, pages 57–66, 1997.
- [8] J. Vlissides. Composite Design Pattern (They Aren’t What You Think). *C++ Report*, June 1998.
- [9] J. Vlissides. Notation, Notation, Notation. *C++ Report*, April 1998.