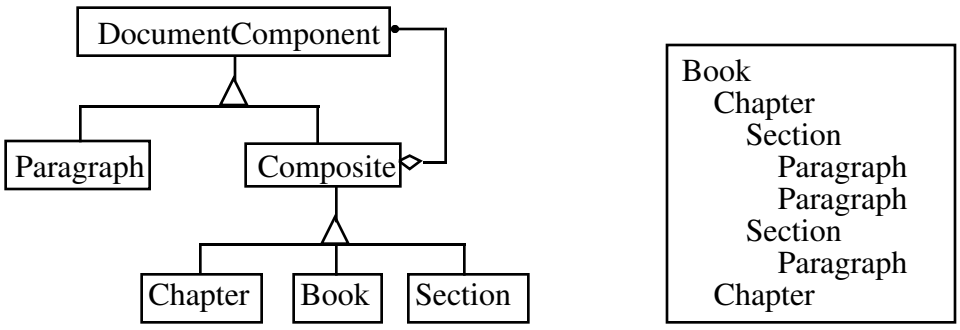
Patterns of Thought **Patterns Work in Teams**

My first column introduced patterns and described one of the most common object-oriented design patterns, Composite. I claimed that experts organized their knowledge as patterns, and that if we wrote down the patterns that we used and gave names to them then we could communicate our models and designs more efficiently, we could think about them better, and we could train new analysts and designers faster. However, I only gave one example of a pattern, and many of you probably didn't think it was very compelling. A single example is rarely compelling, and even less so with patterns, because the power of patterns comes in the way sets of patterns work together, not in the power of a single pattern. This column will describe how a set of patterns works together.

Each pattern solves a particular problem. Composite solved the problem of how to represent part-whole hierarchies with as little duplication as possible by making the composite be like the components. The Decorator pattern makes it easy to add an attribute to an object at run-time. The Strategy pattern makes it easy to change the way that an object responds to a particular request. Once you learn a large enough set of patterns, you will find that design becomes easier because most of the design problems you face can be solved with one of the design patterns that you already know. Thus, you don't have to solve your problems from first principles, but can reuse old solutions.

An Example

Consider a document processing system in which a document might contain chapters, sections, subsections, figures, paragraphs, and so on. One way to represent a book is to make a Book class, a Chapter class, a Paragraph class, and so on. Each Book would contain a list of Chapters, and a Chapter would contain a list of Sections, Figures, and Paragraphs. Since a book consists of a large whole-part hierarchy, it makes sense to apply the Component pattern. The result would be a DocumentComponent class with subclasses Paragraph, Figure, and DocumentComposite, where Chapter, Section, and the other composite classes would all be subclasses of DocumentComposite. Applying the Component pattern reduces the duplication between the various classes and makes it easier to mix and match components to make a book, but it doesn't reduce the number of classes.

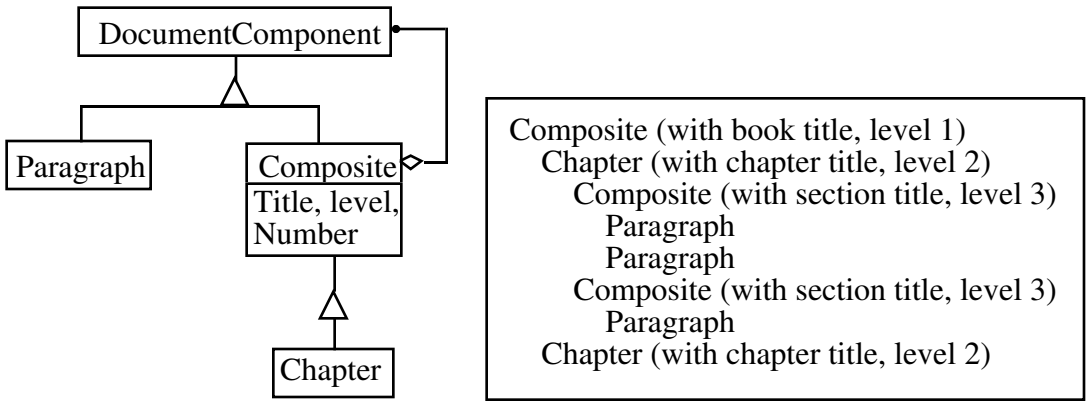


The DocumentComponent class and instance hierarchy

However, it is possible to reduce the number of subclasses of DocumentComponent substantially. Consider a chapter, a section, and a subsection. All of them have a title. They can all be optionally numbered. A chapter starts on a new page, while the others don't. A chapter is level 1 in the tree, while a section is level 2 and a subsection is level 3. But most of their functionality consists of having components that are paragraphs, figures,

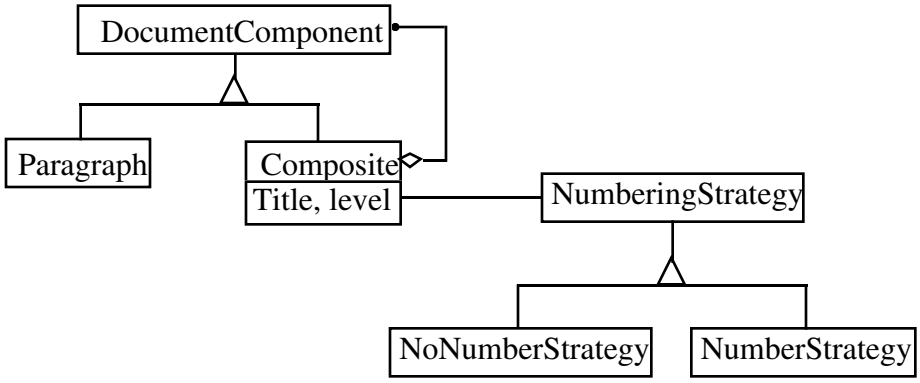
etc., and that is inherited from DocumentComposite. Thus, they will not necessarily be very different.

One solution is to give DocumentComposite a title, level, and optional number. Sections and subsections would then be normal instances of DocumentComposite. Chapter would need to be a subclass of DocumentComposite so that a chapter would also start a new page. However, most of the other classes of DocumentComposite would no longer be needed.



Distinguishing components by attributes, not classes

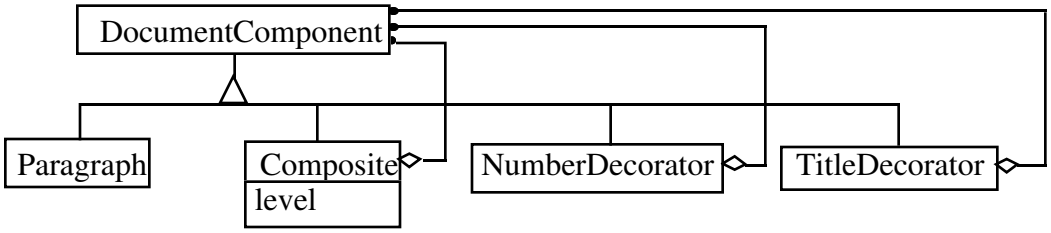
How do we specify whether a DocumentComposite is numbered? One alternative is for each one to have a Boolean that indicates whether it is numbered. But a better solution is for a DocumentComposite to contain an object that is responsible for keeping track of the numbering scheme. An object that represents an algorithm is called a strategy. There are at least two strategies for numbering parts of a document, one for parts that are numbered (NumberStrategy) and one for parts that are unnumbered (NoNumberStrategy). A part's NumberStrategy can also keep track of its number. This way we only have to allocate space for a variable when it is needed.



Class hierarchy using Strategy

A third way to specify whether a DocumentComposite is numbered is to use the Decorator pattern. Instead of making the algorithm for numbering a part be stored in the part, we can `decorate' the part with the algorithm. A DocumentComponent decorator is a subclass of DocumentComponent that has a DocumentComponent as a component. It will add an attribute or change a property of its component. In this case, NumberDecorator will add a number to the component that it decorates.

Earlier, we made the title be a part of the DocumentComposite. But the title could be a decoration, too. We could either make the title be part of the NumberDecorator, since we never number a component unless it has a title, or we could have a separate TitleDecorator class. We'll probably need a separate TitleDecorator class because many components with titles are never numbered. It might make sense to give every NumberDecorator a title anyway, but we will not do that to ensure that we see an example of one of the strengths of the Decorator pattern, which is the ability to recursively embed decorators.



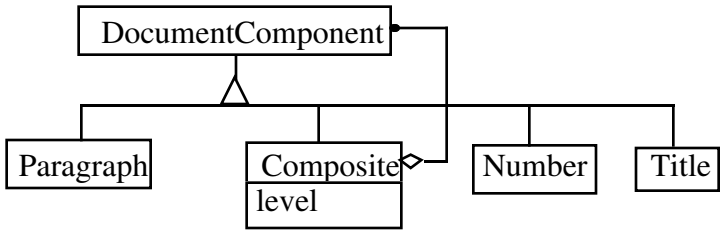
Class hierarchy using Decorator

```
TitleDecorator (with book title)
   Composite (with level 1)
      TitleDecorator (with chapter title)
         NumberDecorator
            Composite (with level 2)
               TitleDecorator (with section title)
                  NumberDecorator
                     Composite (with level 3)
                        Paragraph
                        Paragraph
...
            Composite (with level 2)
               TitleDecorator
                  NumberDecorator
                     Composite (with level 3)
                        Paragraph
                        Paragraph
      TitleDecorator
         NumberDecorator
            Composite (with level 2)
```

Instance hierarchy using Decorator

Decorator and Strategy are classic patterns that are alternatives to each other. The main advantage of Decorator is that it lets you customize every component and it can be more space efficient when components are usually not decorated. The Decorator pattern makes it easy to put a number and title on any component, such as a paragraph or a picture, not just on a composite. The main disadvantage is that recursive decorators can be hard to understand, especially when they are several layers deep. The Strategy pattern usually results in designs that are easier to understand. If you never want to give a paragraph a title and if every section and subsection has a title then it is probably better to use the Strategy pattern.

There is a fourth alternative, which is to make the title and number be a normal DocumentComponent that is the first element of a chapter or section. This alternative can't be used when the property you want to add must apply to the entire component. However, titles and numbers always show up at the beginning of a section, so this alternative will work fine for representing titles.



Class hierarchies using normal DocumentComponents

```
Composite (with level 1)
   TitleDecorator (with book title)
   Composite (with level 2)
      TitleDecorator (with chapter title)
      NumberDecorator
      Composite (with level 3)
           TitleDecorator (with section title)
            NumberDecorator
            Paragraph
            Paragraph
...
         Composite (with level 3)
              TitleDecorator
               NumberDecorator
               Paragraph
               Paragraph
    Composite (with level 2)
      TitleDecorator
      NumberDecorator
```

Instance hierarchies using normal DocumentComponents

Which of these alternatives is best? As usual, it depends on what you want to do with the design. For example, a design based on the Decorator or Strategy pattern makes it easier to implement an outline mode that lets you move a section by dragging its title. A design based on making the title be a normal DocumentComponent would make it easy for the user to decide not to have a section title by deleting the title. Nearly any feature could be implemented with any design. However, each design alternative makes some features easier to implement.

The Prototype Pattern

The last three designs all let you create new kinds of document components by composing existing ones. You can design a sub sub sub sub section by putting a DocumentComposite six levels down from the top of the document, instead of by making a new class. An editor

will probably have menu items for things like "add a section", and these menu items can't hardcode the class of the object to create because there will be no Section class. Instead, the menu item will be associated with an example or prototype of the object to be added. "Add a chapter" will copy the object being used as a prototype of chapters and add it to the document. You can make a new kind of object to be added to your document by creating an object and adding it to the menu as a prototype.

The Prototype pattern is common in compositional designs because it lets the user make new kinds of objects by composing existing. The major disadvantage of the Prototype pattern is that it requires that each object be able to copy itself, and complex objects must be able to copy all their components, ensuring that any relationships between the original objects are present between the copies. It also adds a little indirection to a design, because you will replace uses of class names like Chapter with uses of variable names that hold the equivalent prototypes. But unlike some design patterns, it does not add any inefficiency in time or space, nor does it increase the number of classes. Thus, it is an important and popular design pattern.

Pattern Languages

The usual name for a set of patterns is a "pattern language". A pattern language is more than just a set of patterns that work together, though. There is an order to a pattern language, and the goal is to be able to use a pattern language by going through it from beginning to end.

The patterns I've discussed (Composite, Decorator, Strategy, Prototype) are four of the twenty-three patterns in a book I'm writing with Erich Gamma, Richard Helm, and John Vlissides (Design Patterns: Elements of Reusable Object-Oriented Software). We should have the final copy off to the publisher before this column is printed.

Our patterns are not a pattern language because we do not impose an order on them. We describe how patterns like Decorator and Strategy are competitors, and we describe how patterns like Composite and Prototype work together, but we don't tell you the order that you should apply the patterns. One of the reasons is that we disagree about the order. For example, I tend to apply Prototype late in the design of a system, while John tends to apply it early. This is probably because he mostly uses C++, while I mostly use Smalltalk. Prototype is more important in C++ than in Smalltalk because it makes it easy to parameterize an object with the kinds of objects it will create, while Smalltalk programmers can use classes for that purpose. Probably each of us has our own pattern language, influenced partly by the environment we work in and our past experience. This makes it hard to develop a common pattern language.

It isn't obvious that there ever will be a complete pattern language for developing software. Writing a compiler is different from writing a word processor, and they are both different from writing an accounting system. Many principles apply to all three, of course, but the threory of parsing only applies to compilers, user interface design principles are more important to a word processor, while database design is important for an accounting system. Although we could try to combine patterns for all these kinds of designs into one pattern language, it seems like it would be better to keep them separate.

Patterns are Not Just for Design

The patterns I have discussed are all design patterns. It would be easy to think that patterns are just about design. But there are many kinds of patterns, and they don't have to be about designing object-oriented programs, or even programs. Christopher Alexander's "A

Pattern Language" is about designing communities and buildings. Hal Hildebrand has a pattern language for using Tensegrity, his company's OODBMS. Kent Beck is working on a pattern language for using CRC cards. Jim Coplien is working on a pattern language for designing software development organizations. I had a paper in OOPSLA'92 about a pattern language for using HotDraw. Patterns and pattern languages are useful ways of organizing all kinds of "how to" information.

I think that pattern languages will eventually play a central role in software engineering. We will use pattern languages to write architecture handbooks, user manuals for frameworks and other software packages, and specifications of our software processes. We will describe our systems in terms of an underlying pattern language.

But we aren't there yet. None of the pattern languages for software that I mentioned are as well-developed as they should be. Developing a pattern language is difficult. But even the relatively unorganized set of patterns in the design pattern catalog are extremely valuable. Even when patterns are not organized into a language, they still can be used together synergistically. As we organize them and add to them, they will become even more valuable.

Ralph E. Johnson
Department of Computer Science
1304 W. Springfield Ave.
Urbana, IL 61801
phone: 217-244-0093
e-mail: johnson@cs.uiuc.edu