Patterns of Thought

Analysis and design methods tend to focus more on notation and less on what is written in the notation. But experts not only know how to use a notation, they remember past models and designs, and they often solve a problem by recalling similar situations from the past and reusing old solutions in the new context. These problem/solution pairs tend to fall into families of similar problems and solutions, with each family exhibiting a pattern in both the problems and the solutions. An expert knows and uses a set of these patterns, learned from experience and from other people. This applies to experts in areas other than object-oriented analysis and design, of course, but it applies as much to us as it does to anyone else.

There have been people in the object-oriented community who have been trying to describe and organize patterns since the late 80's. My work has been especially influenced by that of Kent Beck, Ward Cunningham, and Erich Gamma. But recently a community of like-minded people has formed, mostly because of a series of OOPSLA workshops; "Towards an Architecture Handbook" organized by Bruce Anderson in '91, a continuation of that workshop and in one on "Patterns" that was organized by Peter Coad in '92, and a pattern workshop in '93 in which they joined forces. Kent Beck organized the Hillside Group to promote the explicit use of patterns in designing software, which is sponsoring a conference this August on Pattern Languages of Programs. Several books are in the works, and several people teach courses on patterns. Many groups are using patterns explicitly in software projects.

I intend to use this column to share some of the pattern-related work that is going on. The column will not only describe particular patterns that have been discovered, but will describe some of the issues involving patterns, such as how to use them, how to find them, and how they change the way we develop software. Since discovering and documenting patterns is a group activity, sometimes other members of the "pattern community" will write the column.

The Composite Pattern

All experts use patterns. But usually these patterns don't have names, are not written down, and so are hard to teach to non-experts. Giving these patterns names lets us talk about them, which is not only useful for teaching them to new-comers, but also lets experts quickly describe their models and designs to each other. Our patterns become a kind of language that we can use to document our systems and to think about our designs.

The Composite pattern is so common that probably all of you have used one form of it. But there are many variations of it, and some of them are better than others. There are also common mistakes that people make when they use the Composite pattern. If you know the pattern and the kind of mistakes that you are likely to make when you use it then you can check for those mistakes and so can catch errors earlier. Thus, using the Composite pattern explicitly can lead to more reliable systems, as well as systems that are easier to understand. After all, if you are going to use a pattern, you should use it correctly.

The problem that leads to Composite is that there are complex objects that form part-whole or containment hierarchies. For example, a document processing system might have documents that contain chapters that contain sections that contain paragraphs. A company might have divisions that contain departments that contain projects. A user interface might have windows that contain subwindows. All of these examples have part-whole hierarchies that can be arbitrarily deep.

The key to the Composite pattern is objects that form a tree. You won't necessarily want to use Composite to represent a car just because it contains an engine, wheels, and doors, but you might be if its engine contains pistons and spark plugs. When objects form deep part-whole hierarchies, we usually want to treat individual objects and composite objects in the same way. The Composite pattern lets us do so. Larry Williams told me that this pattern is taught at ObjectTime with the motto "Make a composite like its components". But knowing that you want to make a composite like its components is only half the battle; you also have to know how to do it.

Using the Composite Pattern

The first step in using the Composite pattern is to make three kinds of classes, Component, Composite, and Leaf. Figure 1 shows the relationship of these classes using OMT notation. Component is an abstract class that defines the interface of all components. Composite represents a component with components. Leaf represents a component without components. There are usually several Leaf classes. In a document processing system, the Component class might be called "TextElement", the Composite class "CompositeTextElement", and the Leaf classes might be "Paragraph" and "Figure".

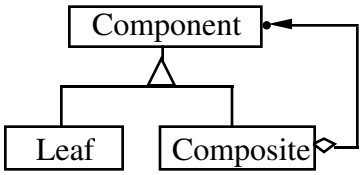

Figure 1: The Composite Pattern

The next step in using the Composite pattern is to make sure that the Leaf and Composite classes have the same interface, and to describe that interface with the Component class. There are two sides to this step, making the Composite class support all the operations that any other component supports, and making the Component class support operations that are needed to use Composite.

A Composite class usually implements component operations by delegating them to its components. It may only need to use a few of its components. For example, a text component probably needs to know its starting line number and its ending line number. A composite text element can use the starting line number of its first component as its own, and the ending line number of its last component as its own. More often a composite will implement an operation by iterating over its components, performing the same operation on each. For example, a composite text element will display itself by displaying its components.

Usually a composite supports operations to add a component, to remove a component, and to enumerate its components. Some of these must be supported by the component interface, too. The main composite operation that must be defined in class Component is iteration. It is easy to support, because a component that is not a composite will iterate over its components by iterating over the null set. Other composite operations are both harder to implement for leaf components and also less important. They are harder to implement because it isn't possible to add a component to a leaf without making it no longer a leaf, and deleting a component from a leaf is always an error. One solution is to define default versions of the add and remove operations in class Component that always generate a run-time error, but this defeats static type-checking. Fortunately, it is usually not necessary to implement these operations in Component because you can redesign your system so that

they are always performed on an object known to be a composite, never on just a component. For example, a text editor could have context sensitive menus so that the only place where you could add an element is in a composite. This not only solves a problem in the design of the program, it also makes the design of the user interface better.

Common Variations on the Composite Pattern

The main variation with the Composite pattern is whether a component knows its parent. If a component does not know its parent then the resulting design is simpler and it is possible for an object to be the component of several composites. In other words, the component hierarchy will be a directed acyclic graph instead of a tree. On the other hand, components often need to know their composite. For example, if a component knows its composite then we can delete an element from a composite by asking the component to delete itself instead of asking a composite to remove an component.

If each component knows its parent then

1) Component will have a "parent" or "container" variable that indicates its composite.
2) A component will ensure that it is removed from any previous parent before it is added to another.
3) When a composite adds a component to itself, it will always register itself as the component's parent.

This will solve one of the common errors with composite objects; one of the composite's components has the wrong parent.

Common Mistakes with the Composite Pattern

One common mistake is merging the Composite and Component classes. If the Component class defines a collection of components then every component becomes a composite. Although this eliminates the problem of deciding what part of the Composite interface will be supported by all components, it means that every leaf component can have components, which doesn't make sense. In fact, many of the component classes will never have components, but they will still inherit everything that they need to have components.

A classic example of this mistake is class View in the original Model/View/Controller. Instead of having a separate View and CompositeView class, View defined an instance variable for subviews, so each view could have subviews. Most subclasses of View, such as TextView and ListView, could never have subviews, but they did not enforce this restriction, so it was easy for a programmer to mistakenly give them subviews, with odd results. ParcPlace fixed this problem when they rewrote Model/View/Controller as part of Release 4.0, and so now CompositeView is a subclass of VisualComponent (VisualComponent is a generalization of View).

I made the same mistake in an early version of *Accounts,* a framework for accounting. Accounts can be nested, because accounts for particular inventory items can be nested in accounts for a department, which can be nested in accounts for stores, which can be nested in a single account for the entire company. In spite of the fact that I know the Composite pattern well, and have taught it to many people, we made it possible for every account to have components. When I realized that we should be using the Composite pattern, we quickly redesigned *Accounts* to have an abstract class Account with subclasses SimpleAccount and CompositeAccount. In fact, we ended up with two kinds of composite accounts and one leaf account (SimpleAccount), which is unusual, but still fits the pattern perfectly. This example shows that just knowing a pattern isn't enough, you have to

recognize when to apply it. Knowing the pattern makes it a lot easier to see where to apply it, of course, and it was not hard to apply once we saw we needed it.

Another common mistake is for the Composite class not to be a subclass of Component, but to be an unrelated class. This makes it impossible to recursively embed a composite inside a composite. This can lead to a proliferation of composite classes that are very similar to each other. For example, Chapter, Section, Subsection, and Subsubsection could all be different classes, or there could be a CompositeTextElementWithTitle class that could be used to implement all of them.

Sometimes it takes a conceptual breakthrough to realize that you should use the Composite pattern when the classes that should be composite and component are unrelated. For example, the RTL System is a compiler back-end written in Smalltalk that represents a program as a collection of register transfers. It converts register transfers into machine instructions by combining register transfers and checking whether they are a legal machine instruction. An early version of the RTL System had three kinds of classes for representing register transfers; RegisterTransfer represented a single register transfer, RTLList represented a list of register transfers, and RTLRecord represented a list of register transfers that matched a machine instruction. In addition to an RTLList, an RTLRecord kept track of the machine instruction that the register transfers matched and the dependencies between the register transfers in the RTLList and those in other RTLRecords. The design was greatly simplified when we realized that most RTLLists had only one register transfer in them, and we should treat the rest as exceptions by representing them with CompositeRegisterTransfer. This let us realize that dependencies and machine instructions should really be the responsibility of RegisterTransfers, thereby eliminating two kinds of objects from the design and reducing the size of the design.

Note that both "Everything is a composite" and "Composite and component are different" are not really faulty versions of the Composite pattern, they are just different patterns. "Everything is a composite" works well when you don't want different kinds of elements to have different behaviors, and so want to represent all elements with a universal "tree node" class. When you start making different kinds of elements then some of them will never have components and it will be confusing for them to be a subclass of a class that does. "Composite and component are different" usually models the world better than the Composite pattern; skin, blood, arms and cells are all part of a human body, but it is pretty artificial to have an abstract "body part" class. Giving composites and components the same interface makes the systems that use them be more extensible and makes those parts be more reusable. Thus, we use the Composite pattern when we are more concerned with making our artificial world be powerful than with trying to perfectly model the real world.

Conclusions

Nearly every large object-oriented system I've seen has used the Composite pattern. It was rarely pointed out . Instead, I had to reverse-engineer it from the design. It is a lot easier for me to understand a system once I can see patterns like this in it, because each pattern explains and influences a large part of the system.

There are many advantages to explicitly using patterns like Composite. It helps us document the systems we build. It helps us build better systems. It prevents common mistakes. But we will lose these advantages if we don't name and describe these patterns and expect people to learn them by example instead of by teaching them directly.

Postlude

The Composite pattern is described in *Design Patterns: Elements of Object-Oriented Software Architecture* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, to appear later in 1994. There is an Internet mailing list on patterns; contact patterns-request@cs.uiuc.edu to join. There is also a conference on Pattern Languages of Programs August 5-7, 1994. Ask me for details.

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