

## Pattern Hatching

# BRIDGE à la Java

John Vlissides and Erich Gamma

*Programmers Report*, January/February 2002

© 2002 by John Vlissides and Erich Gamma. All rights reserved.

Isn't a circle a special case of an ellipse? Your average geometry student would think so. But expressing this basic relationship in Java isn't exactly child's play.

The proverbial “simplest thing that could possibly work” would be to declare an `Ellipse` class along with a `Circle` subclass. That's dandy from a modeling standpoint, as it faithfully represents the geometric relationship. Trouble is, it's a wasteful implementation. Making `Circle` a subclass of `Ellipse` ensures that a circle will consume as much memory as an ellipse, even though it takes strictly less memory to describe a circle. You can always dissolve the inheritance relationship to save bytes, but then the code no longer expresses that fact that a circle is a kind of ellipse. You'd look like a math flunky.

Here lurks a classic conflict in object-oriented design: A class decomposition that best models a domain may well constitute a lousy implementation, and vice versa. Needless to say, we want to have our cake and eat it too.

Enter BRIDGE<sup>1</sup>, the pattern we'll be renovating today. I say “we” not in the royal sense but in the plural, because I've enlisted the services of Erich Gamma for this effort. Rather than rattling off changes section-by-section, as has been my wont, Erich and I will be considering only the choicest issues, new and old. The definitive rewrite must await a future column or—dare I mention it?—a new edition of *GoF*<sup>1</sup>.

## Raison d'être

BRIDGE has a simple goal: to offer clients a nice domain-specific abstraction, one unencumbered by the ugly realities of a fast-changing, heterogeneous computing world. The benefits can be huge. Folks who implement the abstraction needn't worry about a million application domains and their divergent needs; they focus on raw functionality. And competitors don't have to work together to get interoperability—an endearing trait to certain parties.

That's the theory, anyhow. The key to pulling it off is *delegation*. More specifically, BRIDGE splits functionality into Abstraction and Implementor hierarchies, the former delegating to the latter.

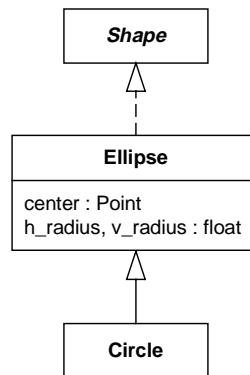


Figure 1: Conventional Ellipse–Circle implementation

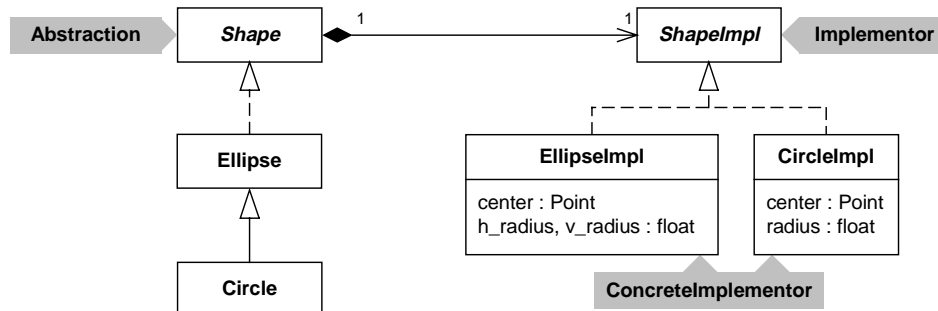


Figure 2: BRIDGE-based design

How does BRIDGE apply to the Ellipse–Circle example? Instead of doing it all in one hierarchy (Figure 1), delegation in BRIDGE lets you defer the implementation to a separate hierarchy (Figure 2). Delegation makes at least three things possible here:

1. Each implementation can be optimized independently. Circles don't have to store information meant for ellipses just because Circle inherits from Ellipse.
2. Clients aren't affected by changes to an implementation. That's true even at run-time: one ConcreteImplementor object can replace another dynamically, as conditions warrant.
3. Anyone can reuse Implementors, even classes completely divorced from the Abstraction hierarchy. For example, any class can store the representation of a circle in a CircleImpl object. No need to reinvent the wheel.

That last item may have raised an eyebrow or two. Why would anyone bother to reuse so trivial a class as CircleImpl? A very good question. Short of a very good answer, the Ellipse–Circle example leaves something to be desired.

But there's an even bigger problem with this example: it's spurious. The space savings we had hoped for just aren't there. We saved a `float` by moving the implementation out of the Ellipse class, only to add a reference to ShapeImpl, the Implementor. That little reference makes Circle at least as memory-hungry as it would have been were it derived from Ellipse! If BRIDGE has any advantage here, memory savings isn't it.

So BRIDGE doesn't always pay. Indeed, it doesn't even *work* unless you can come up with a reasonable common interface for all implementations. We'll talk more on that later; for now, assume such an interface exists. Then at least one of the following must apply before BRIDGE pulls its weight:

- You want to switch implementations dynamically.
- There's enough implementation to reuse independent of the abstraction. (BRIDGE may also promote reuse of Abstraction classes, though that's less common.)
- There's substantially more than a reference's worth of storage to save (cf. Ellipse–Circle).
- You need a profusion of classes to cover all combinations of interface and implementation.

These conditions conspire to make BRIDGE less common than it could be. Still, there's at least one high-profile example that every Java developer can appreciate, warts and all.

## A truer-to-life example

To the developer in the trenches, the most familiar instance of BRIDGE may well be AWT. It's so familiar, in fact, that AWT's limitations tend to obscure the pattern's benefits—limitations serious enough to have

provoked a successor GUI\* toolkit, Swing (now with problems of its own). Even so, AWT is functional and prevalent enough for a compelling demonstration of BRIDGE.

AWT provides the usual set of widget classes for implementing user interfaces. What makes these widgets interesting is how they work across platforms. When you create a button in AWT, you're actually creating a platform-specific button—say, a Windows button if you're on a Windows machine, or a Mac button on a Mac. All the platform-specific magic happens under the covers, leaving you blissfully ignorant about such things.

To pull that off, AWT's developers faced two challenges:

1. Thanks to Java's original "write once, run everywhere" charter, an application's platform, look-and-feel, and hence its implementation are determined at run-time—no earlier. That means the platform-specific widget implementation must be chosen dynamically.
2. While platforms have different look-and-feels, there are more similarities than differences. One platform's scroll bar, for example, is pretty much like another's. The commonalities translate into code that ought to be shared, not replicated.

The developers applied BRIDGE (knowingly, it appears) to meet these challenges.

Delegation is the key to the first one. For each class of widget, AWT introduces a corresponding *peer* class. Peers are Implementors. Each class of peer interacts with the corresponding widget class that's native to the platform. For example, Button is an Abstraction with a corresponding ButtonPeer Implementor (Figure 3). A ConcreteImplementor like MacButtonPeer implements the ButtonPeer interface in terms of real Mac button code. As with any application of BRIDGE, clients never access a peer (the Implementor) directly; they always talk to the matching AWT widget class (the Abstraction). Peers are interesting only when porting AWT to another platform, something no application programmer should ever have to do.

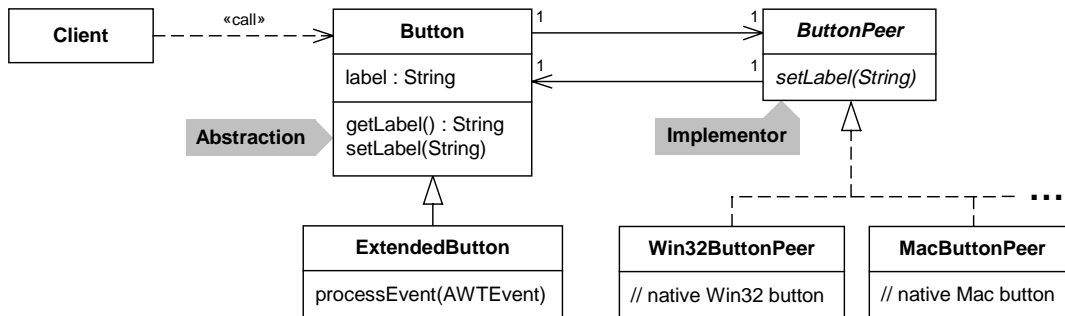


Figure 3: BRIDGE in AWT

Oddly enough, the second challenge—reusing widget code across platforms—is addressed not in the Implementor hierarchy but on the Abstraction side. For example, common code for button event-handling appears in the Button class and its subclasses, as opposed to the peer hierarchy (Figure 3 again). As a corollary, BRIDGE lets you enhance and extend Button in a platform-independent way, through subclassing.

AWT defines a basic set of platform-independent widget classes, each playing the Abstraction role in BRIDGE, and each descending from the Component class. There's a corresponding interface hierarchy rooted in ComponentPeer. Each peer Implementor is an interface, not a class, to take advantage of Java's support for multiple interface inheritance.

Consider an implementation of peers for Windows, whose widgets have commonalities unique to that platform. These commonalities are captured in a Win32ComponentPeer class. The implementation of Win32ButtonPeer can thus practice a "marriage of convenience": it inherits from Win32ComponentPeer to share code, and it implements the ButtonPeer interface to get the conformance BRIDGE requires.

\* Graphical User Interface

Peers are created lazily; a component's peer isn't created until the component is added to a container. A component uses a factory called Toolkit to instantiate the right kind of peer—an application of the ABSTRACT FACTORY pattern<sup>1</sup>. However, clients of AWT are unaware of this factory. They create widgets with standard constructor calls on Abstraction classes—e.g., `new Button("myLabel")`.

## A non-GUI example

More than once we GoF have been taken to task for appealing too much to GUI examples. We're quick to point out all the non-GUI examples in *GoF*, but the fact remains that GUI toolkits have been a mother lode of patterns. GUIs, after all, have made up a disproportionate percentage of OO code, historically speaking.

Hence we feel compelled to give a real non-GUI example, and the one in `java.net` is as good as any. That package uses BRIDGE in its `Socket` class, an Abstraction that defines an endpoint of communication between two machines (Figure 4). `Socket` delegates most of its behavior to a `SocketImpl` Implementor. A socket thus may avail itself of multiple `ConcreteImplementors`, for example, to permit navigation through firewalls.

`PlainSocketImpl` is the default `ConcreteImplementor`. `SocketImpl` is bound to `Socket` through an application of ABSTRACT FACTORY, `SocketImplFactory` playing the role of `AbstractFactory`. The `ConcreteFactory` that implements the `SocketImplFactory` can be set only once per Java run-time.

By the way, earlier versions of the JDK declared `Socket final` to restrict extension to the `SocketImpl` hierarchy. Over time this proved a faux pas, inducing contorted solutions to simple problems on the abstraction side. The powers that be recognized the error, and the `final` keyword no longer precedes `class Socket`.

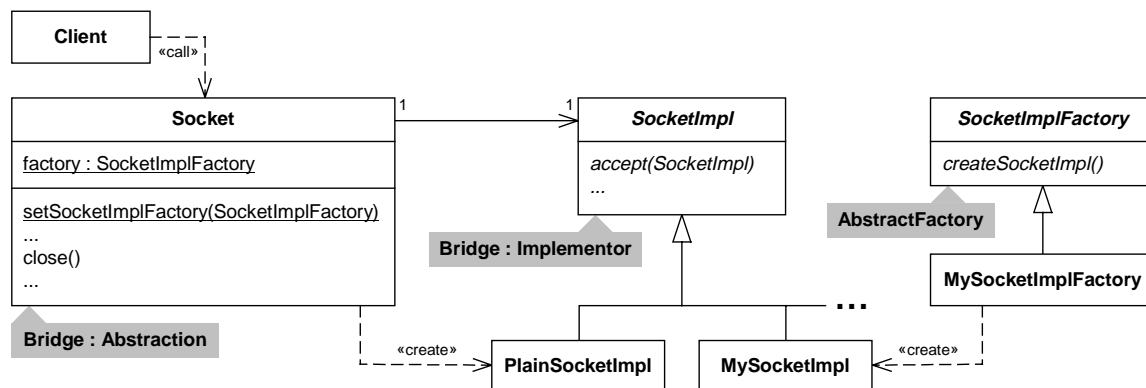


Figure 4: Socket example

## Implementation

BRIDGE is undeservedly light on Implementation items. Here are a few new ones along with new insights on an old one.

1. *Java interfaces.* Java's explicit support of interfaces, though helpful, doesn't solve the problem BRIDGE is meant to address. You can express a good domain model in the interface hierarchy, but it's still hard to get implementation reuse without delegating—especially in Java, with its lack of multiple class inheritance. Moreover, the static nature of interfaces does nothing to let you change implementations dynamically.

Then there's the instantiation problem. With BRIDGE, a client creates a button with a standard constructor call like `new Button("Label")`. Without BRIDGE, you need some sort of factory to shield clients from implementation-specific classes. Clients would have to write something like `aFactory.newButton("Label")`. BRIDGE averts this added complexity.

Interfaces are certainly useful in implementing BRIDGE, especially on the Implementor side. You can employ the marriage of convenience seen in AWT, for example, and you can use package-visible scope to hide ConcreteImplementor classes from clients.

2. *Beware final*. You can ensure that all extensions are limited to the Implementor hierarchy by declaring Abstraction `final`. But that's a drastic measure, as the developers of the Socket API discovered. It presumes there can be no reason to extend the abstraction. Unless security is paramount, it's better to document the reasons against extending the abstraction than to preclude it outright.
3. *Union or intersection of functionality?* How do you design the Implementor interface? Is it a union of the capabilities of all concrete implementors, the intersection, or something in-between?

Choosing an intersection of functionality—that is, operations that are guaranteed to make sense to all concrete implementors—often simplifies the Implementor hierarchy at the expense of more complexity in the Abstraction. That's because Abstractions have to make up any shortfall in Implementor functionality, even if all but one ConcreteImplementor implements that functionality.

What's more, clients won't be able to exploit uncommon features, forcing them to accept a lowest-common denominator of functionality. AWT illustrates this problem. Eminently useful widgets such as trees and tabbed folders aren't offered on all platforms, so AWT doesn't support them at all. That makes modern GUIs hard to build using AWT alone.

Conversely, a union of functionality shifts the complexity to the Implementor hierarchy. ConcreteImplementors must support broad functionality even if the native code base doesn't implement it.

As the disparity among implementations increases, so does the importance of striking the right balance between a union and an intersection of functionality.

4. *Abstraction–Implementor communication*. Whenever two type hierarchies cooperate, the question of how they're connected arises. Figure 2 depicts the simplest connection between the Abstraction and Implementor hierarchies: a one-way reference from Abstraction to Implementor. Many other possibilities exist. It may be that implementors need to access information in abstractions, in which case a two-way reference is needed. In AWT, for example, the widget abstraction passes itself as an argument to the peer on creation, while the peer informs the widget that an event has arrived.

Where these references are kept is yet another issue. They are usually stored as high in a class hierarchy as possible. Two-way references offer the most flexibility at the expense of added references and (possibly excessive) coupling. Sometimes you can get the effect of two-way references with only one. Assume for a moment that only Circle methods will call CircleImpl methods. Then Circle can supply a reference to itself in any CircleImpl operation that needs to access the circle. Thus CircleImpl doesn't have to store a reference to a Circle—at the cost of an extra parameter in each Implementor method. This approach won't work when an Implementor must initiate the communication, as is the case with events in AWT.

It's also possible to remove these references from the Abstraction and Implementor hierarchies entirely and put them into an associative store such as a hash table. The MEDIATOR pattern describes how to do just that<sup>1</sup>.

5. *Lazy creation of ConcreteImplementor objects*. Implementation item 2 in BRIDGE discusses “how, when, and where” to instantiate ConcreteImplementor classes. That discussion is still valid, but it skirts an important issue: lazy instantiation of ConcreteImplementors and its implications.

On the plus side are the usual benefits of lazy evaluation—you don't pay for what you don't use. In AWT, a widget buried under a little-used tab in a dialog box needn't incur all the costs of creation unless and until that tab is clicked. The main disadvantage of laziness, apart from added decision-making complexity, lies in the potential for inconsistent behavior. That same AWT widget might report different dimensions before and after its peer is created.

In general, avoid lazy creation if it risks inconsistent behavior.

## Pop quiz

Visit <http://www.bitmine.com/anythings/index.htm> and explain how Anythings illustrate BRIDGE. Include working code that replaces implementors on the fly. You have 50 minutes, starting now.

## Acknowledgments

We thank Richard Helm, Heinz Kabutz, Craig Larman, and Dirk Riehle for many helpful comments.

## References

<sup>1</sup> Gamma, E., et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.