software development
The lifecycle starts here.

Enlightened Java Style

March 2002

Is your code less than elegant? Don't shell out your hard-earned cash for a stylebook—refine your technique and make your projects more efficient by employing the XP values of simplicity, communication, feedback and courage.

By Jeff Langr

So you want to learn how to code Java properly? Well, you could go out and buy one of the several books on Java coding style that are available—or you could save the $30 and spend it on something nice, like a couple of CDs or a gift for your best friend. Even if the CDs are bad, you've still gotten something concrete and durable for your money—a debatable point if you'd bought a coding stylebook.

I happened to write such a book, *Essential Java Style* (Prentice Hall, 1999), which borrowed the pattern language from Kent Beck's *Smalltalk Best Practice Patterns* (Prentice Hall, 1996) and applied them to the Java language. Well, that's not quite accurate: Technically, I stole the patterns from Beck (he did give me permission). I foisted Smalltalk on Java—not a terrible idea, but perhaps not a great one, either. Most of the concepts fit very well; they're just a lot uglier in Java. But Java is Java, and I've learned that dealing with a language in its own terms results in more clarity than trying to solve its deficiencies with idioms from a different language.

Never mind that. Ignoring the book's other problems, the biggest one is that you should need to actually spend money on such a thing.

Coding style is always a contentious topic. Everyone has a strong opinion. Discussions about what constitutes good and bad code can quickly result in heated arguments. Personally, I always felt that I knew bad code when I saw it, and I figured my code wasn't usually bad.

Don't discount the importance of this visual sense of good or bad code. One would-be reader of my book took a quick look, and commented that it had no value because I didn't follow Sun formatting standards. This immediate dismissal reveals how important the visual appearance of code can be.

The problem with most coding standards is that they lack a consistent means of evaluating code quality, and are only vaguely held together by an unstated notion of the nature of that important glue.

XP is a wonderfully consistent development framework, built around an underlying set of four values: simplicity, communication, feedback and courage. I've taken these values to heart and realize that they mesh very closely with my own personal value system.

I contend that appropriate coding style, particularly on an XP project, can and should be based on these same four values. First, though, I'll offer some practical guiding patterns for use in coding. Then I'll return to the XP values with a few specific examples.

Patterns

Since I'm unable to remember more than, say, a half-dozen things at a time, I've settled on a very small set

of coding style "patterns" to guide my development. I've whittled down the dozens of coding patterns documented in Essential Java Style to three survivors: *Composed Method, Intention-Revealing Name* and *Comment.*

Note that my dependence on these few patterns presumes that I'm doing test-first design (TfD). Without a solid body of tests, I'm reluctant to apply any of these patterns to poorly styled code. With the tests resulting from practicing TfD, I have the confidence to refactor mercilessly, using these three patterns as my guide.

Composed Method is about communication and simplicity, or rather, simple communication. First, code must work; next, it must communicate. Code that communicates poorly costs more money. Since all hands must ultimately touch all code in an XP project, all code must be clear enough for every developer to readily understand. If code is unclear, time (which equals money) will be wasted in the extra effort required to comprehend it.

Composed Method states that a method should do things at one level of abstraction. It should be highly cohesive. It should do one thing, and do it well. Each of the following bullets offers an example of what a nice, small Composed Method should contain:

- Loop through a collection and tell `someOtherMethod` to operate on each element.
- Do something in a statement or two. If `aParticularCondition` is true, call `doSomethingSpecial`.

Rapid understanding is most important here—it should take me no more than a minute or two to understand what a developer intended the method to do.

Intention-Revealing Names are extremely important. Despite taking the utmost care to ensure that your methods are well composed, if you're careless with naming, your code will end up being indecipherable. There goes more money.

The first key to good naming is thinking in terms of encapsulation. When I name a variable or method, the name never suggests how it is implemented; rather, it indicates what it is or what it is used for. A message parameter variable named `usMsg` tells me that it is (possibly) declared using an unsigned short variable type. Too much information. Why not just message?

If you declared a collection as `v_employees` or `employeeVector` under JDK 1.x, and were interested in moving to the Java 2 Collections Framework, you'd have to rename all references to this collection to `employeeList`, or something similar. But why not just name the variable employees? Usage in the code will quickly indicate what sort of collection it is.

Method naming is even more critical. Method names tell a developer how a class should be used, and also how it is constructed. Given a list of only method names (that is, the message interface for a class), I should know exactly what each method does. When I actually look at the code implementation, I should find no surprises.

Intention-Revealing Name and Composed Method go hand in hand. The smaller a method is and the less it does, the easier it is to come up with a succinct Intention-Revealing Name.

Commenting is more of an anti-pattern than anything else. Comments indicate that code is not communicating clearly. As others have said, "Comments are lies." You can't trust comments; the only thing you can truly depend on is working code.

Strive to eliminate comments in your code. If I have to explain a method, it's not simple enough. Break it down into pieces that can be readily comprehended. Use an Intention-Revealing Name.

If I have to step through a method:

```
private void doLotsOfThings() {
// take pizza order
  ... code ...
// dispatch order to pizza builder
  ... code ...
// queue order for delivery team
  ... code ...
}
```

the code cries out for Composed Method. Often, the comments themselves suggest the new extracted method names:

```
private Order take()
private void dispatch(Order, Builder)
private void queue(Order)
```

Certainly, code can be misleading. I can code a method whose name belies what it truly does. Keeping methods small with Composed Method and meaningful method names will allow for quick recognition of such code inconsistencies. These will be ferreted out and reconciled far more rapidly than errant comments.

The bulk of comments are ultimately a waste of time—yes, that means even more wasted money. Generally, I use comments only as placeholders for incomplete work.

Visual Recognition

All three of these style principles are closely related. Ultimately, the idea of good style boils down to a notion of near-instant visual recognition. I want my methods to fit neatly in a half-height VisualAge for Java code pane. Methods should be a dozen lines or less.

Scrolling is bad. As soon as you scroll, you decrease what you knew and increase what you have to think you know. Though I knew what those first few lines did, as I scroll down, my recollection fades. Or, though I may know what the method does as a whole, line 3 is 110 characters wide. I scroll right, and can no longer see what the rest of the method is doing.

Code that forces scrolling might work fine for the fortunate few with photographic memory, but for the rest of us, it's a very taxing exercise.

When I look at code, I'm thinking at two levels. First, I must understand the basic structure or flow of what's going on, at a high level. Ideally, this is imparted by good method names. I should be able to read a method name and understand what it is responsible for doing. If I need more (perhaps I've identified a method I need to modify), I should be able to quickly read the code within the method and discern precisely how it accomplishes what the method name advertises.

There's also a certain aesthetic to well-structured code. Visually offensive code probably needs to be restructured. Michael Feathers provides the following example:

```
   void printReport() {
      printHeader();
      printBody();
      stream.println(
        "Reported $ " + getWeek() +
        " Total: " + getTotal());
   }
```

This is a violation of Composed Method—there are two levels of abstraction here. The `println` message sent to the stream object is at a lower level, and should be represented in another method.

Composed Method aside, visual awkwardness should be your first clue that something is not quite right. Two of the lines represent message sends to the same receiver (this), while the third line sends a message to a stream object. This stands out like a sore thumb. Soothe it:

```
void printReport() {
   printHeader();
   printBody();
   printTotal();
}
```

Simplicity

First and foremost, a coding standard should be simple. There is nothing worse than a long laundry list of rules that must be followed. People can only follow a few rules at a time—who among us can explain all the various fouls in the game of basketball?

You can find at least one style guide for a specific computer programming language that lists over 120 separate coding rules—all of them numbered. It frightens me to consider that this conversation could ever have taken place, or that someone might have intended it to take place:

"Joe—you didn't follow rule #85 properly!"

"Sure I did—go look at the code. You'll see that I applied rule #39, which, when used in the context described by rule #102, is perfectly valid."

Essential Java Style attempts to at least alleviate a random numbering scheme by replacing style rules with a system of low-level patterns. One of the chief benefits of patterns is that they're much easier to communicate, since each has a name that makes it easier to internalize. But ultimately, I couldn't name all the patterns in my own book if my life depended on it.

Simplicity is knowing that method size is important, that naming is key, and that comments indicate something needs improvement.

Simple Design

As espoused in XP, there are four rules of simple design:

1. All the tests run.
2. No duplication (once and only once).
3. The code expresses what you intend it to express.
4. Minimal number of methods and classes.

These rules, particularly number 3, help guide the organization and style of code to some extent. Following these simple rules will lead your application not only to a better design, but also to a more comprehensible, maintainable system.

Safety Braces

A good number of rules have been around in programming—probably since Grace Hopper first killed a moth with her "buggy" code (see, bad code can kill). One that continues to rear its ugly head is the old saw about always using braces on an `if` (or a `while`, or whatever) statement. According to this maxim, instead of using this legal construct:

```
if (conditional)
   single-statement;
```

```
you should immediately stick braces around the single statement:

if (conditional) {
```

```
    single-statement;
}
```

The reasoning: If you're using the single-statement construct, what if someone comes along and wants to add a second statement that should execute if the conditional holds true? They code the second statement immediately after the first, indenting it to line up nicely. Unfortunately, they forget to add the braces (they should probably use Python if they do this regularly):

```
if (conditional)
    single-statement;
    second-statement;
```

Uh-oh. The line of code represented by `second-statement` will execute regardless of the conditional—hence the recommendation that you always add the braces to protect you from the likelihood that someone will come along and screw things up.

Living in Fear

This is fear-based programming. It has no place in confident, courageous coding. What makes a coder courageous? Tests. If you're building code in an XP shop, you'll have produced tests for virtually all your code. If I'm not attentive and forget my braces, I'll discover my error in the next few minutes.

In terms of visual style, unnecessary braces detract from valuable vertical space. If we're following the Composed Method guideline that most methods should fit in a window 12–15 lines high, extra braces make it that much harder to constrain the method height. Besides, braces are ugly and hard to type.

Eliminating the unnecessary braces also emphasizes another XP catchphrase (catch-acronym?), YAGNI—You Ain't Gonna Need It. YAGNI suggests that you code the minimum necessary to meet the current functionality requirements. Use of the phrase "what if" is a sure sign that this guideline is about to be violated. *What if someone adds another line and forgets to add the braces?* Scratch that—you won't need the braces. Don't expend any extra effort to type two additional characters that aren't needed and only clutter the code.

Early Method Return

I vaguely remember a rule about not returning from the middle of a function:

```
char* getChar() {
    doA();
    doB();
    // ... do gobs more things
    if (condition)
        return;
    // ... do even more things
}
```

The return in the middle of the `getChar()` function is often considered a bad thing to do, mainly because it's difficult to follow the control flow within `getChar()` if returns are sprinkled throughout the method.

Composed Method helps you ensure that methods are brief. Visually, I can quickly spot an early return from a method if the entire method is a dozen lines long—no big deal. Returning early from a method can be particularly useful when done to support a guard clause.

Code should speak. The use of Composed Method, Intention-Revealing Name and Comment will support this directly. Enough said.

The effectiveness of these style suggestions will return to you in the form of feedback, both from your tests and from your peer developers.

The iterative nature of test-first design may help indicate whether style choices are a good idea or not. If you find yourself spending too much time between successful tests, consider that the existing style may be making it difficult for you to comprehend a section of code.

In an XP environment, where pairs are switching frequently, all eyes will ultimately be on the code you've written. Comprehension is key. Other developers will certainly let you know when your code does not communicate well.

Refactoring

How does this all relate to Martin Fowler's book, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999)? Fowler's book is great for the actual low-level mechanics of organizing code so it better communicates. When I picked up the book, I realized that I had already ingrained almost all mechanics discussed (Extract Method, PullUp Method and so on) into my own coding techniques. But, once again, if I were asked to name more than a half-dozen refactorings, I probably couldn't do it. Keep it simple: The basic style principles as listed above (plus the Simple Design rules) will guide your code through all the refactorings that Fowler details.

Refactoring, by the way, is the joy in my development day. Getting software to actually work is gratifying, indeed, but I love tossing around code, striving to make it as clear as possible. With testing as a backup, I can do as much of this ripping as I want, with a high level of confidence that I'm not breaking code that was already working.

Evolving a Style

Ultimately, good style isn't just a matter of your own opinion or even mine; your team has to agree on a common style. You don't have to spend all day to put this together—take, say, 15 minutes to compile a few important style notions and 45 minutes to bicker about them. The agreed-upon standards should fit on a single sheet of paper, if you feel compelled to write them down.

In an XP environment, any outstanding style issues will be quickly resolved. Dynamic pairing will force the standard to evolve. The developers will settle into a consistent style. New developers will gather the accepted style conventions by working with the existing code that clearly demonstrates them.

A Final Thought—Curly Braces

It would be nice if we could all agree on where curly braces go. Curly brace placement is a supposedly arbitrary formatting decision. But I have slowly accepted that I should change my ingrained habit of making sure the braces align top-to-bottom:

```
if (someConditional)
{
   // some code
}
```

Recently, I've begun to use the more common curly brace form:

```
if (someConditional) {
   // some code
}
```

This is a horrendously difficult habit to break, mind you. Part of it is forced through visits to customer shops, most of which now use the second form of brace placement. But, more importantly, I realize that I must practice what I preach. The first form helps me quickly visually line up from the end of a block to where it begins.

Yet this is a crutch. If I am following Composed Method and ensure that all methods do only one thing, I really shouldn't be coding blocks of more than a few lines. I should almost never have more than one indentation level in a given method.

If my methods and blocks are brief, the need for lining up braces diminishes. The braces then just get in the way; the aligned brace form takes up one more precious line of vertical space.

Due to my successes with XP, I've significantly altered my style and development outlook. I no longer concern myself with fear-based style issues. I prefer to make methods protected instead of private. I return raw—not "unmodifiable"—collections from my object. If someone wants to destroy the integrity of my class, my tests will prove that out.

Consider using the ideas in this article as a basis for a style standard for your group. The three specific examples above (safety braces, early method return and curly braces) demonstrate that you can tie every style decision to one of the underlying XP values, either directly or by virtue of the three style patterns (Composed Method, Intention-Revealing Name and Comment).

Hopefully, you'll no longer feel compelled to buy a stylebook. If that's the case, I've saved you $30—and you're welcome to mail me the $2 I might have otherwise received.