The value of interfaces

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

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

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

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

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

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

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

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

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

Bill Venners: How can I write to an implementation?

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

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

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

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

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

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

&lt;< **Page 2 of 4** >>