



Designing in the Future

Rebecca J. Wirfs-Brock

Vol. 26, No. 1
January/February 2009

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.



© 2008 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

For more information, please see www.ieee.org/web/publications/rights/index.html.

Designing in the Future



Rebecca J. Wirfs-Brock

The best preparation for good work tomorrow is to do good work today.—Elbert Hubbard

What can we do better to be prepared for what lies ahead? I wasn't surprised when several thoughtful folks collectively hemmed and hawed when I asked them to speculate on future design trends. As Niels Bohr observed, "Prediction is very difficult, especially if it's about the future." However, they were willing to make modest conjectures about what will continue to be important.



Several Conjectures

Dave Thomas posited that some design ideas that were important in his past shouldn't be ignored. As fads come and go, data-driven design techniques such as decision tables, rules, and state machines will continue to be important mechanisms to manage complexity. (Dave wrote about the practical use of decision tables in "Agile Programming: Design to Accommodate Change," *IEEE Software*, May/June 2005, pp. 14–16.) But we shouldn't ignore the potential of powerful functional and map-and-reduce algorithms to help us more readily solve massive data-crunching problems. Choosing the right tool for the job can make our jobs significantly easier.

Even so, Bob Martin doesn't hold out hope that emerging technologies will make our jobs that much easier:

Some folks have put a great deal of hope in technologies [that automatically generate code from models]. There is no language that can eliminate the programming step, because

the programming step is the translation from requirements to systems irrespective of language. Some folks have speculated that we'll have intelligent agents based on some kind of AI technology, and that these agents will be able to write portions of our programs for us. The problem with this is that we already have intelligent agents that write programs for us. They are called programmers. It's difficult to imagine a program that is able to communicate to a customer and write a program better than a human programmer.

Jim Coplien, in his forward to Martin's *Clean Code: A Handbook of Agile Software Craftsmanship* (Addison-Wesley, 2009), remarks that

it is crucial to continuously adopt the humble stance that the design lives in the code. And while rework in the manufacturing metaphor leads to cost, rework in design leads to value. We should view our code as the beautiful articulation of noble efforts of design—design as a process, not a static endpoint. It's in the code that the architectural metrics of coupling and cohesion play out.

Jim also notes a shift from late-1990s notions of design driven only by the tests and the code to one where responsible designers give some time to thinking and planning at a project's outset. Although in Jim's view, thinking and planning are important to ensure design quality, he also admonishes us to

pay attention to small things, but also ... be honest in small things. This means being hon-

est to the code, honest to our colleagues about the state of our code and, most of all, ... honest with ourselves about our code.

Focusing on the Details

I remember being exposed to this idea of quality in small things early in my engineering career. A wiser developer asked me to change my linker to report “No errors,” “1 error,” and “23 errors” instead of lumping everything under a generic “xx error(s)” message. At first, I was annoyed by his nit-picky suggestion even though I needed to add only one extra `If` statement. As I continued to work with more experienced, quality-minded designers, their values rubbed off on me. I learned that focusing on just these kinds of details set the tone for everything you do as a designer. In hindsight, my attitude and sense of aesthetics—not my code—needed the bigger adjustment.

Thinking at Opportune Times

Attention to details is important. Thinking and planning add value, too. How much value to place in up-front thinking is a hot topic of debate among design gurus (for example, see the podcast discussion between Bob Martin and Jim Coplien at www.infoq.com/interviews/coplien-martin-tdd).

Where you weigh in on the relative merits of up-front thinking puts you squarely in either the enthusiastic evolutionary-design camp or the think-a-bit-first camp. In some circles, any up-front design thinking is equated with big design up front (BDUF), which is almost always equated to wasted effort. But why must we choose between “thinking then doing” and “thinking while doing”? Those who encourage such polarized views are creating a false dichotomy. Up-front thinking is rarely wasted effort, especially when tackling complex or novel design problems.

I suspect it will require more experimentation before we come to a deeper understanding of good patterns for fitting various design rhythms and design thinking into development. But we won’t make progress if we let the gulf widen between the “thinking then doing” and “thinking while doing” camps. There’s a time and place for both. Simply because we refine our design ideas doesn’t mean we should always test, code, and refactor our way

to an acceptable solution. Sometimes we need to pause, think, and discuss a while before we start on a test-code-refactor path.

The Design Value of Well-Structured Requirements

I experienced this “wait, give me more time to think!” feeling last summer while attending Steve Freeman and Mike Hill’s tutorial *Style and Taste in Writing Fit Documents* (www.exdriven.co.uk/fitstyleandtaste/Style%20and%20Taste.pdf). *Fit*, a testing framework conceived by Ward Cunningham, allows “customers, testers, and programmers to learn what their software *should* do and what it *does* do” (<http://fit.c2.com>). *Fit* automatically compares expected values written in tabular form to results returned from running the program.

Steve and Mike had observed teams struggling to use *Fit* documents effectively. They wanted to teach us how to spot and correct problems with poorly specified *Fit* tables. So, they gave us several refactoring puzzles to solve, working in pairs. Maybe the problem was that it was the early morning hour or the morning after the conference banquet, but we all struggled to refactor *Fit* documents in the time allotted.

The tutorial exercises reinforced my experience that detailed pattern matching, hypothesizing about correlated factors, and invention of simplifying concepts and abstraction is hard work. These activities take time and the right frame of mind.

Up-front thinking is rarely wasted effort, especially when tackling complex or novel design problems.

Feeling rushed doesn’t help. Steve and Mike hadn’t poorly timed their exercises; they’d just given us a good taste of wrestling with moderately complex, poorly articulated real-world problem specifications. Now imagine how much more difficult a 20-by-20 Sudoku puzzle is than the traditional 9-by-9 grid. Although their problems were realistic, I’ve seen much more complex requirements.

As I was solving their *Fit* refactoring problems, I couldn’t avoid thinking about how I might design the code to work. If I simply knew the meaningful factors, I might structure a set of extensible decision tables. But because I wasn’t handed a clear problem statement, I didn’t feel overly confident in that design approach.

The consequences of poorly structured requirements obviously have enormous consequences on design. Given that problems rarely are well formed, what responsibility should we designers take to bring clarity to the problem? Whether this is official design work or not, I keep backing up to clarify problems in order to bring clarity to my design. If I don’t, coming up with simple, comprehensive solutions on the fly is difficult. Messy problems don’t lead to clean design. And small refactorings don’t always collectively add up to appropriate design abstractions. I hope the future will yield better techniques for understanding and structuring problems as well as design solutions.

But is there any new, earthshaking technology that’s ready to rock the software design world? I’m not sure. But I know I can’t wait. From my vantage point, becoming a better designer means getting better at what we do now while not getting lulled into accepting the status quo. We can and should expect better software tools, technologies, and development practices. Design rhythms and rituals will change, too. To stay effective as designers, we need to continue to learn, adapt, keep an open mind, and work to perfect our craft. 🌀

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