Composition versus inheritance

Leading-Edge Java
Design Principles from Design Patterns
A Conversation with Erich Gamma, Part III
by Bill Venners
June 6, 2005

<< **Page 4 of 4**

Bill Venners: The other principle of object-oriented design that you offer in the GoF introduction is, "Favor object composition over class inheritance." What does that mean, and why is it a good thing to do?

Erich Gamma: I still think it's true even after ten years. Inheritance is a cool way to change behavior. But we know that it's brittle, because the subclass can easily make assumptions about the context in which a method it overrides is getting called. There's a tight coupling between the base class and the subclass, because of the implicit context in which the subclass code I plug in will be called. Composition has a nicer property. The coupling is reduced by just having some smaller things you plug into something bigger, and the bigger object just calls the smaller object back. From an API point of view defining that a method can be overridden is a stronger commitment than defining that a method can be called.

In a subclass you can make assumptions about the internal state of the superclass when the method you override is getting called. When you just plug in some behavior, then it's simpler. That's why you should favor composition. A common misunderstanding is that composition doesn't use inheritance at all. Composition is using inheritance, but typically you just implement a small interface and you do not inherit from a big class. The Java `listener` idiom is a good example for composition. With listeners you implement a listener interface or inherit from what is called an adapter. You create a listener object and register it with a `Button` widget, for example. There is no need to subclass `Button` to react to events.

Bill Venners: When I talk about the GoF book in my design seminar, I mention that what shows up over and over is mostly using composition with interface inheritance for different reasons. By interface inheritance I mean, for example, inheriting from pure

virtual base classes in C++, or code font `interface` inheritance in Java. The `Listener` example you mention, for instance, has inheritance going on. I implement `MouseListener` to make `MyMouseListener`. When I pass an instance to a `JPanel` via `addMouseListener`, now I'm using composition because the front-end `JPanel` that's holding onto that `MouseListener` will call its `mouseClicked` method.

Erich Gamma: Yes, you have reduced the coupling. In addition you now have a separate listener object and you might even be able to connect it with other objects.

Bill Venners: That extra flexibility of composition over inheritance is what I've observed, and it's something I've always had difficulty explaining. That's what I was hoping you could capture in words. Why? What is really going on? Where does the increased flexibility really come from?

Erich Gamma: We call this black box reuse. You have a container, and you plug in some smaller objects. These smaller objects configure the container and customize the behavior of the container. This is possible since the container delegates some behavior to the smaller thing. In the end you get customization by configuration. This provides you with both flexibility and reuse opportunities for the smaller things. That's powerful. Rather than giving you a lengthy explanation, let me just point you to the `Strategy` pattern. It is my prototypical example for the flexibility of composition over inheritance. The increased flexibility comes from the fact that you can plug-in different strategy objects and, moreovers, that you can even change the strategy objects dynamically at run-time.

Bill Venners: So if I were to use inheritance...

Erich Gamma: You can't do this mix and match of strategy objects. In particular you cannot do it dynamically at run-time. 

Next week

Come back Monday, June 13th for the next installment of this conversation with Erich Gamma. If you'd like to receive a brief weekly email announcing new articles at Artima Developer, please subscribe to the Artima Newsletter.

Talk back!

Have an opinion about the design patterns topics discussed in this article? Discuss this article in the Articles Forum topic, Design Principles from Design Patterns.

Resources

[1] Erich Gamma is co-author of *Design Patterns: Elements of Reusable Object-Oriented Software*, which is available on Amazon.com at: http://www.amazon.com/exec/obidos/ASIN/0201633612/

[2] Erich Gamma is co-creator of JUnit, the defacto standard Java unit testing tool: http://www.junit.org/index.htm

[3] Erich Gamma leads the Java development effort for the Eclipse tool platform: http://www.eclipse.org/

[4] See "Extension Object," in Robert Martin, *Pattern Languages of Program Design 3*. Addison- Wesley, 1997, available on Amazon.com at: http://www.amazon.com/exec/obidos/ASIN/0201310112/

[5] "Evolving Java-based APIs," by Jim des Rivières: http://eclipse.org/eclipse/development/java-api-evolution.html

[See also] *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*, by Erich Gamma and Kent Beck, is available on Amazon.com at: http://www.amazon.com/exec/obidos/ASIN/0321205758/

About the author

Bill Venners is president of Artima Software, Inc. and editor-in-chief of Artima Developer. He is author of the book, *Inside the Java Virtual Machine*, a programmer-oriented survey of the Java platform's architecture and internals. His popular columns in JavaWorld magazine covered Java internals, object-oriented design, and Jini. Bill has been active in the Jini Community since its inception. He led the Jini Community's ServiceUI project, whose ServiceUI API became the de facto standard way to associate user interfaces to Jini services. Bill also serves as an elected member of the Jini Community's initial Technical Oversight Committee (TOC), and in this role helped to define the governance process for the community.