IEEE Software
www.computer.org/software

Designing Then and Now

Rebecca J. Wirfs-Brock

Vol. 25, No. 6
November/December 2008

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

Designing Then and Now

Rebecca J. Wirfs-Brock

The changing of a vague difficulty into a specific, concrete form is a very essential element in thinking.
—John Pierpont Morgan

Twenty-five years ago I was programming in 8086 assembly language for a low-cost graphics terminal. Our team followed structured-design practices. We developed module decomposition diagrams and printed them on an ozalid printer. Our software blueprints really were blue! We wrote functional specifications and defined the subroutine call structure before implementing any code. We wrote and debugged our coding sitting in a common work area. We held code reviews and shared effective coding practices.

On the graphics display subsystem, where I was team lead, we practiced collective code ownership. Anyone could implement any routine; when you finished one, you just picked up the next on the list. We held a meeting the week before any milestone to discuss tactics. After each milestone, we met to discuss how to make our next iteration successful. Our product-marketing manager maintained a list of features. Anything above the cut line was essential; anything below, a nice-to-have. There's a striking similarity between the rhythms of our work then and today's Scrum practices. Incremental, steady delivery seems to require close teamwork, frequent planning, and attention to detail and design quality.

Changes in Technologies

Since my assembly programming days, technology changes have been dramatic. When I first started programming in Smalltalk in the mid-1980s, I experienced the sheer joy of writing code where I asked objects to do something without having to know how they specifically performed their tasks. Finally, I didn't have to keep a myriad of little details in my head while programming. The key for me grasping the power of this new way of designing was the realization that objects' responsibilities (but not their implementations) interacted. Extremely productive Smalltalk designers didn't decompose problems so much as craft networks of responsible, interacting objects.

In Smalltalk, we wrote and immediately executed code, getting nearly instant feedback. The development environment had a wealth of reusable classes and a browser to locate and organize them. The standard fare of today's development environments was a big deal back then. In Smalltalk, the entire code base was accessible. That openness enabled us to learn by reading and emulating working examples. New Smalltalk developers spent a lot of time browsing code.

The distinction between application code and stable class libraries wasn't clearly demarcated. If you didn't like the implementation of any Smalltalk class, you could change it. This was great for rapid programming but sometimes led to application-specific behaviors being added to the most bizarre places. I developed a design sense by seeing both good and bad code.

Since those open Smalltalk days, however, I've grumbled at poorly designed code that was unchangeable and ugly. Stable class libraries allow developers to use them with confidence, but

0740-7459/08/$25.00 © 2008 IEEE

stability comes at a price. Widely distributed class libraries are difficult to improve without breaking working code. Even if we design to an interface, not an implementation, we're still confounded when interfaces change. That's why today many developers lag behind in adopting the latest versions and migrate to a new release only after it has proven stable.

Although object technology has had a big impact on software development, it isn't the only tool in a well-rounded toolkit. Grady Booch's thoughts on design choice ("Why We Model," *Object Magazine*, Nov. 1996) bear repeating:

> *In software, what models we choose to make greatly affects our world view. If I build a system through the eyes of a database developer, I'll end up with an entity-oriented schema that pushes behaviors into triggers and stored procedures. If I build a system through the eyes of a structured analyst, then I'll end up with a system that's algorithmic-centric, with data flowing from process to process. If I build a system through the eyes of an object-oriented developer, then I'll end up with a system whose architecture is centered around a sea of classes and the patterns of interaction that animate those classes. Any one of these might be right for a given application and a given development culture.*

Our technology choices today are wide and varied. Most complex systems involve a mix of technologies. We have a wealth of competing architectural platforms. Experienced designers must make trade-offs and select from a variety of appropriate technologies for solving the task at hand.

Larry Constantine, coinventor of Structured Design, observes that

> *change (and fads) are going to continue. And although Structured Design might not be on the forefront of many people's design practices these days, the principles and practices that are "best practices" in object design these days exemplify the values of Structured Design—avoiding unnecessary coupling and forming highly cohesive objects.*

Moving to newer technologies means not that we ignore our past but that we adapt and embrace practices that enable our software to flex, grow, be expressive, and meet users' needs.

Bob Martin reflects that, although tools and technology have rapidly changed, programming remains largely the same:

> *But in the face of all this massive change, this rampant growth, this almost unlimited wealth of resources, there is something that hasn't changed much at all: code. Today's modern programming languages may be rich with features and power, but they are not orders of magnitude better than their ancestors. We still write programs made out of calculations, "if" statements, and "for" loops. We still assign values into variables and pass arguments into functions. Programmers from twenty-five years ago might be surprised that we use lowercase letters in our programs, but little else would startle them about the code we write. We are like carpenters who started out using hammers and saws, and have progressed to using air hammers and power saws. These power tools help a lot; but in the end we are still cutting wood and nailing it together. And we probably will be for the next twenty-five years.*

Changes in the Thinking-Designing-Coding Cycle

Twenty-five years ago, I spent a fair amount of time thinking about and sketching solutions before I coded them. This was essential when programming in assembly language or when I could squeeze only one or two programming cycles into each day owing to lengthy compile-link-execute times. That lag time—and the disconnect between design ideas and their program representation—magnified the need to desk-check my work.

Today's power tools enable us to cut code and test our design ideas much more quickly. This is a significant improvement. Yet the more code we create, the more opportunity we have for it to grow unwieldy, inconsistent, and unmaintainable. Unless we want our code base to grow into a Big Ball of Mud (Brian Foote and Joseph Yoder, "Big Ball of Mud," *Proc. 4th Conf. Pattern Languages of Programs*, 1997), it requires care and attention. With frequent design-test-code cycles, we can rapidly validate design ideas. These cycles also reduce chances to create large tangles of untested, untestable code. That's one reason proponents of test-driven development (TDD) are gaining traction. TDD advocates propose it as a design method, not just a testing practice.

But whether you adopt TDD—writing tests before you implement code—as a design practice, attention to code quality and design maintenance is ever important. Ward Cunningham coined the term "technical debt" to explain how we should manage this effort:

Today's power tools enable us to cut code and test our design ideas much more quickly. This is a significant improvement.

I have always promoted writing excellent code, not cruft [unpleasantly built-up code]. I am happy to see code that solves today's problems, not imaginary problems that might appear in the future. As time advances, once excellent code shows its limitations and requires attention. I argue that the required work can be scheduled as part of a management policy, much like funding a company includes managing financial debt. In fact, well-managed debt serves as an accelerator. Part of that management includes paying back principal. A design that gracefully accepts refactoring will prove more valuable than big design up front.

We can continue to grow and evolve a large code base only when we pay attention to technical debt. The challenge is choosing an opportune time to pay off that debt. We need to know enough about our design's current limitations and challenges but not be inundated with so much debt that any rework appears daunting. Best design efforts rarely happen when we feel hurried, pinched by project schedules, or overwhelmed by the task's sheer magnitude.

Changes in Design Expression

When I programmed in assembly language, I didn't put much thought into how I expressed my design in code. I followed naming schemes and established conventions. But I remember being intrigued by Donald Knuth's literate programming. The central ideas behind literate programming are human readability and code comprehension. This approach combines explanatory documentation and source code so that they don't get out of synch.

We still live with that problem, and we don't write literate programs. I suspect that's because we aren't comfortable writing prose about our code. Instead, we hope it will be comprehended on its own (with judiciously placed code comments). Yet, the combination of code as prose and prose about the code is what adds value. Those who advocate intention-revealing names and expressive code echo these values, but they don't go far enough.

Code alone isn't abstract enough to convey design ideas to others because it's too easy to get lost in its nonessential details. So, we've always informally drawn and told stories about our software, in spite of any modeling tool or expressive programming language.

Although the Unified Modeling Language (UML) consolidated many different design concepts, most designers I rub shoulders with aren't fluent in it. They know rudimentary class and sequence diagrams but don't see the merit in learning any of its sophisticated nuances. They typically use UML to describe their software after they have constructed it, not to formulate its design. Even with advances in modeling tools, programming languages, and design practices, we're still exploring how best to express our designs. The "best" approach depends on design context as well as our collaborators' interests, inclinations, and skills.

Over the past 25 years, we've made great advances in tooling, technologies, and techniques that make software design more concrete. But design still requires careful thought. We still must exercise judgment. There isn't one simple way to think about and describe software. Informal design tools and techniques complement more formal ones. New techniques and design approaches will come along. And responsible designers will keep learning and improving their craft.

Acknowledgments

The quotes from Larry Constantine, Bob Martin, and Ward Cunningham are from personal conversations and emails. I thank them for providing these important insights.

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