

Leading-Edge Java

## How to Use Design Patterns

A Conversation with Erich Gamma, Part I

by Bill Venners

May 23, 2005

### Summary

Among developers, design patterns are a popular way to think about design, but what is the proper way to think about design patterns? In this interview, Erich Gamma, co-author of the landmark book, *Design Patterns*, talks with Bill Venners about the right way to think about and use design patterns.

Erich Gamma leapt 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) [see 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 [see 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 [see 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 this first installment, 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.

### The real value of design patterns

**Bill Venners:** Bruce Eckel and I teach design classes, and we've found that people really want to know the Gang of Four (GoF) patterns. Patterns help sell seminars. There's a lot of marketing hype around design patterns.

**Erich Gamma:** Still, after 10 years?

**Bill Venners:** Yes. People want to know patterns, and I suspect a great deal of that is because "patterns" is still a buzzword. I'd like to cut through the hype and find out what you think people should actually do with patterns. What should their attitude be about patterns? How can people use patterns to do a better job? What is the real value?

**Erich Gamma:** I think patterns as a whole can help people learn object-oriented thinking: how you can leverage polymorphism, design for composition, delegation, balance responsibilities, and provide pluggable behavior. Patterns go beyond applying objects to some graphical shape example, with a shape class hierarchy and some polymorphic draw method. You really learn about polymorphism when you've understood the patterns. So patterns are good for learning OO and design in general.

Then on top of that, each individual pattern has a different characteristic to help you in some place where you need more flexibility or need to encapsulate an abstraction, or need to make your code less coupled. This is a really big issue in a large system. How do you preserve your layers? How do you avoid up calls or circular dependencies? The GoF patterns provide you with little tools that help you with these problems. They do so not by giving a pat solution but by explaining trade-offs. Even though patterns are abstracted from concrete uses, they also provide you valuable implementation hints. From my perspective it is the fact that patterns are implementable that makes them so valuable.

Patterns are distilled from the experiences of experts. They enable you to repeat a successful design done by someone else. By doing so you can stand on the shoulders of the experts and do not have to re-invent the wheel. However, since patterns enable many implementation variations you still have to keep the brain turned on. Finally, since patterns provide you with names for design building blocks they provide you with a vocabulary to describe and discuss a particular design.

The other question was how we should teach patterns. Not that I know exactly what you should do, but I think what you should *not* do is have a class and just enumerate the 23 patterns. This approach just doesn't bring anything. You have to feel the pain of a design which has some problem. I guess you only appreciate a pattern once you have felt this design pain.

**Bill Venners:** What pain?

**Erich Gamma:** Like realizing your design isn't flexible enough, a single change ripples through the entire system, you have to duplicate code, or the code is just getting more and complex. If you then apply a pattern in such a messy situation it can happen that the pain goes away and you feel good afterwards. It's an eye opener to realize that oh, actually this pattern, factory or strategy, is a solution to my problem. And I think that's the really interesting way to teach.

When I started teaching it was really boring, because I was just enumerating the patterns. I found it far more interesting to try to motivate with real examples how to apply patterns. In other words, you really need to present the problem in a realistic context—synthetic examples do not fly. At OOPSLA I received a *Heads First Design Patterns* book [see Resources]. It's a great book, not only because it is fun to read but also because they are able to communicate the essence of design patterns in a novel and highly visual way.

**Bill Venners:** Is the value of patterns, then, that in the real world when I feel a particular kind of pain I'll be able to reach for a known solution?

**Erich Gamma:** This is definitely the way I'd recommend that people use patterns. Do not start immediately throwing patterns into a design, but use them as you go and understand more of the problem. Because of this I really like to use patterns after the fact, refactoring to patterns. One comment I saw in a news group just after patterns started to become more popular was someone claiming that in a particular program they tried to use all 23 GoF patterns. They said they had failed, because they were only able to use 20. They hoped the client would call them again to come back again so maybe they could squeeze in the other 3.

Trying to use all the patterns is a bad thing, because you will end up with synthetic designs—speculative designs that have flexibility that no one needs. These days software is too complex. We can't afford to speculate what else it should do. We need to really focus on what it needs. That's why I like refactoring to patterns. People should learn that when they have a particular kind of problem or code smell, as people call it these days, they can go to their patterns toolbox to find a solution.

**Bill Venners:** That's funny, because my second question was that I have observed that often people feel the design with the most patterns is the best. In our design seminar, I have the participants do a design project, which they present to the others at the end of the seminar. Almost invariably, the presenters want to show off how many patterns they used in their design, even though I try to tell them the goal is a clean, easy to understand API, not to win an I-used-the-most-patterns contest. I just heard you say the same thing, that that's not the right way to think about patterns. If not, what is the proper justification for using patterns in designs?

**Erich Gamma:** A lot of the patterns are about extensibility and reusability. When you really need extensibility, then patterns provide you with a way to achieve it and this is cool. But when you don't need it, you should keep your design simple and not add unnecessary levels of indirection. One of our Eclipse mottos is that we want extensibility where it matters. Actually, if you are interested in how we use patterns in Eclipse I did an attempt to capture their uses in a chapter in the *Contributing to Eclipse* book [see Resources]. In this chapter I used design patterns to explain pieces of the Eclipse architecture.

**Bill Venners:** By extensibility, what do you mean?

**Erich Gamma:** That you can customize behavior without having to touch existing code—one of the classical OO themes. You can reuse something adapted to your particular problem.

## Core abstractions surrounded by design patterns

**Bill Venners:** In an article you wrote with Kent Beck, called "JUnit: A Cook's Tour," [see Resources] you walk the reader through the design of JUnit by, as you wrote, "starting with nothing and applying patterns, one after another, until you have the architecture of the system." I thought perhaps this approach may be inspired by Christopher Alexander, whose work on patterns in architecture inspired the software patterns movement. Do you feel that layering one pattern on top of another until you're done is an effective way to design?

**Erich Gamma:** The Cook's tour is kind of synthetic. We reconstructed the design we did on JUnit. However, we didn't develop JUnit in such a pattern driven way, instead we did it in a strict test-driven way. What is true in JUnit is that there is a core abstraction for a test, and around that core abstraction you see several other design points emerge, which in turn are materialized by pattern instances. That's something you can often observe in mature designs. There are some key abstractions you often see as a design center, and around those key abstractions you want to achieve various things. So you see patterns growing out of such a center. But I wouldn't use this as quality criteria.

**Bill Venners:** Is that what you're referring to when you say, "pattern density?"

**Erich Gamma:** Yes, exactly, patterns popping-up around some central abstraction.

**Bill Venners:** You said TestCase is the core abstraction in JUnit.

**Erich Gamma:** Actually it is the Test interface which is implemented by TestCase, but yes we started with TestCase and expanded from there.

**Bill Venners:** And then, could you define density? The number of patterns around it? You said the JUnit Cook's Tour was kind of synthetic.

**Erich Gamma:** Synthetic in the sense that the cooks tour is what is left over when you strip out all the test activity that happened during our test-driven development. Therefore it is a very compressed presentation. The density shows up in the patterns codified around Test.

We didn't just string patterns together when we designed JUnit. We did it test-driven, starting with a test that we wanted to succeed and once it passed we looked into how we could improve the code. Developing a test framework in test-driven way isn't without its challenges, but once you have the basics working it goes surprisingly smoothly. You see, Kent and I were fluent in patterns when we designed JUnit. So of course, we'd say things like, "Hey, that's composite." Composite is a pattern in JUnit. We also use template method. That's a basic one. We have command. This is of course a key one. We started with test and said, "Oh, this is a command. Oh, this is a template." Because we were fluent in patterns, our conversation was going really fast, enabling a high-velocity design.

This actually illustrates nicely how patterns provide us with design vocabulary. Similarly when you look at a UML diagram, then you see boxes and arrows, but they don't really tell you the meaning behind these relationships. But once you know the pattern then it explains what these relationships are about. If it's observer, it's really clear why there is a link between these two classes. It's because this guy observes that guy. You know exactly what's going on. And I guess that's the key point. Patterns give us a language to talk about design. Actually, the JUnit journey isn't over yet and Kent and I are currently working on JUnit 4. We are continuing to evolve JUnit in a test-driven way. One of our themes is to lower the barriers to entry. To do so we are leveraging J2SE 5 features like annotations to make JUnit easier to use.

## Pattern languages

**Bill Venners:** What is a pattern language, in the Alexandrian sense?

**Erich Gamma:** Alexander had a very ambitious goal which was to create architectures that improve the quality of life. To achieve this Alexander developed a pattern language. This is a set of patterns that build on each other. A pattern language guides a designer's application of individual patterns to the entire design. When we started design patterns we were not that ambitious. We used a more bottom-up approach based on micro-architectures.

**Bill Venners:** What do you mean by bottom up?

**Erich Gamma:** Let me step back a little and describe how I got into patterns. I think that will answer your question. I was working with Andre Weinand on ET++, a comprehensive class library and framework for C++. As I reflected on ET++, it became apparent that a mature framework contains recurring design structures that give you properties like extensibility, decoupling, and last but not least, elegance. Such structures can be considered micro-architectures that contribute to an overall system architecture. I ended up documenting a dozen or so such micro-architectures in my thesis. This approach to patterns differs from pattern languages: Rather than coming up with a set of interwoven patterns top-down, micro-architectures are more independent patterns that eventually relate to each other bottom-up. A pattern language guides you through the whole design, whereas we have these little pieces, bites of engineering knowledge. I confess that this is less ambitious, but still very important and useful.

**Bill Venners:** Is a pattern language like having a context free grammar, and from which you can make a whole bunch of programs?

**Erich Gamma:** There are some similarities at an abstract level. In the same way as a grammar can define a whole bunch of programs a pattern language can generate a bunch of solutions. The way Christopher Alexander describes it is that his patterns describe a solution so that it can be applied many times without ever being the same. But in my opinion at this level the commonality ends.

**Bill Venners:** By generate, what does he mean? If I have a context free grammar, it doesn't generate the programs. I still have to write them.

**Erich Gamma:** You still have to make the decisions, but a pattern language provides you more guidance and it has some flow. Say you want to design a room in which you are comfortable. He says, first put lights on two sides. OK, now that you have done lights on two sides, what comes next? How do you place the windows? There are other patterns that describe a solution to this problem. He basically guides you through the space. This kind of connection is what differentiates a library of patterns as we have described in the GoF book from a pattern language. What we have found is that our micro-architectures are also no islands. They are related. We sketched these relationships on the inside of the book cover, and this is what Alexander enthusiasts consider the only valuable part of our book.

**Bill Venners:** It sounds almost like a design methodology. You go this way, do this, this, and this, and you end up with a beautiful, comfortable to sit in room.

**Erich Gamma:** Yes, when you follow Alexander's patterns approach you follow the patterns in some sequence. We don't prescribe a particular order. If you have a problem, we have the solution for that, but we don't have the next step. We don't give you hints on what to do next. Alexander is way more thorough in this regard. JUnit has a bit of this pattern language approach,

because it can help you write a test case. In the JUnit documentation, Kent and I wrote a mini pattern language on how to implement a test. You start with a test, then you want to factor out common setup code, then you want to group tests together and so on. 🍷

## Resources

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/>

Erich Gamma is co-creator of JUnit, the defacto standard Java unit testing tool:

<http://www.junit.org/index.htm>

Erich Gamma leads the Java development effort for the Eclipse tool platform:

<http://www.eclipse.org/>

*Head First Design Patterns*, by Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra, is available on Amazon.com at:

<http://www.amazon.com/exec/obidos/ASIN/0596007124/>

You can download a free chapter of *Head First Design Patterns* from Artima's chapters library:

<http://www.artima.com/chapters/book.jsp?num=90281>

*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/>

*JUnit: A Cook's Tour*, an article by Erich Gamma and Kent Beck, walks you through the design of JUnit by "starting with nothing and applying patterns, one after another, until you have the architecture of the system.":

<http://junit.sourceforge.net/doc/cookstour/cookstour.htm>

## 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.

Leading-Edge Java

# 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]. 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 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. 

## 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/>

[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.

# Design Principles from Design Patterns

A Conversation with Erich Gamma, Part III

by Bill Venners

June 6, 2005

## 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 interfaces 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 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.

## The value of interfaces

**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 implementation," 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.

## When to think about interfaces

**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.

## Composition versus inheritance

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. 

## 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.

## Patterns and Practice

A Conversation with Erich Gamma, Part IV

by Bill Venners

June 21, 2005

### Summary

In this interview, Erich Gamma, co-author of the landmark book, *Design Patterns*, talks with Bill Venners about how design patterns are problem solution pairs, how design patterns help you understand intent and tradeoffs, and how to become a better designer through practice.

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 Part III. Design Principles from Design Patterns, Gamma discusses two design principles highlighted in the GoF book: program to an interface, not an implementation, and favor object composition over class inheritance.
- In this fourth installment, Gamma discusses how design patterns are problem solution pairs, how design patterns help you understand intent and tradeoffs, and how to become a better designer through practice.

## Design patterns are problem solution pairs

**Bill Venners:** My first real insight into object-oriented programming came from reading Scott Meyers' *Effective C++*. In that book he has a guideline that says, "Make sure public inheritance models 'is-a.'" That guideline helped me truly understand inheritance for the first time. However, a later *Effective C++* guideline says, "Model 'has-a' or 'is-implemented-in-terms-of' through composition." That guideline didn't help me as much to figure out what to do with composition.

In the GoF book, however, I found several more concrete prescriptions for using composition. GoF talks about using composition to model relationships like "adapts-a," "proxies-a," and "decorates-a." If you just look at the UML diagrams in GoF, a lot of them look similar. Most of them use composition with interface inheritance. Where the patterns differ is in the problem you are trying to solve with those composition and inheritance relationships. Even though the class diagram might look similar, the design intent is different. What is the role of intent in design patterns?

**Erich Gamma:** A pattern is always a problem-solution pair that can be applied in a particular context. Although the solutions might look similar in different patterns, the problems they are solving are different. In fact from ten thousand meters most patterns solve a problem by adding a level of indirection. What is interesting is how this indirection comes about and in particular why it needs to happen. Therefore if you just look at the solution to the problem, it isn't that enlightening and everything starts to look the same. When we wrote design patterns we often had this feeling—they all started to look like the Strategy pattern.

Initially we described only solutions. We had an initial catalogue of about 20 pages. Experienced developers had no problem understanding what we were up to and they rewarded us with comments like, "Yes, I've done that." However, we noticed that the rest had a hard time to get it at all. We showed different flavors of indirection. We showed a delegation. But basically we had a solution looking for a problem. Alexander's [4] view on patterns combined with the feedback from early readers helped us to not only focus on the solution but also on the problem part. A pattern has a problem and a solution, and you need to see both. For example, strategy and state have the same solution: you delegate to a separate object, and use a class hierarchy of objects conforming to an interface to vary behavior. But the problem is different. Strategy is about plugging in an algorithm, and state is about changing behavior when a class's state changes, as in a state machine.

## Understanding intent and tradeoffs

**Bill Venners:** In the GoF book, you say, "Knowing the design patterns in this book makes it easier to understand existing systems." How?

**Erich Gamma:** You can explain an existing system with patterns way more compactly than without. Patterns help you compress a dialogue about design. A good example is what Kent Beck and I did in the last section of our Eclipse book [5]. We described some of the Eclipse designs with patterns. It was really amazing how compact you can get. You can say, "that's a composite." You don't have to say much more about it. People know what it is.

What's interesting is that we did this pattern analysis on an earlier version of Eclipse, and the analysis is still correct for the latest version. So while the screenshots are out of the date the patterns we identified are still up to date.

Once you understand that the system is using a certain set of patterns, and you know that those patterns have certain limitations and liabilities, you're in a better position to understand the intent of the developer that came up with the solution. The pattern tells you about the intent but also about the tradeoffs. Once you know the limitation of a

design, you're in a better position to be a good citizen of say a reusable design like a framework. One of the worst things that can happen is that you start to fighting against a design, because you didn't really understand what it was, not honoring its intent and you start sounding a little like Frank Sinatra "I'll do it my way".

**Bill Venners:** You just said that patterns tell us about tradeoffs. What do you mean by that?

**Erich Gamma:** Design is always about tradeoffs. There are alternatives and each alternative has different consequences. When I design, I always make decisions. And each decision has advantages and disadvantages. That was also an important lesson we had learned when we wrote *Design Patterns*. Initially we were so excited about the patterns; we only saw the positive effects. It took some time until readers pointed out that this isn't realistic. So we made another pass, and also discussed the liabilities. At this point we also learned that identifying a pattern is much simpler than actually writing it. So here is an example: you have just added the Strategy pattern, and you have more flexibility. But the tradeoff is that you now have more objects and an additional level of indirection. Right? Everything has a price in engineering. That's what I mean by tradeoff. One of the key values of a pattern is that it captures these tradeoffs so that you don't have to do the analysis again. When you're walking along in a design flow, a pattern can act as a signpost. If you go this way, then you know that this is the tradeoff. I think this is highly valuable.

## Applying design patterns

**Bill Venners:** In the GoF book you and your coauthors say, "Knowing these design patterns can make you a better designer." How? Don't I still need to know when to apply them?

**Erich Gamma:** Yes, do not turn off the brain; your creativity is still required. It isn't always clear when to apply a design pattern. In addition you also need to know which variation of a pattern to apply, and how to tweak the pattern. You always need to adapt a pattern to your particular problem.

This might be a good spot for a little confession about *Design Patterns*—in the book we only tell when to apply a pattern, but we never talk about when to remove a pattern. Removing a pattern can simplify a system and a simple solution should almost always win. Coming up with simple solutions is the real challenge.

**Bill Venners:** A lot of people want to become a better designer, and they want to go to this book to help them get there, but just reading the book doesn't...

**Erich Gamma:** ...doesn't make a better designer. I agree. Learning patterns—and even more important, design—from reading books just doesn't work. And, these days there are also several other interesting pattern books, so don't stop pattern reading after the GoF book.

**Bill Venners:** What does make them a better designer? What do they need to do?

**Erich Gamma:** In addition to reading books, you need to read and understand lots of code, see how existing systems solve a particular problem and what experienced designers did. Basically what design patterns do is to tell you what these developers have done. But, just reading about it isn't enough. You become a master by mimicking the work of excellent developers. There are many interesting open source projects available that can serve as interesting reading material. Eclipse is just one of them. It also doesn't hurt to contribute to an open source project. Not only do you learn about a particular development process you will also learn how to communicate about a design in a group of developers. As a good designer you not only come up with good designs you also communicate and defend them. You have to practice, like an apprentice in a way. Over time you'll become as experienced as experienced designers.

## Practice, practice, practice

**Bill Venners:** In the GoF book, you write, "It is easiest to see a pattern as a solution, as a technique that can be adapted and reused. It is harder to see when it is *appropriate*—to characterize the problems it solves and the context in which it's the best solution." How do new designers figure out *that*. Let's say they go through the book, and they understand the patterns. Now they've got to do their job. How do they learn to know when it is appropriate to use a pattern versus when they are succumbing to the "patternitis" disease of trying to use as many patterns as possible?

**Erich Gamma:** Unfortunately, a lot of this you only learn later on in the game, once you can look back on what you did—with experienced eyes. If I were a new developer, I'd want to have a mentor or buddy. I'd program with them, and then the mentor would say, "Ping, now this is really ugly," or "oops, you have just expressed the same again that you expressed over there" . The mentor provides immediate feedback and provides pointers to dig deeper. This is just one flavor of pair programming. It's also an excellent way to improve your skills. I always learn something from a pair programming session and I wish I would do it more often.

Bottom line is that you learn patterns by programming. And, not just toy examples; real life examples. But no, you cannot learn patterns just from reading a book. With just a book you might not initially understand them fully. Once you start applying a pattern to one of your own programming problems, you start to understand it a lot better and are ready for the next learning iteration. 🎯

## 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] Christopher Alexander described architectural patterns in works such as, *A Timeless Way of Building*. Oxford University Press, 1979, available on Amazon.com at: <http://www.amazon.com/exec/obidos/ASIN/0195024028/>

[5] *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.



## Eclipse's Culture of Shipping

A Conversation with Erich Gamma, Part V

by Bill Venners

June 28, 2005

### Summary

In this interview, Erich Gamma, co-author of the landmark book, *Design Patterns*, talks with Bill Venners about the development process used by the Eclipse team, the team's "culture of shipping," and the importance of transparency in building community around a product.

Erich Gamma leapt 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 is being published in multiple installments in *Leading-Edge Java* on Artima Developer, Gamma gives insights into software design and development. Unlike the other articles in this series, this interview installment is based on a phone interview that took place on June 16th, 2005, shortly before Eclipse version 3.1 was released.

- 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 Part III. Design Principles from Design Patterns, Gamma discusses two design principles highlighted in the GoF book: program to an interface, not an implementation, and favor object composition over class inheritance.
- In Part IV: Patterns and Practice, Gamma discusses how design patterns are problem solution pairs, how design patterns help you understand intent and tradeoffs, and how to become a better designer through practice.
- In this fifth installment, Gamma describes the development process used by the Eclipse team, the team's "culture of shipping," and the importance of transparency in building a community around your product.

## A culture of shipping

**Bill Venners:** How long have you been involved with the Eclipse project, and what are the most significant lessons you feel you and the Eclipse team have learned about the development process through that experience?

**Erich Gamma:** I've been with Eclipse since the beginning. It is actually difficult to give a fixed starting date, because there were some projects that were naturally evolving into Eclipse. For example, I had already been working on the Visual Age Micro Edition integrated development environment. Some components started in this project and were continuously refined and became a part of Eclipse. It was also during this project that SWT was born. I guess it's been about six years.

When the Eclipse project began, OTI had a culture that focused on shipping on time which has always put people in the center of the development process. Since then I've been amongst others working with John Wiegand tweaking the way we develop. During the past six years we have continually adapted. We tried out things. The things that didn't work we stopped doing and things that worked we kept on doing. And, our teams made it all work.

One thing we learned was the importance of teasing out continuous deliverables. Initially we didn't have that. As a consequence we had stressful finish sprints towards the end of a release cycle. Now we have short delivery intervals. Every six weeks is a milestone. This is something we evolved to, and now we are moving to a model of always striving to be ready to ship—we always want to be in beta.

Since we are open source, we also learned the importance of the feedback loop with the community. It is vital to interact with the community and to be transparent so the community can participate. We have achieved a good transparency level, but we are still continuously striving to improve and involve the community even more. It's obvious, isn't it, that you get a better end result with more community involvement?

**Bill Venners:** It seems like there's a culture of shipping that you value.

**Erich Gamma:** That's right. In software, having cool ideas is nice, but shipping them is what counts. For us it only counts if you have shipped the thing. That's really the mindset we have. And given that you focus on shipping, we never want to be in a mode of always being two years away from shipping. You need to have a short-term deliverable. You also plan, decide and act with this mindset. You are very risk-aware—you know what you can do so you can still ship on time with quality.

## Continuous deliverables

**Bill Venners:** How do you accomplish continuous deliverables in the Eclipse project?

**Erich Gamma:** We split the release cycle into milestones at a granularity of six weeks, and each milestone ends with an improved and useable Eclipse build. In general, those six weeks are like a small development cycle, in which we plan, develop, and test. With this kind of fractal major plan, we get in effect several small development cycles for each

release. We slow down at the end of each milestone. We have a day where everybody gets out of the water and does testing. Doing testing for each milestone avoids that we accumulate a larger testing effort until the end of the release cycle. Then we document what's new and noteworthy, and we announce it to the community so they can observe our progress and provide early feedback. Then we plan the next milestone, taking into account both the overall plan and individual component plans.

So we always want to be in beta. In particular, we don't want to go dark, where people can't see what we're doing for months. The goal is to have our community tracking our progress and providing continuous feedback.

**Bill Venners:** Why is it valuable to have the community track your progress? Is transparent development something more appropriate to open source, or is it something you think closed source projects can also find beneficial?

**Erich Gamma:** I think this is independent of open or closed-source. Software creates communities and transparent development is important if you want to grow a *community*. Open source in particular, though, is not just about making source available under some license; it is really about building up a community. And you build a community by showing them what you're up to, which means you make your plans visible. All of our milestone plans and project plans are visible on the web. All of our bugs are visible. The community really sees what's going on. Of course, what we hope for in return is that the community participates. And participation can come in many different forms—for example providing feedback in bug reports, contributing newsgroup replies, providing patches, implementing additional plug-ins, or writing articles. These are the ingredients of a tight feedback loop, and this kind of feedback loop is the key to having a good, shippable product in the end. The fact that Eclipse has such an active community is really cool and a major asset. Having such a community is an asset no matter whether the environment is open-source or closed.

**Bill Venners:** You said you plan, you execute, and you test. How long do you plan at the beginning of one of the six-week mini-development cycles?

**Erich Gamma:** We usually try to have our milestone plan posted after one week. The last week in a cycle is the "chill down." Typically we do a candidate build on Tuesday, testing on Wednesday, fixing critical bugs on Thursday, and declaring the milestone build on Friday. To declare a milestone build each component team needs to say, "Yes, my component is good to go." If there are still problems with a build the component requests an additional build.

It's kind of like driving a bus, and figuring out how many bus stops you should make. If you make too many bus stops, then all that starting and stopping slows you down. That's how we came to six weeks. We need one week to get the plans done, and we need a little more than half a week to stop.

**Bill Venners:** And that gives you four weeks of execution.

**Erich Gamma:** Yes, it gives us at least four weeks of execution, which is enough time to do something significant. You have to be able to solve bigger problems in one cycle. We

started with four-week cycles, but found the starting and stopping in a four-week cycle was too much overhead.

**Bill Venners:** Do you have any decompression time built in between these iterations?

**Erich Gamma:** After we ship a major release, we have around one month of decompression time. We deliver our releases usually in June or July, before people take their vacations. We then have some planned decompression time where we look back and reflect on what worked, what didn't, and what should improve. This is also the time for exploring new stuff. We then transition into thinking about the next release.

**Bill Venners:** Is there a mini-decompression time at the beginning of each one of the mini-release cycles?

**Erich Gamma:** You need decompression time when you have really been compressed, but the idea of the milestone releases is to *not* be continuously compressed. Rather, we want a regular rhythm of sustained development. We start. We stop. At the end before we ship an actual—not milestone—release, then we have some compression. This is the end game, which is intense. We have to do testing and very careful fixing to address last-minute polish or performance items. We know we can only do this for a certain amount of time. The end game is tiring, and afterwards, you need decompression, because everyone has to do a lot of compressed development towards the end.

**Bill Venners:** What's the end game?

**Erich Gamma:** The end game is the last six weeks. We have milestone, milestone, milestone, milestone, then basically the last milestone is the end game. It is usually also six weeks. In the end game we want to have convergence. We want to stabilize and move forward carefully. These six weeks are planned in detail and the end game plan is published on the web. The end game plan defines a rhythm of test, fix, test, fix, test, fix. The whole team, supported by the community does that. All the developers and the community test for two to three days, then we fix for four or five days. We test again for two or three days. And, with each fix pass, the rules become tougher for getting changes in. In other words, the later in the end game, the harder it is to get a change in. For example, for a bug to be fixed it needs to be posted to the developer mailing list and the fixes need to be peer-reviewed.

What can also be stressful in the end game is that sometimes we have ripples that have to percolate up the higher layers. We really prefer to work on planned things, but during the end game it is difficult to plan everything.

**Bill Venners:** What kind of ripples?

**Erich Gamma:** For example, we find that a lower layer has added a new mechanism that isn't leveraged yet in all the upper layers. These kinds of ripples are difficult to plan. An upper layer might have jumped on a new function and a lower layer thinks they are already done.

**Bill Venners:** That sounds like a communication issue. The people who are doing the lower layers need to communicate better with the people doing the upper layers.

**Erich Gamma:** Absolutely, I mention it to show that even after shipping on-time for several years we're not perfect. The best solution we've found is where the lower layers help to carry forward the required changes by providing patches and help to the upper layers. The mindset we're trying to foster is that you are not done with a feature before it has rippled through all the layers.

## Agile practices in Eclipse development

**Bill Venners:** What agile practices does the Eclipse team follow?

**Erich Gamma:** You'll see our agile practices not only in what we do, but often also codified in the tool we build. We support refactoring, unit testing, fear-free development with a local history and tightly integrated versioning, automated build process support with Ant--things like that--directly in Eclipse.

As far as the agile practices we follow when developing Eclipse, we always test early, often, and automated. For each build we run over 20,000 tests. We have nightly builds that are automated. We get build reports that tell us the failures. Recently in 3.1, we added performance tests. So we not only test for correctness, but also for performance. This has helped us a lot during the 3.1 cycle and actually you will notice significant performance improvements in version 3.1.

We practice incremental design. We invest in APIs up front. But we're also improving them incrementally. We refactor, but when we refactor we keep the stability of already published API in mind. We cannot just break clients. We also eat our own dog food daily. Each component exports its latest plug-ins and runs on top of it. This is great because you are the first one to discover your own errors. And of course when you're building an IDE or tool, you're in a deluxe position to do that. Sometimes we deploy more than once a day. So if we introduced a bug in the last two hours we'll often find them even before the daily build kicks in.

**Bill Venners:** You actually deploy the components to the developers' work stations, and they are using the currently released version for the ongoing development?

**Erich Gamma:** Yes, exactly. The developers install the latest built versions of their plug-ins and then run with them.

Customer involvement is an agile practice. And mapped to open source, this means community involvement. And we have a very active and large community. The community isn't shy. It makes ambitious requests, but in the end it is highly rewarding and fun to work with such a community.

Another agile practice is continuous integration. We do nightly, weekly, and milestone builds. We also have a notion of build promotion. Nightly builds build the latest code. We use it as an early warning system to uncover integration problems across components. The next promotion level is an integration build: once a week, we want to have something that is good enough for other eclipse committers to use. The next promotion level is the milestone build, which is every six weeks. Here we want to have a build that

is not just good enough for us committers, but is good enough for the entire community. So we practice continuous integration with promotion.

We also practice short development cycles. Our development cycles are not as short as XP suggests. As I explained with the bus stop metaphor, we feel we can't go shorter than six-week cycles.

We also do incremental planning with time boxing. Given that we have the six-week cycles, this is our time box. Our top-level plan is also a dynamic plan that is incrementally updated. That is, we don't come up with a final plan up front. We have an initial plan that lists the milestones, the themes and major work items, but we adapt the plan over time. This was also a lesson learned. Originally we had static plans, which meant the plan was accurate when we started, but not later on. We opted to plan as we go, and now the final version of the plan is done the day we ship. All of our planning is done transparently. For each plan item we state whether the item is committed or proposed and we update the status regularly. Therefore the community sees what we have committed to, and what we have proposed. We usually want to hit our target, and if we can't hit it, we are willing to cancel proposed items, but we really hate to drop committed items. Looking back on 3.1 I'm glad to see that this was indeed rare. 🍷

## 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/>

## 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.