IEEE Software
www.computer.org/software

Connecting Design with Code

Rebecca J. Wirfs-Brock

Vol. 25, No. 2
March/April 2008

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

Connecting Design with Code

Rebecca J. Wirfs-Brock

Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. —Donald Knuth

Jon Bentley wrote his thesis on divide-and-conquer algorithms and came to greatly admire C.A.R. Hoare's original quicksort algorithm. Yet for years, Bentley "tiptoed around its innermost loop" because he didn't understand it (*Beautiful Code*, O'Reilly, 2007). It was only after he implemented his own quicksort based on an elegant partitioning scheme for *Programming Pearls* (Addison-Wesley, 1999) that he truly understood the reason for that inner loop. He also trimmed the original bulkier algorithm to a mere dozen tight lines of code.

Code clutter and unnecessary complexity can obscure a design. However, connecting design decisions to code won't happen unless developers embrace the practice of writing code as if expressing design intent matters.

Start with style

Kent Beck in *Implementation Patterns* (Addison-Wesley, 2007) presents a set of principles for creating well-structured, legible code. While legibility isn't sufficient to make design intent clear, it certainly helps me make connections between what some code is supposed to do and why it's written a particular way. I appreciate the design more readily when code is structured so changes to any code have local consequences, unnecessary duplication is eliminated, and similar operations follow recognized coding conventions.

A fundamental principle Beck applies when programming is symmetry. According to Beck, "Symmetry in code is where the same idea is expressed the same way everywhere it appears in the code." I prefer to think of this simply as being consistent, maintaining particular standards with minimal variation. Symmetry leads us to expect a `remove()` to accompany an `add()` operation. Consistency leads us to assign similar names to classes and methods with analogous responsibilities and to structure the code in a similar manner. Code that varies from established patterns and practices will then deservedly grab my attention. Variations should appear for a reason; intentional variation draws the reader's attention to important differences. Code that's riddled with accidental inconsistencies is that much harder to read, making it difficult to comprehend its design purpose.

Although the following method is easy to read, Beck claims we could improve its consistency (or symmetry):

```
void process() {
        input();
        count++;
        output()
}
```

Beck suggests replacing the line that increments count with a call to a method that performs this action. He further suggests that naming that method `tally()` instead of, say, `incrementCount()`, would better reflect intent and match the naming conventions of `input()` and `output()`.

 0740-7459/08/$25.00 © 2008 IEEE

Beck's revisions might express design intent slightly better than the original, but it's difficult to see how tweaking only three lines of code greatly improves clarity. I find myself arguing that the work of tallying the count is so inconsequential it doesn't warrant its own method because I imagine the code in `input()` and `output()` to be more substantial.

If there had been several lines of tallying code between calls to input and output, I'd be more inclined to factor out this related code into a separate method as Beck suggests for reasons of symmetry. That's because I've learned to divide any complex action into two parts—one that sequences the substeps of that operation and another that invokes separate helper methods that implement each substep. Separating action details from controlling execution makes my design intent more obvious. The difficulty is knowing when to chunk a complex operation into smaller parts. Some developers have a greater capacity for keeping lots of details in their head than others and don't feel compelled to employ this divide-and-conquer strategy. Bentley advises programmers who long to write beautiful code to practice paring code down to its essence, suggesting that they practice on small fragments containing at most two dozen lines.

What surprises me about Beck's simple example is how subtle this notion of symmetry is and how it can profoundly influence low-level programming choices. It's debatable whether a single choice made in the name of consistency improves code clarity, but when I read a large amount of code written by someone who has pursued consistency, I encounter affordances that connect design intent to the code. Code that has been purposefully structured to aid reading comprehension helps me "see" its design. Beck asks programmers to take as much care crafting their code as an author does in crafting prose.

Provide insight into your intent

However, consistent code isn't all that's needed to connect design ideas with their implementation. Depending on code complexity, a deep understanding and appreciation of that code can require extensive study and reflection or experimentation as well as a conversation with the code's author. For me, a critical aspect to connecting design to code is to be able to generally grasp what a section of code should do and then be able to examine how it works in greater detail.

When I read code, I welcome every clue that helps me get inside the developer's head. What was that person thinking? I look for code comments that point out important decisions and remark on specific processing details or nuances. I want just enough discussion interspersed with the code so that I can follow the designer's original train of thought. Too much chitchat or too many innocuous comments only distract or annoy me. Jef Raskin claims that good documentation and code comments are essential, advising, "Do not believe any programmer, manager, or salesperson who claims that code can be self-documenting or automatically documented" ("Comments are More Important than Code," *ACM Queue*, Sept./Oct. 2007). Raskin believes that good commentary contains background information that you can't derive from reading code—for example, why did the author choose this hashing scheme? What's the reasoning behind his or her threading strategy? What are the code's limits and why?

A senior developer recently remarked, "At some point in the [development] process, implementing the design turns into writing the code, and the overall design that I remember laboring over so intently fades into the background." It can be hard to remember your initial design ideas when you're buried in code. For him, design fades away once he gets deep into programming because his attention turns to making the code work, not implementing the design. I suspect that if he started adding constructive commentary to his code as Raskin suggests, he might feel a stronger connection. Design ideas evolve as we implement them. Leaving a trail of our design choices and clarified ideas interspersed with code can help us remember our design journey. It benefits others who work with our code, too.

Bentley remarks that programming "involves much more than typing symbols. One implements the program in code, runs it first on a few test cases, then builds thorough scaffolding, drivers, and a library of [test] cases to beat on it systematically."

Proponents of test-driven development echo Bentley's sentiments but further suggest that we can improve code quality by shortening the delay between thinking about design and implementing it. TDD involves short cycles of writing tests that "prove" the design, followed by writing just enough code to pass those tests. The key to TDD is incremental design and implementation of tests and production code. Design becomes a matter of choosing what line of code to write next and what test to write next to best express how your current implementation works. Each time you rework your code to make it more complete, you have an opportunity to revisit your current design, remove any accidental complexity, and rethink how best to express design intent.

It can be difficult to find the design in a code base that contains dense, complex code written by developers who show no interest in or talent for expressing symmetry, writing clean and consistent code, or leaving any clues about their design intent. Connecting design decisions to code won't happen unless developers make the effort to write code and commentary with readability and design intent in mind.

Leaving a trail of our design choices and clarified ideas interspersed with code can help us remember our design journey.

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