Erich Gamma on Flexibility and Reuse  
A Conversation with Erich Gamma, Part II  
by Bill Venners  
May 30, 2005

**Summary**  
Developers are often faced with decisions about how much flexibility to design into their software. In this interview, Erich Gamma, co-author of the landmark book, *Design Patterns*, talks with Bill Venners about the attitude he believes developers should adopt towards flexibility and reuse.

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](http://www.artima.com/lejava/articles/reuse3.html#resources)]. 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](http://www.artima.com/lejava/articles/reuse3.html#resources)], 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](http://www.artima.com/lejava/articles/reuse3.html#resources)].

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](http://www.artima.com/lejava/articles/gammadp.html), 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 this second installment, Gamma discusses the importance of reusability, the risks of speculating, and the problem of frameworkitis.

The importance of reusability

**Bill Venners**: The first sentence of the GoF book is, "Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder." How important is reusability?

**Erich Gamma**: Nobody ships applications today without reusing system level class libraries. Our environments are too complex to build applications without them. It is obvious that reuse is important, we just do it. An interesting question is whether there is a larger scale reuse than system level class libraries. When we started exploring frameworks years ago, we had big hopes for them. We thought the way to create software was to build high-level, domain-specific frameworks, and then you just customize them and reuse all the design that is codified into them. That was the reuse nirvana. Since then, I've gotten a little more realistic, because I have learned that it's hard to create highly reusable frameworks. They become complex, hard to learn, and even harder to maintain. I was on both the framework consumer and the framework producer side, and it can be hard from either perspective.

A key challenge in framework development is how to preserve stability over time. The more miles a framework gets the better you understand how you should have built it in the first place. Therefore you would like to tweak and improve it. However, since your framework is heavily used you are highly constrained in what you can change. At this point it is crucial to have well defined APIs and to make it clear to the clients what is published API and what internal code is. For published APIs you should commit to stability and for internal code you have the freedom to change it.

A good example of how I like to see reuse at work is Eclipse. It's built of components we call plug-ins. A plug-in bundles your code and there is a separate manifest where you define which other plug-ins you extend and which points of extension your plug-in offers. Plug-ins provide reusable code following explicit conventions to separate API from internal code. The Eclipse component model is simple and consistent too. It has this kernel characteristic. Eclipse has a small kernel, and everything is done the same way via extension points. The combination of a component model and the focus on APIs is one of the key ingredients of Eclipse. Controlled extensibility is another important one.

**Bill Venners**: What do you mean by controlled extensibility?

**Erich Gamma**: The first object-oriented thing they told us was: OO is way cool. You can subclass and customize anything!" These days I consider this the attitude of object exhibitionists. You can go and expose everything, and people can change anything. The original Smalltalks had some of this flavor. The problems start when the next version comes along. If you have exposed everything, you cannot change anything or you break all your clients. That's why the component model, the API focus, designing what's internal and what's published, becomes so critical when it comes to reuse. Also when you study the Eclipse API you will find that we go further than just specifying which classes are published API. The Eclipse API also specifies whether a class is intended to be subclassed. A key lesson here is that API is not just a documented class. And, APIs don't just happen; they are a big investment.

The other important aspect of what we call controlled extensibility in the context of Eclipse plug-ins is extensions and extension points. An extension point defines where you can contribute to, and extend, a plug-in. An extension point has a name and comes with a specification that defines the interfaces you have to conform to when you make a contribution.. As a plug-in writer you are not done until you have also thought about the extension points your plug-in might offer.

**Bill Venners**: Define framework.

**Erich Gamma**: I see three levels of reuse. At the lowest level, you reuse classes: class libraries, containers, maybe some class "teams" like container/iterator. Frameworks are at the highest level. They really try to distill design. They identify the key abstractions for solving a problem. They represent them by classes and define relationships between them. JUnit is a small framework, for example. It is the "Hello, world" of frameworks. You have Test, TestCase, TestSuite and relationships defined. A framework is larger grained than just a single class, typically. Also, you hook into frameworks by subclassing somewhere. They use the so-called Hollywood principle of "don't call us, we'll call you." The framework allows you to define your custom behavior, and it will call you back when it's your turn to do something. Same with JUnit, right? It calls you back when it wants to execute a test for you, but the rest is done in the framework.

**Bill Venners**: You said there were three levels of reuse?

**Erich Gamma**: Right, there also is a middle level. This is where I see patterns. Design patterns are both smaller and more abstract than frameworks. They're really a description about how a couple of classes can relate to each other and interact with each other. The level of reuse increases when you move from classes to patterns and finally frameworks.

What is nice about this middle layer is that patterns offer reuse in a way that is less risky than frameworks. Building a framework is high risk and a significant investment. Patterns allow you to reuse design ideas and concepts independent of concrete code.

# The risk of speculating

**Bill Venners**: The GoF book says, "The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so they can evolve accordingly. To design a system so that it's robust to such changes, you must consider how the system might need to change over its lifetime. A design that doesn't take change into account risks major redesign in the future." That seems contradictory to the XP philosophy.

**Erich Gamma**: It contradicts absolutely with XP. It says you should think ahead. You should speculate. You should speculate about flexibility. Well yes, I matured too and XP reminds us that it is expensive to speculate about flexibility, so I probably wouldn't write this exactly this way anymore. To add flexibility, you really have to be able to justify it by a requirement. If you don't have a requirement up front, then I wouldn't put a hook for flexibility in my system up front.

But I don't think XP and patterns are conflicting. It's how you use patterns. The XP guys have patterns in their toolbox, it's just that they refactor to the patterns once they need the flexibility. Whereas we said in the book ten years ago, no, you can also anticipate. You start your design and you use them there up-front. In your up-front design you use patterns, and the XP guys don't do that.

**Bill Venners**: So what do the XP guys do first, if they don't use patterns? They just write the code?

**Erich Gamma**: They write a test.

**Bill Venners**: Yes, they code up the test. And then when they implement it, they just implement the code to make the test work. Then when they look back, they refactor, and maybe implement a pattern?

**Erich Gamma**: Or when there's a new requirement. I really like flexibility that's requirement driven. That's also what we do in Eclipse. When it comes to exposing more API, we do that on demand. We expose API gradually. When clients tell us, "Oh, I had to use or duplicate all these internal classes. I really don't want to do that," when we see the need, then we say, OK, we'll make the investment of publishing this as an API, make it a commitment. So I really think about it in smaller steps, we do not want to commit to an API before its time.

**Bill Venners**: Well, there are speculative requirements. So if you say you're going to wait until there's a requirement, doesn't that simply move the problem? Because someone could say, "Well, we're going to need this." They are saying that's a requirement. But then someone else might say, "Well, do we really need this, yet?" I'm not quite sure I understand when to speculate and when not to speculate. Who decides when a requirement is really a requirement? How do I decide as a programmer?

**Erich Gamma**: Well, you must have a customer or consumer of what you produce. For Eclipse, our customer is the community who writes plug-ins. And so we interact with them, and also kind of sync with them, based on the concrete uses we know ourselves, what they might want to use. Therefore my recommendation to a programmer is that when you have to speculate, make sure that you have a least one of your clients involved, preferably more. It also helps when I already have something in my hand.

**Bill Venners**: What do you mean by having something in your hand?

**Erich Gamma**: I have a concrete example. And then I go and speculate about how to make it more abstract, and not the other way around. Ideally once I have speculated I immediately ask for feedback from my potential clients.

**Bill Venners**: So you build a concrete example first.

**Erich Gamma**: Maybe two or three, until it truly hurts. I duplicate the code once. I duplicate it twice. And then, wow, I had to duplicate it again. At this point the abstraction process starts. So, I say, it would be really nice if this class would allow me to plug in this custom behavior so that I do not have to duplicate the rest.

# The problem of frameworkitis

**Bill Venners**: What is the difference between a framework, a platform, and a toolkit, and what are the different flexibility needs?

**Erich Gamma**: With a platform I associate long term stability. It is safe to build on top of a platform. A platform makes compatibility guarantees. Frameworks often do not have this quality and I have seen many framework failures with regard to stability. If you look at Eclipse, yes it includes frameworks, toolkits, and provides platform APIs. All of this is bundled as plug-ins. Frameworks abstract and provide higher level default functionality. To do so the framework needs to be in control. This loss of control can lead to what is sometimes called frameworkitis.

**Bill Venners**: You mean the disease of wanting to make frameworks out of everything?

**Erich Gamma**: Frameworkitis is the disease that a framework wants to do too much for you or it does it in a way that you don't want but you can't change it. It's fun to get all this functionality for free, but it hurts when the free functionality gets in the way. But you are now tied into the framework. To get the desired behavior you start to fight against the framework. And at this point you often start to lose, because it's difficult to bend the framework in a direction it didn't anticipate. Toolkits do not attempt to take control for you and they therefore do not suffer from frameworkitis.

**Bill Venners**: And toolkits don't because...

**Erich Gamma**: With toolkits you create and call toolkit objects and register listeners to react to events. You're in control. Frameworks try to be in control and tell you when to do what. A toolkit gives you the building blocks but leaves it up to you to be in control.

**Bill Venners**: So a framework might be like an EJB container, because I subclass to create an EJB.

**Erich Gamma**: Right. Also, if we do frameworks, we try to make them small frameworks. We prefer many small frameworks over one heavyweight framework.

**Bill Venners**: Why?

**Erich Gamma**: Because the bigger the framework becomes, the greater the chances that it will want to do too much, the bigger the learning curves become, and the more difficult it becomes to maintain it. If you really want to take the risk of doing frameworks, you want to have small and focused frameworks that you can also probably make optional. If you really want to, you can use the framework, but you can also use the toolkit. That's a good position that avoids this frameworkitis problem, where you get really frustrated because you have to use the framework. Ideally I'd like to have a toolbox of smaller frameworks where I can pick and choose, so that I can pay the framework costs as I go. http://www.artima.com/images/ib.gif

# Next week

Come back Monday, June 6th 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/account.jsp).

# Talk back!

Have an opinion about the design patterns topics discussed in this article? Discuss this article in the Articles Forum topic, [Erich Gamma on Flexibility and Reuse](http://www.artima.com/forums/flat.jsp?forum=226&thread=112325).

# 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/](http://www.amazon.com/exec/obidos/ASIN/0201633612/billvennersA/)

[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/>

[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/](http://www.amazon.com/exec/obidos/ASIN/0321205758/billvennersA/)

# 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.