



Adding pattern related information in structural and behavioral diagrams

Jing Dong

Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083 USA

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 unified modeling language (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 the structural and behavioral aspects of each pattern in the applications and compositions of design patterns. The notations allow us to maintain pattern-related information in class and collaboration diagrams. Thus, a design pattern is identifiable and traceable from its application and composition with others in these diagrams. A case study is used to illustrate our approach.

© 2003 Published by Elsevier B.V.

Keywords: Design patterns; Unified modeling language; Notations; Extensions

1. Introduction

Design patterns [5] 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 [2,8] 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) [1]. UML is a general-purpose language for specifying, constructing, visualizing, and documenting artifacts of software-intensive systems. It provides a collection of notations to capture different aspects of the system under development. Although UML has become the de facto standard for modeling software systems, it is sometimes still not expressive enough. 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 [4] 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. In normal applications, a design pattern can have many isomorphism instances not accompanied by the original pattern. 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. 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

E-mail address: jdong@utdallas.edu (J. Dong).

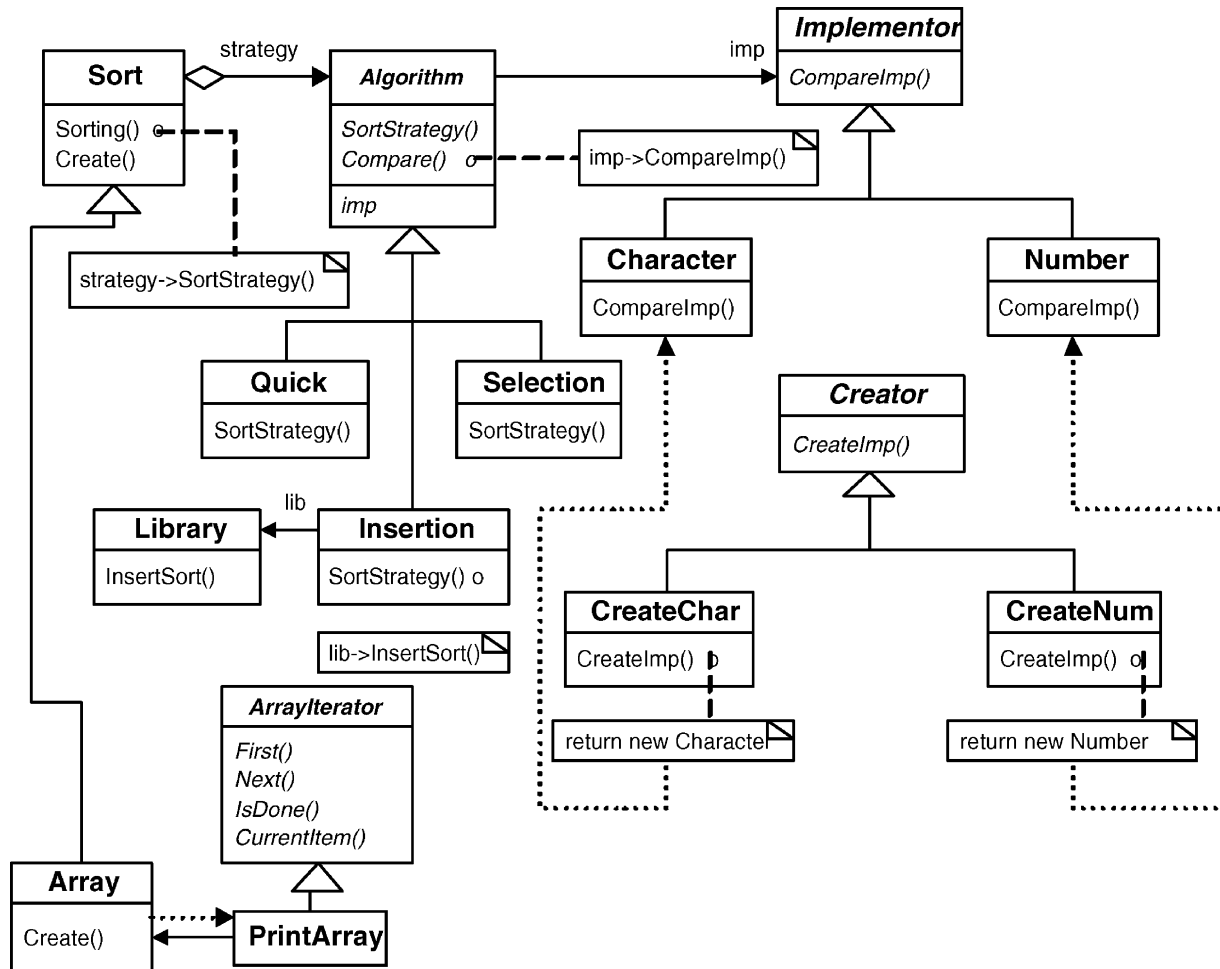


Fig. 1. A design of system sort.

traceable to the original design pattern. It becomes even worse when the application of a design pattern is composed with those of other patterns. For instance, Fig. 1 presents a design of system sort (see Section 4 for details) that contains the applications of five design patterns: Strategy, Bridge, Adapter, Iterator, and Abstract Factory [5]. It is not obvious which patterns each class participates in Fig. 1.

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 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 4. 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 [5]. In this paper, we do not make the distinction between a composite design pattern¹ and an arbitrary pattern composition. We use a composition² of the Composite pattern [5] and the Decorator pattern [5] to illustrate our notations for representing pattern compositions.

In Section 2, we present the current solutions to this problem and discuss the shortcomings of these solutions. In Section 3, we introduce several notations to solve this problem. In Section 4, we describe a case study to show that these extensions overcome the shortcomings of the previous solutions.

¹ A 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.

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

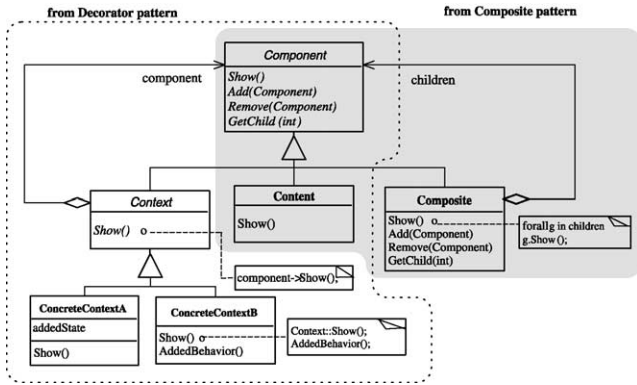


Fig. 2. Venn diagram-style pattern annotation.

2. 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 for identifying patterns in a class diagram is based on Venn diagrams [9]. Fig. 2 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 [1]. This notation can

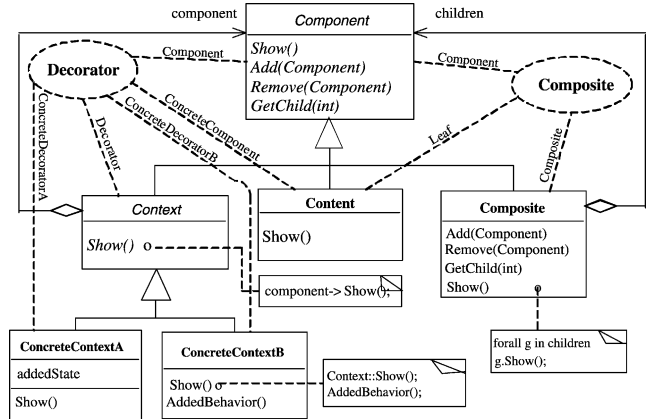


Fig. 3. UML collaboration notation.

depict design pattern structure by representing patterns and their participants in a class diagram as shown in Fig. 3. 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 defined 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. Fig. 4 depicts that the pattern-related

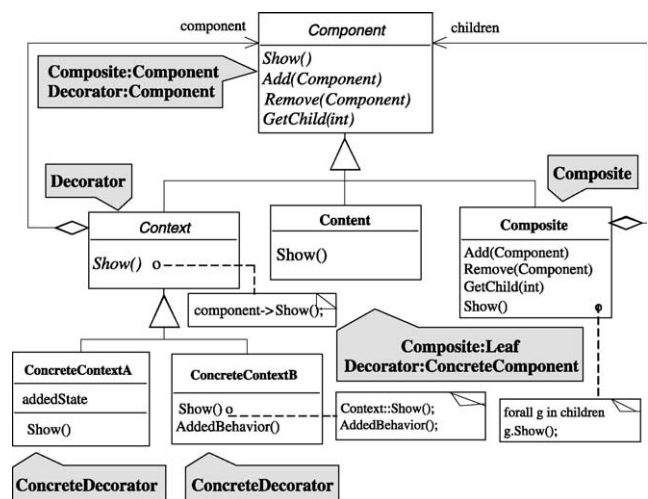


Fig. 4. Pattern: role annotations.

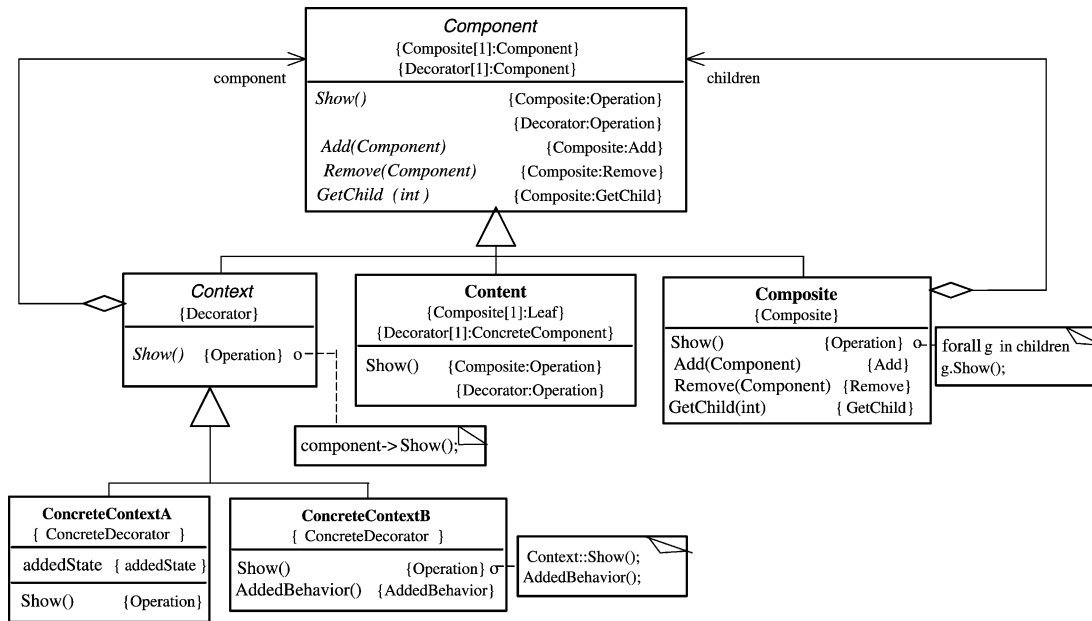


Fig. 5. Tagged pattern annotation (class diagram).

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. Moreover, it is, sometimes, difficult to squeeze in a shared box to a proper position in a diagram, especially when the shared box is used to represent the roles of a class in the middle of the diagram.

3. Tagged pattern notation

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. It is also unclear whether the

notations are applicable for visualizing the behavioral aspect of design pattern in addition to the structural one. 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

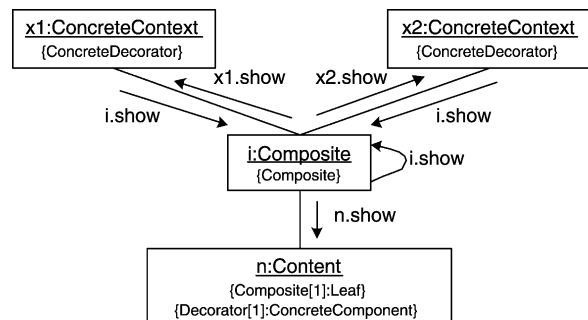


Fig. 6. Tagged pattern annotation (collaboration diagram).

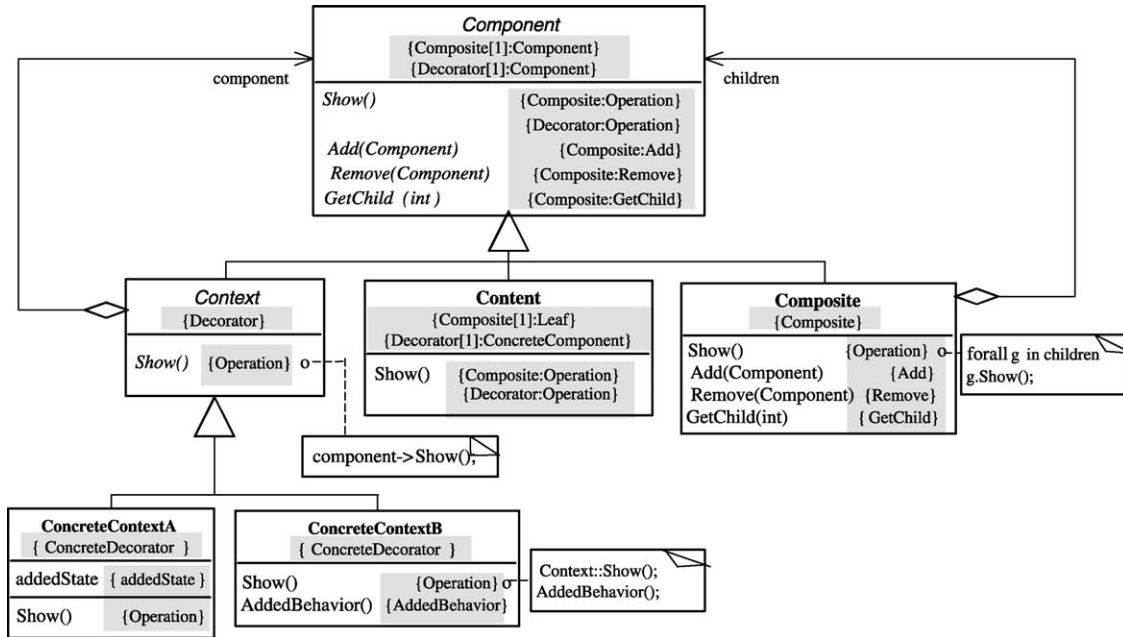


Fig. 7. Tagged pattern annotation with shading.

defined mainly by applying the UML built-in extensibility mechanisms. This extension forms a basis for a new UML profile [3], 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}’.

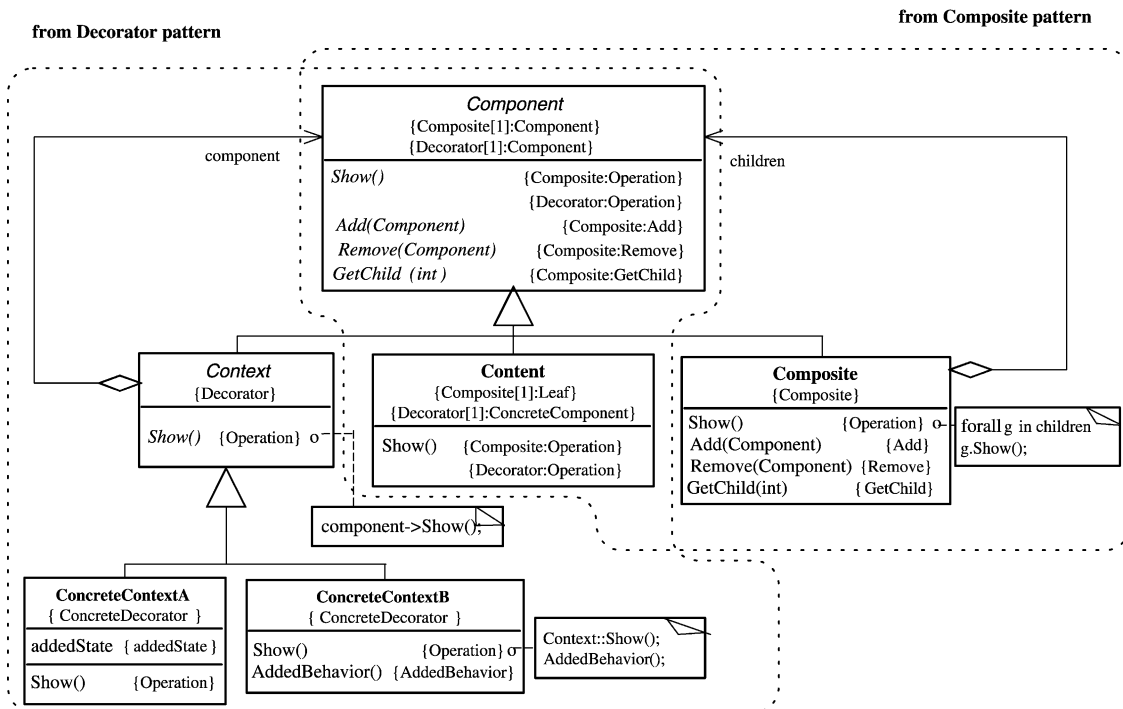


Fig. 8. Tagged pattern annotation with bounding.

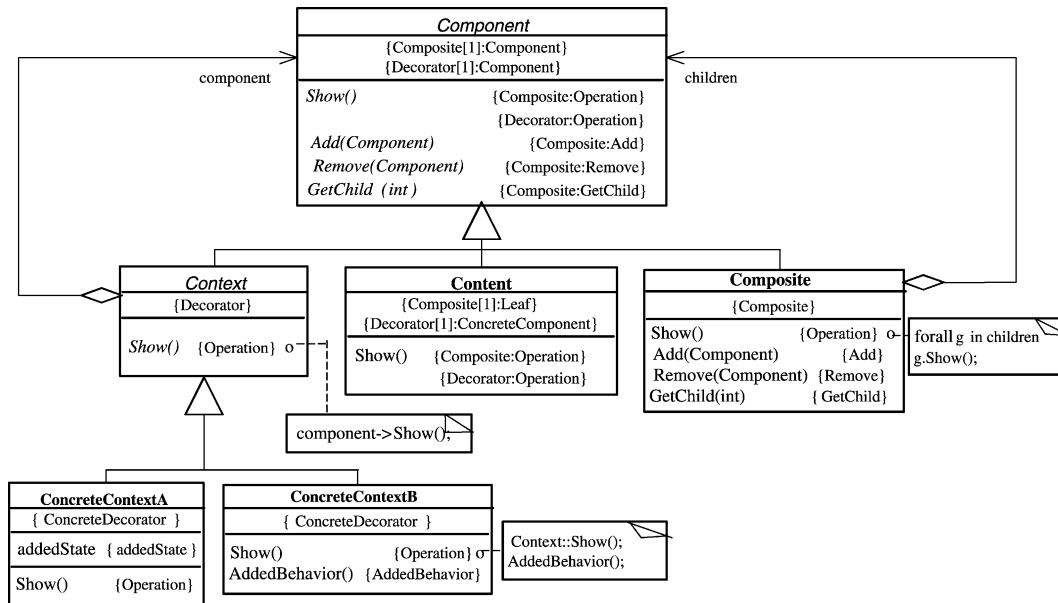


Fig. 9. Tagged pattern annotation with new compartments.

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) associated 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. Fig. 5 displays the class 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.

In addition to attaching new notations to class diagrams, we create new tagged values for the collaboration diagrams to represent the behavioral aspect of design patterns. The format of the tag is the same as that for structural aspect. Fig. 6 presents the collaboration diagram of the behavioral composition of the `Decorator` and the `Composite` patterns based on our notation, where x_1 and x_2 are two objects of the `ConcreteContext` class, which plays the role of `ConcreteDecorator` in the `Decorator` pattern; an instance (i) of the `Composite` class plays the role of `Composite` in the `Composite` pattern; n is an object of the `Content` class which participates both in the first instance of the `Composite` pattern as a leaf and in the first instance of the `Decorator` pattern as a `ConcreteComponent`.

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 2) by bounding each pattern with dashed circles so that the pattern boundaries are explicitly depicted as shown in Fig. 8.

Besides the two improvements to make pattern-related information explicit (see Fig. 7) and pattern boundaries explicit (see Fig. 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 Fig. 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.

³ If 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 Fig. 7. The behavioral aspect can be represented by shading the pattern-related information in the collaboration diagram. We have omitted the collaboration diagrams in this and the following cases for brevity.

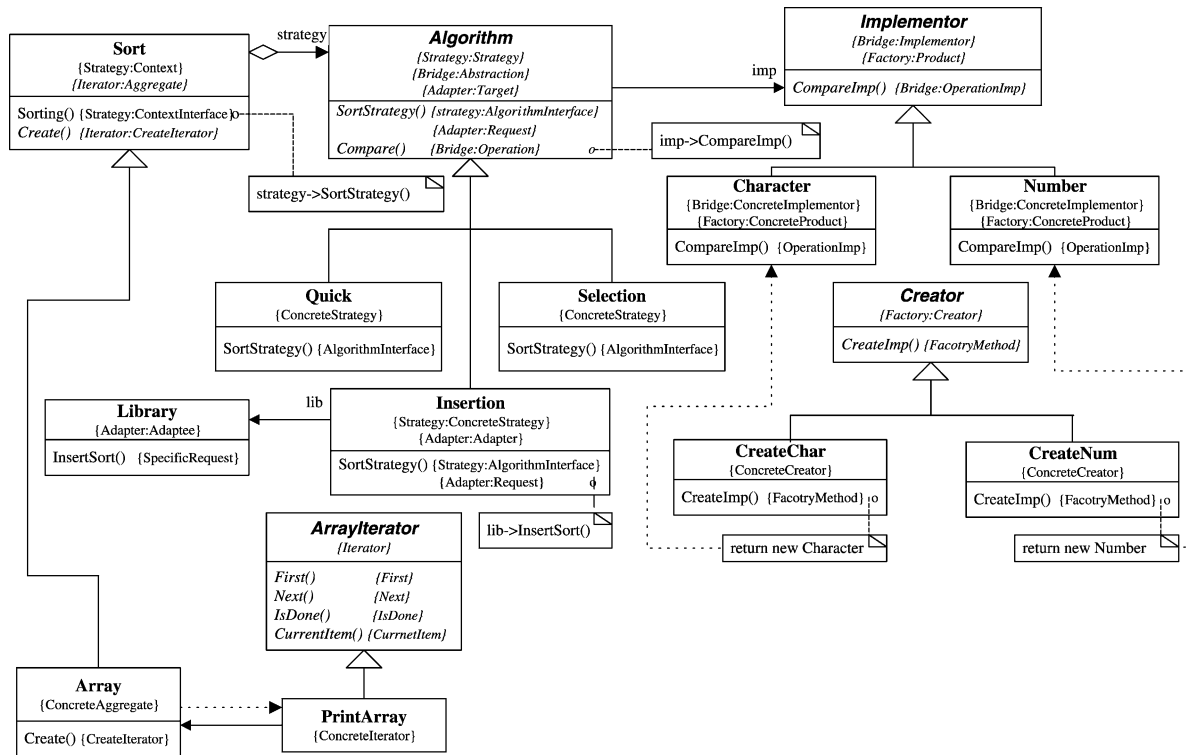


Fig. 10. A design of system sort in tagged pattern notation.

4. Case study

As an illustration of our notations to represent pattern compositions, 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 Fig. 1) contains five design patterns: Adapter, Bridge, Abstract Factory, 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 Abstract Factory 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 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.

Fig. 10 is the class diagram of 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 Fig. 1. 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.

⁴ Since 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. We omit the collaboration diagram of this design as well.

5. 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 class or collaboration 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 forced to communicate at the class and object level, instead of the pattern level. The design decisions and tradeoffs captured in the pattern are lost too. Therefore, the notations provided in this paper help on the explicit representation of design patterns and accomplishing the goals of design patterns.

References

- [1] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language user Guide*, Addison-Wesley, 1999.
- [2] J. Dong, P. Alencar, D. Cowan, Ensuring structure and behavior correctness in design composition, *Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS)*, Edinburgh UK April (2000) 279–287.
- [3] D. D’Souza, A. Sane, A. Birchenough, First class extensibility for UML-packaging of profiles, stereotypes, patterns, *Proceedings of the Second International Conference on the Unified Modeling Language (UML)*, LNCS1723, Springer, 1999, p. 265–277.
- [4] M. Fontoura, W. Pree, B. Rumpe, UML-F: a modeling language for object-oriented frameworks, *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)* July (2000) 63–82.
- [5] E. Gamma, R. Helm, R. Johnson, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [6] A. Lauder, S. Kent, Precise visual specification of design patterns, *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)* July (1998) 114–134.
- [7] G. Rossi, D. Schwabe, A. Garrido, Design reuse in hypermedia applications development, *Proceedings of the ACM International Conference on Hypertext* April (1997) 57–66.
- [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.