IEEE Software
www.computer.org/software

Does Beautiful Code Imply Beautiful Design?

Rebecca J. Wirfs-Brock

Vol. 24, No. 6
November/December 2007

Editor: Rebecca J. Wirfs-Brock ■ Wirfs-Brock Associates ■ rebecca@wirfs-brock.com

Does Beautiful Code Imply Beautiful Design?

Rebecca J. Wirfs-Brock

What beauty is, I know not, though it adheres to many things. —Albrecht Durer

In the short piece, "Treating Code as an Essay," Yukijiro Matsumoto, chief designer of the Ruby programming language, compares writing programs to writing essays:

> *For both essays and computer code, it's always important to look at how each one is written. Even if the idea itself is good, it will be difficult to transmit to the desired audience if it is difficult to understand. The style in which they are written is just as important as their purpose. (*Beautiful Code, *O'Reilly, 2007)*

According to Matsumoto, the most important question a code reader asks is, "What does it do?" If a program's purpose isn't clear, it isn't good, let alone beautiful. And, Matsumoto claims brevity is one of the most important contributors to beautiful code. Although brevity can contribute to code beauty—clarity of purpose, expressive use of the programming language, and design elegance also play a part. But is there more to good design than beautiful code?

Beautiful code: Brevity vs. fluency

To illustrate his point about program brevity, Matsumoto contrasted "hello world" as written in Ruby (as well as Perl and Python)—`println "Hello World"`—with the equivalent Java code (see figure 1).

The Java program is bulkier because it includes type and class declarations and syntactic elements for specifying access rights to methods and variables. But I've seen plenty of Java code that reads well.

Contrasting Java code with Ruby illustrates that strongly typed languages carry more programming constructs and require that more details be specified. Most programmers, however, quickly enough become familiar with the complexities of the languages they program in and can create reasonable code in their language of choice. Of course, when I first wander into any new programming language, my programs lack elegance because of my limited fluency in the language. Fluency demands practice and observation of other fluent programmers who can share with you their code and reasons for programming the way they do. If you're around others with more skill and practice in that language, and you read a lot of good code, you can pick up a language's nuances. This helps you develop a programming style that fits within the language's constraints and exploits its strengths. A great C++ program might not match my Smalltalk sense of aesthetics, but it still can be beautiful.

Beautiful design: Adding context

Recently, I participated in a grand experiment run by Michael Feathers and Emmanuel Gaillot to explore whether so-called experts (those attending a conference's discovery session) could agree on the merits of various code snippets. Even more ambitiously, Feathers and Gaillot wanted to determine how to structure code examples to help novices more quickly develop an intuitive feel for when a solution is appropriate.

After scanning over 100 small program-

 0740-7459/07/$25.00 © 2007 IEEE

ming samples in an hour, I found it increasingly difficult to judge whether any particular coding example had merit. Spotting quirky parts that would make me grumble if I had to maintain the code was much easier. Code that lacked expressive variable and argument names, had hardwired constants, had poor indentation, or was filled with obscure programming hacks really irritated me.

What I found most disconcerting, however, was the lack of any design context for the code I read. Without any meaningful design discussion, I felt adrift. I didn't feel confident rating a sample as good if I wasn't certain what it was doing. And any bad programming habits I spotted proved increasingly annoying. In fact, most code samples were devoid of meaningful comments that could have shed some light on the design. Comments that were present were banal or private asides.

After a while, I began wondering whether I was harsh with my judgments simply because I didn't understand the code's purpose. Or maybe I didn't like the code because I was hoping to spot some ineffable elegant coding patterns. This experiment confirmed my belief that one of the biggest myths is that well-structured code is self-documenting. If you can't understand the surrounding design context, the purpose of most code snippets isn't obvious.

Habitable code

Richard Gabriel, in *Patterns of Software* (Oxford, 1996), argues against clarity or beauty as an overarching software goal and suggests instead that we should strive for habitability: "the characteristic of source code that enables programmers, coders, bug-fixers, and people coming to the code later in its life to understand its construction and intentions and to change it comfortably and confidently."

Habitable code provides a place where, Gabriel says, "developers can feel at home, [and] place their hands on any item without having to think deeply about where it is." He believes that clarity often proves too elusive and that most programmers and writers rarely demonstrate brilliance. He says that intricately beautiful code often proves too constraining to the maintainer who has to sustain all that beauty and elegance while adding to it after its creator has moved on.

```
public class HelloWorld {

  public static void main(String args[]) {
    System.out.println("Hello World");
  }
}
```

Figure 1. "Hello World" written in Java.

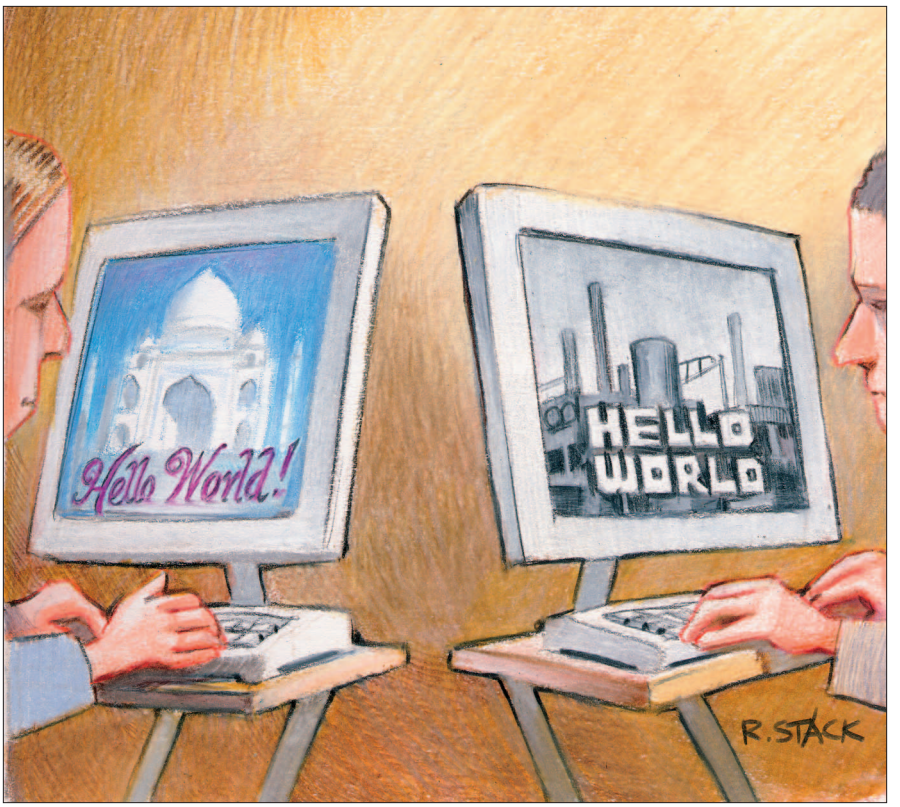
Having worked with overly zealous framework developers, I can attest to similar frustrations with what seemed to me unnecessary design embellishments. But I've seen elegant frameworks too. They usually aren't overreaching in their goals; rather, they provide the affordances needed to extend and use them without much mental effort.

In the chapter "Framework for Integrated Test: Beauty through Fragility" of *Beautiful Code*, Feathers discusses the beauty of the flexible yet concise FIT Framework. Instead of supporting multiple formats for tests, FIT supports just HTML. Each framework class is relatively simple but designed to let someone easily change it. All methods are public and thus changeable. Feathers claims that the beauty of FIT is a consequence of it being small, useful, and understandable, yet open to change.

That's an example of simple beauty, but what about inherently more complex systems?

Can complex systems be beautiful?

For the last couple of months, I've been helping a small team refactor their code. They didn't want to be the only ones who could sustain it. Their software performs exceedingly complex calculations, and, to compound their design challenge, they're constantly adding new special cases. Noth-



ing ever goes away, and the required processing complexity keeps increasing. Consequently, over the years, their code had grown incredibly dense and tangled. And they found it increasingly difficult to add new functionality. Any beauty, if it had ever been present, had become obscured. Although they couldn't simplify their processing requirements, they hoped to simplify how their code worked.

First, they cleaned up a class hierarchy that had grown somewhat arcane because of a previous designer's stylistic convention that resulted in extraneous classes. Next, they reworked the code that controlled the processing to clarify decision making and make the processing steps more explicit. Finally, they tackled an overgrown class, refactoring it into a couple of smaller service provider classes with carefully chosen, expressive class and method names. The responsibilities of each of these simpler classes were much easier to understand.

The developers were pleased with their efforts, because they felt the design intent was more evident in their refactored code. While the resulting redesign wasn't perfectly beautiful (we stopped when they deemed it good enough), the code certainly became more habitable as a result of design rework that made individual classes' responsibilities simpler, more straightforward, and consistent. And, as an added bonus, they reduced the number of lines of code. Yet their implementation was still very complex.

Can complex designs that are implemented by complex code ever be considered beautiful? Most of us recognize simple beauty when we see it. But finding beauty in complex systems seems more difficult. It takes time to appreciate the code base and understand the design. And if there are glimmers of beauty in places, that beautiful code isn't necessarily understood or appreciated by the casual reader.

It's hard to scale functionality, preserve a designer's intent, and keep a system beautiful (if indeed it ever was) when continually adding behaviors. But when the designer's intent becomes lost, it's hard to find much beauty, even if there's a brilliantly coded method or two. A glimmer of design beauty is preserved in complex systems when responsibilities are reasonably factored among design elements and the behavior of any individual class or method is comprehensible—given you know the design context.

I look to create design solutions that reflect the needs of those who will sustain the code after I move on. A good design is more than cleanly, clearly, and consistently expressed code. Beautiful code is beautiful only if it preserves and makes evident the designer's intent. What was once a good design often degrades as new functionality is added. It would be great to throw code away and rebuild it anew every few years, but that's not practical. Instead, we should strive to make our code habitable. If we do, we'll preserve the beauty and elegance that does exist in our designs a while longer. Are complex, pragmatic solutions to complex problems ever beautiful? I suppose it all depends on what aesthetics you apply in judging beauty.

Rebecca J. Wirfs-Brock is president of Wirfs-Brock Associates. Contact her at rebecca@wirfs-brock.com; www.wirfs-brock.com.