

Pattern Hatching

GoF à la Java

John Vlissides

Java Report, March 2001

© 2001 by John Vlissides. All rights reserved.

The trouble with flaunting one's e-mail address (as below) is that it attracts spam—lots of it, in my case. Once in a while, though, I get something worth opening, something from a real live reader. Usually it concerns one of the following:

1. Kudos for, or corrections to prior columns. The former outnumber the latter, I think.
2. Thinly veiled programming problems for me to solve—gratis, of course.
3. Queries about plans for an updated *Design Patterns*.¹

That last topic is the most common, and the most easily dispatched. “Sure there'll be a new edition,” I say. The more interesting question is, *when*?

I'm afraid I can't answer that, having neither the gumption nor the authority. And if I did, I'd be lying. But beyond offering practical help for the here and now, I do hope to use this newfangled column as a driver for the revision. Pattern Hatching will retain the flavor of its namesake in *C++ Report*, elaborating and building on the patterns in *GoF*.*

The main difference, naturally, will be a focus on Java, both as an implementation vehicle and as a rich source of pattern examples. Java is even more apropos than you might realize. Legend has it that many of the patterns in the Java Platform were actually inspired by *Design Patterns*. Whether this suggests profundity or simple inbreeding is another question.

Language–pattern interplay

In any case, now might be a good time to review how programming languages affect design patterns and vice versa. Here's what we had to say about it in *GoF*:

*The choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily. If we assumed procedural languages, we might have included design patterns called "Inheritance," "Encapsulation," and "Polymorphism." Similarly, some of our patterns are supported directly by the less common object-oriented languages. CLOS has multi-methods, for example, which lessen the need for a pattern such as Visitor (page 331). In fact, there are enough differences between Smalltalk and C++ to mean that some patterns can be expressed more easily in one language than the other.*²

Penned nearly a decade ago, those remarks are as true as ever. Yet the language landscape is different now. C++ has been standardized, growing to encompass a sprawling set of libraries. Smalltalk's profile has lowered a fair bit, thanks largely to Java's rise. But Java acolytes shouldn't be too cocky. In terms of market penetration, all three languages are eclipsed by Visual Basic—the COBOL of the '90s and beyond.

* For the uninitiated, “GoF” stands for “Gang of Four,” which in this context refers either to the authors of *Design Patterns* or, when italicized, the book itself. (Not my idea.)

Another big change is the expanded role libraries play in the programming drama. Smalltalkers may scoff at the “newness” of that role, but they won’t deny its importance. Time was when you could be productive with a new language after spending a few hours learning its syntax. No longer. These days, programming takes more than a knowledge of language constructs; fluency also requires knowing how to work the libraries that come with a language.

That’s good news for pattern fans. Modern libraries are highly likely to harbor design patterns, even libraries for non-object-oriented languages. Not one but two recent books are devoted to patterns in Visual Basic, of all things.^{3,4} Both describe pattern instances in the VB platform and how to implement them yourself. Meanwhile, Jezequel, et al.⁵ have done a similar service for Eiffel, a niche OO language. (Demonstrating, by the way, that a language community doesn’t have to be big to be vibrant.) Alexandrescu’s GPI library⁶ takes this trend to its logical conclusion, simplifying pattern implementation in C++ through the magic of template metaprogramming.

Design patterns in the JDK

For their part, the Java libraries have been larded with patterns from the beginning. Erich Gamma wrote a nice article on that topic way back in 1996,⁷ not long after Java’s debut (and this magazine’s). Old-timers will recall how those were the 1.0 days, back when AWT ruled. Swing, Listeners, and inner classes were all yet to be.

Even so, that article is still illuminating on many counts. Erich shows how the original Java team—knowingly or unknowingly—made patterns “dovetail and intertwine to create a greater whole.” Focusing on AWT, he details the application of no fewer than five GoF patterns to five base classes:

1. The relationship between Component and Container reflects the COMPOSITE pattern, Component (oddly enough) playing the Component role, with Container as the Composite.
2. Container also plays the Context role in the STRATEGY pattern. LayoutManager is cast as the Strategy. LayoutManager subclasses implement different GUI layouts in a declarative way: FlowLayout for laying out components as you would text, GridLayout for row-column arrangements, and so forth.
3. Component and ComponentPeer exemplify the BRIDGE pattern, with Component as the Abstraction and ComponentPeer as the Implementor. ConcreteImplementors like Win32ButtonPeer and MotifButtonPeer encapsulate actual widgets from Win32 and Motif, respectively. This was key to AWT’s cross-platform GUI capabilities, such as they were.
4. Supporting BRIDGE in the write-once-run-everywhere objective is an application of ABSTRACT FACTORY. The Toolkit class is an AbstractFactory whose concrete subclasses bind AbstractProducts (Button, Dialog, etc.) to native toolkit implementations.
5. It didn’t make sense to allow more than one instance of Toolkit at a time, hence the application of the SINGLETON pattern to prevent it.

Erich illustrated these relationships using a notation he later dubbed “pattern-role annotations.” I’ve recreated his diagram in Figure 1, replacing the vintage OMT notation he used with UML.

It still amazes me to see this many patterns stuffed into so few classes, and to good effect. You can create a fine mess with too many patterns, but that didn’t happen here. They impart no functionality or flexibility that isn’t needed. Besides, no class participates in more than two patterns except Component, which won’t tax the intellect. To paraphrase Mr. Iacocca, if you can find a better class design, build it.

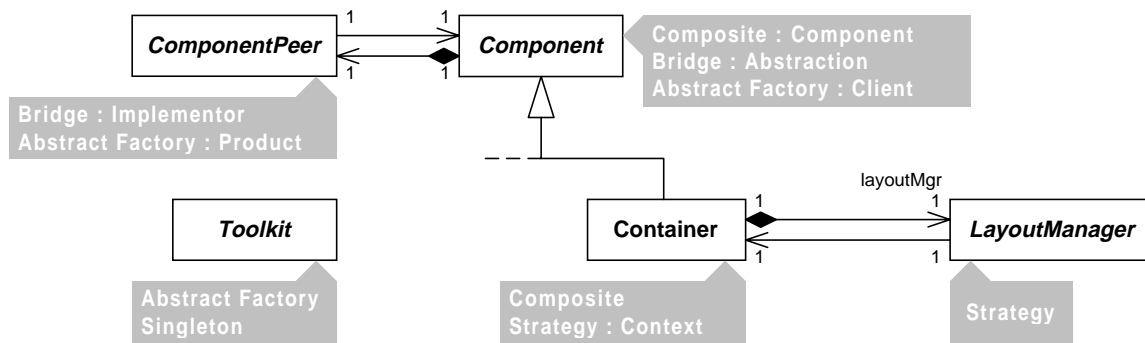


Figure 1: Patterns in AWT

Of course, Erich was barely scratching the surface in his article. There are many more pattern manifestations in the original Java libraries, and the number only increases in subsequent versions. Here are some examples:

- COMMAND: UndoableEdit, AccessibleAction
- DECORATOR: FilterReader, FilterWriter
- ITERATOR: Enumeration, Iterator
- OBSERVER: Observer/Observable, ActionListener/AWTEventMulticaster
- PROXY: Proxy
- FACTORY METHOD, TEMPLATE METHOD, PROTOTYPE: everywhere

The language's impact

As important and pattern-rich as the JDK is, I'd be remiss if I didn't discuss the Java language proper and its impact on pattern implementation.

Even though (or perhaps because) our patterns are biased toward C++ and Smalltalk implementations, we use concepts that can't be expressed directly in either language. For example, we use "interface" to describe the set of operations in a class apart from their implementation. C++ and Smalltalk have no such construct, which means you have to express interfaces idiomatically, say, through pure virtual functions or by protecting constructors. Java, on the other hand, supports this idea explicitly through the `interface` keyword. That makes it easy to tell an interface from an abstract class (which Java makes explicit as well) from a concrete class. The same goes for abstract operations. Another example is the `final` keyword, which lets you say up front that a class cannot be extended or that a method mustn't be overridden. Little things like these can make a big difference by bringing a pattern's implementation closer to its write-up.

Bigger things have an impact, too. Take garbage collection. Even if it can't guarantee that a program will use memory sensibly, few will gainsay its benefits. But there are two patterns in particular that it works wonders for.

The first is SINGLETON, which is famously silent when it comes to reclaiming the Singleton instance. Who deletes it, when, and how? After discussing the problem a bit in my *Pattern Hatching* book,⁸ I conclude that any solution short of garbage collection necessarily involves a lot of hackery, at least in C++. Score another advantage for Java (and Smalltalk, and Eiffel, and...). I also discuss problems that arise in multithreaded C++ environments—problems that Java's synchronization facilities can mitigate, if not solve entirely.⁹

The other pattern that profits greatly from garbage collection is FLYWEIGHT. Its gist is to let you enjoy the benefits of objects even when there are too many of them for your computer's own good. The pattern shows you how to use sharing to give the illusion of zillions of objects without incurring the cost. But sharing can be troublesome: Who's responsible for deleting a shared object? It's hard to know in general without extra bookkeeping, such as reference counts. Once again, garbage collection renders the issue moot.

There are other Java language facilities that ease pattern implementation. Here are a few to whet your appetite:

- *Packages let you group classes and manage their visibility, potentially improving your program's modularity and reusability.* That's useful in a wide range of patterns. In BRIDGE, for example, you can use packages to keep clients ignorant of ConcreteImplementor classes. Or you can make a Facade's subsystem explicit by putting it in its own package. In fact, you can use packages consistently to segregate interfaces, abstract classes, and concrete classes. A package of interfaces defines a framework; an accompanying package of abstract classes adds default implementations. Packages of concrete classes adhering to the framework act like components.
- *Inner classes tend to reduce the number of top-level classes.* This is a boon to patterns that are apt to create a lot of trivial classes, like ADAPTER and COMMAND.
- *Reflection.* Power tools can kill, and reflection is no exception. It can certainly kill performance. But there are great benefits to be had with reflection when used judiciously. One pattern that can benefit is VISITOR, specifically to get around the difficulty in adding new ConcreteElement classes.¹⁰ Another is PROXY, as evidenced by JDK 1.3's Dynamic Proxy API. And though I don't have a reflection example handy for DECORATOR, it's a pattern that usually benefits from any language mechanism that benefits PROXY.

Not quite perfect

Lest you think Java has it all over C++, the latter does have certain advantages, some of them relevant to patterns. Fairness compels me to mention them:

- A class may use `friend` to grant access to its private parts, so to speak. `friend` can be a useful scoping mechanism. In MEMENTO, Originators can have privileged access to Mementos by making the Originators their friends.¹¹ Java implementations have to rely on other approaches, like type laundering.¹²
- You can fake `enum` in Java with `final statics`, but it's not the same. No big deal, but once in a while it's nice to declare a closed set of tags explicitly, like maybe `male` and `female`.
- More often, you'd like to give a type name an alias, especially when it's a long and unwieldy name. You might use `mm` to abbreviate `Millimeter`, for example, or refer to `AbstractColorChooserPanel` simply as `Palette`. `typedef` lets you do both. It's particularly useful for the huge names that can result from deeply nested templates.
- Speaking of which, templates are a world unto themselves, a world I don't yet fully appreciate. Fuller appreciation must await their support in Java, since I don't program in C++ anymore. But you can sure do astounding things with them, as STL and GPI attest.
- *Default parameters.* I find myself wishing for these when I'm drowning in overloaded methods. Sometimes you can reduce their number drastically, even down to one, using default parameters—specifically, when overloading is there just to provide defaults.
- *Multiple implementation inheritance.* I know, I know. You don't need it—*normally*. Only one of our patterns uses it (ADAPTER), and even then just as one of two variant implementations. However, friends who build code generators tell me that multiple implementation inheritance would simplify both the generators and the code they generate.¹³

Stay tuned

I must say I'm thrilled and honored to be a columnist for *Java Report*. As I wind down this inaugural column, I'm reminded that no matter how familiar a topic seems at the outset, I always end up getting an education as I write about it. I hope you have the same experience with each installment of Pattern Hatching.

Acknowledgments

Many thanks to Erich for the inspiration and to Dirk Riehle for the perspiration.

References

- ¹ Gamma, E., et al. *Design Patterns*, Addison–Wesley, Reading, MA, 1995.
- ² *Design Patterns*, page 4.
- ³ Stamatakis, W. *Microsoft Visual Basic Design Patterns*, Microsoft Press, Redmond, WA, 2000.
- ⁴ Griver, Y.A., et al. *Visual Basic Developer's Guide to UML and Design Patterns*, Sybex, Alameda, CA, 2000.
- ⁵ Jezequel, J.-M., et al. *Design Patterns and Contracts*, Addison–Wesley, Reading, MA, 2000.
- ⁶ Alexandrescu, A. *Modern C++ Design*, Addison–Wesley, Reading, MA, 2001.
- ⁷ Gamma, E. “Applying design patterns in Java,” *Java Report*, November/December 1996, pp. 47–53.
- ⁸ Vlissides, J. *Pattern Hatching*, Addison–Wesley, Reading, MA, 1998, pp. 61–72.
- ⁹ Fox, J. “When is a Singleton not a Singleton?,” *JavaWorld*, <http://www.javaworld.com/javaworld/jw-01-2001/jw-0112-singleton.html>.
- ¹⁰ Blosser, J. “Reflect on the VISITOR design pattern,” *JavaWorld*, <http://www.javaworld.com/javatips/jw-javatip98.html>.
- ¹¹ *Design Patterns*, p. 287.
- ¹² *Pattern Hatching*, pp. 102–110.
- ¹³ Harrison, H., et al. “Mapping UML Designs to Java,” *OOPSLA 2000 Conf. Proc.*, October 2000, pp. 178–187.