

## When to think about interfaces

Leading-Edge Java

Design Principles from Design Patterns

A Conversation with Erich Gamma, Part III

by Bill Venners

June 6, 2005

[<<](#) **Page 3 of 4** [>>](#)

Advertisement

**Bill Venners:** On the subject of interfaces, the GoF book includes some UML class diagrams. UML diagrams seem to mix interface and implementation. When you look at it, you often see the design of the code. It isn't necessarily obvious what's API and what's implementation. If you look at JavaDoc, by contrast, you see the interfaces. Another place I see the lack of differentiation between interface and implementation is XP. XP talks about the code. You're changing this amorphous body of code with test-driven development. When should the designer think about interfaces versus the whole mass of code?

**Erich Gamma:** You might think differently when you design an application than when you design a platform. When you design a platform, you have to care at every moment about what to expose as part of your API, and what to keep internal. Today's refactoring support makes it trivial to change names so you have to be careful not to change published APIs by accident. This goes beyond just defining which types are published. You also have to answer questions like: do you allow clients to subclass from this type? If you do, it imposes big obligations. If you look at the Eclipse API, we try to make it very explicit whether we intend that clients subclass from a type. Also with Jim des Rivières [5] we have an API advocate in our team. He helps us not only to comply with our rules but even more importantly Jim helps us to tell a consistent story with our APIs.

When it comes to applications, even there you have abstractions that have multiple variations. For your design you want to come up with key abstractions, and then you want to have your other code just interact with those abstractions, not with the specific implementations. Then you have flexibility. When a new variation of an abstraction comes up, your code still works. Regarding XP, as I mentioned earlier, the modern refactoring tools allow you to introduce interfaces into existing code easily and

therefore is consistent with XP.

**Bill Venners:** So for an application it's the same thought process as for a platform, but on a smaller scale. Another difference is that it is easy for me if I have control of all the clients to this interface I can update them if I need to change the interface.

**Erich Gamma:** Yes, for an application it is the same thought process as for a platform. You also want to build an application so that it lasts. Reacting to a changing requirement shouldn't ripple through the entire app. The fact that you have control over all the clients helps. Once you have given out your code and you no longer have access to all the clients, then you're in the API business.

**Bill Venners:** Even if those clients are written by a different group in the same company.

**Erich Gamma:** Even there, absolutely.

**Bill Venners:** So it sounds like thinking about interfaces becomes more important as the project scales up. If the project is just two or three people, it is not quite as important to think about the interfaces, because if you need to change them you change them. The refactoring support tools...

**Erich Gamma:** ... will do it all for you.

**Bill Venners:** But if it is a 100-person team, that means people will be partitioned into groups. Different groups will have different areas of responsibility.

**Erich Gamma:** A practice that we follow is to assign a component to a group. The group is responsible for the component and publishes its API. Dependencies are then defined through API. We also resist the temptation to define friend relationships, that is, where some components are more equal than others and are allowed to use internals. In Eclipse all components are equal. For example, the Java development tool plug-ins have no special privileges and use the same APIs as all other plug-ins.

Once you have published an API then it is your responsibility to keep them stable. Otherwise you will break the other components and nobody is able to make progress. Having stable APIs is a key to making progress in a project of this size.

In a closed environment as you have described it you have more flexibility when it comes to making changes. For example, you can use the Java deprecation support to allow other teams to gradually catch-up with your changes. In such an environment

you can remove deprecated methods after some agreed on time-interval. This might not be possible in a fully exposed platform. There deprecated methods cannot be removed since you may still break a client somewhere.

[<<](#) **Page 3 of 4** [>>](#)

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**

Advertisement

**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, moreover, 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](http://www.artima.com/lejava/articles/designprinciples3.html).

## 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.