Design Principles from Design Patterns

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

Page 1 of 4 >>

Summary

In this interview, Erich Gamma, co-author of the landmark book, *Design Patterns*, talks with Bill Venners about two design principles: program to an interface, not an implementation, and favor object composition over class inheritance.

Erich Gamma lept onto the software world stage in 1995 as co-author of the best-selling book *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995) [1]. This landmark work, often referred to as the Gang of Four (GoF) book, cataloged 23 specific solutions to common design problems. In 1998, he teamed up with Kent Beck to produce JUnit [2], the de facto unit testing tool in the Java community. Gamma currently is an IBM Distinguished Engineer at IBM's Object Technology International (OTI) lab in Zurich, Switzerland. He provides leadership in the Eclipse community, and is responsible for the Java development effort for the Eclipse platform [3].

On October 27, 2004, Bill Venners met with Erich Gamma at the OOPSLA conference in Vancouver, Canada. In this interview, which will be published in multiple installments in *Leading-Edge Java* on Artima Developer, Gamma gives insights into software design.

- In Part I: How to Use Design Patterns, Gamma describes gives his opinion on the appropriate ways to think about and use design patterns, and describes the difference between patterns libraries, such as GoF, and an Alexandrian pattern language.
- In Part II: Erich Gamma on Flexibility and Reuse, Gamma discusses the importance of reusability, the risks of speculating, and the problem of frameworkitis.
- In this third installment, Gamma discusses two design principles highlighted in

the GoF book: program to an interface, not an implementation, and favor object composition over class inheritance.

Program to an interface, not an implementation

Bill Venners: In the introduction of the GoF book, you mention two principles of reusable object-oriented design. The first principle is: "Program to an interface, not an implementation." What's that really mean, and why do it?

Erich Gamma: This principle is really about dependency relationships which have to be carefully managed in a large app. It's easy to add a dependency on a class. It's almost too easy; just add an import statement and modern Java development tools like Eclipse even write this statement for you. Interestingly the inverse isn't that easy and getting rid of an unwanted dependency can be real refactoring work or even worse, block you from reusing the code in another context. For this reason you have to develop with open eyes when it comes to introducing dependencies. This principle tells us that depending on an interface is often beneficial.

Bill Venners: Why?

Erich Gamma:Once you depend on interfaces only, you're decoupled from the implementation. That means the implementation can vary, and that's a healthy dependency relationship. For example, for testing purposes you can replace a heavy database implementation with a lighter-weight mock implementation. Fortunately, with today's refactoring support you no longer have to come up with an interface up front. You can distill an interface from a concrete class once you have the full insights into a problem. The intended interface is just one 'extract interface' refactoring away.

So this approach gives you flexibility, but it also separates the really valuable part, the design, from the implementation, which allows clients to be decoupled from the implementation. One question is whether you should always use a Java `interface`s for that. An abstract class is good as well. In fact, an abstract class gives you more flexibility when it comes to evolution. You can add new behavior without breaking clients.

Bill Venners: How's that?

Erich Gamma: In Java when you add a new method to an `interface`, you break all your clients. When you have an abstract class, you can add a new method and provide a default implementation in it. All the clients will continue to work. As always there is a trade-off, an interface gives you freedom with regard to the base class, an abstract class gives you the freedom to add new methods later. It isn't always possible to define an interface in an abstract class, but in the light of evolution you should consider whether an abstract class is sufficient.

Since changing interfaces breaks clients you should consider them as immutable once you've published them. As a consequence when adding a new method to an interface you have to do so in a separate interface. In Eclipse we take API stability seriously and for this reason you will find so called I*2 interfaces like `IMarkerResolution2` or `IWorkbenchPart2` in our APIs. These interfaces add methods to the base interfaces `IMarkerResolution` and `IWorkbenchPart`. Since the additions are done in separate extension interfaces you do not break the clients. However, there is now some burden on the caller in that they need to determine at run- time whether an object implements a particular extension interface.

Another lesson learned is that you should focus not only on developing version one, but to also think about the following versions. This doesn't mean designing in future extensibility, but just keeping in mind that you have to maintain what you produce and try to keep the API stable for a long time. You want to build to last. That's been an important theme of Eclipse development since we started. We have built Eclipse as a platform. We always keep in mind as we design Eclipse that it has to last ten or twenty years. This can be scary at times.

We added support for evolution in the base platform when we started. One example of this is the `IAdaptable` interface. Classes implementing this interface can be adapted to another interface. This is an example of the Extension Object pattern,. [4]

Bill Venners: It's funny nowadays we're so much more advanced, but when we say build to last, we mean ten or twenty years. When the ancient Egyptians built to last, they meant...

Erich Gamma: Thousands of years, right? But for Eclipse, ten to twenty years, wow. Quiet honestly, I don't envision a software archeologist finding an Eclipse installation stored somewhere on a hard disk in ten or twenty years. I really mean that Eclipse should still be able to support an active community in ten or twenty years.

Page 1 of 4 >>

Leading-Edge Java

Design Principles from Design Patterns
A Conversation with Erich Gamma, Part III

by Bill Venners

June 6, 2005

<< **Page 2 of 4** >>

Bill Venners: Above you said interfaces are more valuable. What is their value? Why are they more valuable than the implementation?

Erich Gamma: An interface distills the collaboration between objects. An interface is free from implementation details, and it defines the vocabulary of the collaboration. Once I understand the interfaces, I understand most of the system. Why? Because once I understand all the interfaces, I should be able to understand the vocabulary of the problem.

Bill Venners: What do you mean by "vocabulary of the problem?"

Erich Gamma: What are the method names? What are the abstractions? The abstractions plus the method names define the vocabulary. The Java Collections package is a good example for this. The vocabulary for how to work with collections is distilled in interfaces like `List` or `Set`. There is a rich set of implementations for these interfaces, but once you understand the key interfaces you get them all.

Bill Venners: I guess the core of my question about "program to an interface, not to an implemenation," is this: In Java, there's a special kind of class called `interface` that if I'm writing I put in code font—the Java `interface` construct. But then there's the object-oriented interface concept, and every class has that object-oriented interface concept.

If I'm writing client code and need to use an object, that object's class exists in some type hierarchy. At the top of the hierarchy, it's very abstract. At the bottom, it's very concrete. The way I think about programming to interfaces is that, as I write client code, I want to write against the interface of the type that's as far up that hierarchy as I can go, without going too far. Every single one of those types in the hierarchy has a contract.

Erich Gamma: You're right. And, writing against a type far up in the hierarchy is consistent with the programming to an interface principle.

Bill Venners: How can I write to an implementation?

Erich Gamma: Imagine I define an `interface` with five methods, and I define an implementation class below that implements these five methods and adds another ten methods. If only the interface is published as API then if you call one of these ten methods you make an internal call. You call a method that is out of contract, which I might break anytime. So it's the difference, as Martin Fowler would say, between public and published. Something can be public, but that doesn't mean you have *published* it.

In Eclipse we have the naming convention that a package which includes the segment "internal" identifies internal packages. They contain types which we do not consider published types even when the package includes a public type. So the nice short package name is for API and the long name is for internals. Obviously using package private classes and interfaces is another way to hide implementation types in Java.

Bill Venners: Now I understand what you mean. There's public and published. Martin Fowler has nice terms for that difference.

Erich Gamma: And in Eclipse we have the conventions for that difference. Actually we even have tool support. In Eclipse 3.1 we have added support for defining rules for which packages are published API. These access rules are defined on a project's class path. Once you have these access restrictions defined the Eclipse Java development tools report access to internal classes in the same way as any other compiler warnings. For example you get feedback as you type when you add a dependency to a type that isn't published.

Bill Venners: So if I write code that talks to the interface of a non-published class, that's a way I am writing to the implementation, and it may break.

Erich Gamma: Yes, and the explanation from the angle of the provider is that I need some freedom and reserve the right to change the implementation.

[<<](#) **Page 2 of 4** [>>](#)

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** >>

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**

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.

<< **Page 4 of 4**