IEEE Software
www.computer.org/software

Enabling Change

Rebecca J. Wirfs-Brock

Vol. 25, No. 5
Sept./Oct. 2008

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

Enabling Change

Rebecca J. Wirfs-Brock

A designer is an emerging synthesis of artist, inventor, mechanic, objective economist and evolutionary strategist. —R. Buckminster Fuller

Handling a requirements change or implementing new functionality can be an opportunity to leverage increased understanding to improve your design. But, more often than not, designs don't improve, so aging software systems become increasingly difficult to change. What can we do so that the quality of a significant code base won't degrade and turn into a cluttered, crufty (unpleasantly built-up) relic riddled with unnecessary design complexity?

Taking responsibility

Scott Bain, in *Emergent Design: The Evolutionary Nature of Professional Software Development* (Addison-Wesley, 2008), suggests that design deterioration isn't inevitable if we

- follow the creed of "do no harm," holding ourselves to a basic standard of not consciously making our software any worse anytime we make a change,
- integrate validation of our software with users and stakeholders into our software development process, and
- code in a style that allows us and others to do no harm.

Software doesn't have to rot if we refuse to accept decay as inevitable. As designers, it isn't enough for us to deliver working code. We should be accountable for our software's continued health and well-being.

Taking personal responsibility is important. But maintaining code habitability and keeping a design clean requires effort. I'm acutely aware that when I let my design slide, I'm creating technical debt. Sometimes there's no choice—we're in a crunch, and there's not enough time to regroup and fix quick hacks. But design decisions, good or bad, hastily made or not, tend to compound and constrain further choices. Technical design debt becomes increasingly difficult to repay as more decisions pile onto a design based on less-than-optimal choices.

The whole development team should strive to employ techniques and practices that preserve our software's ability to change. But what should we do when we disagree? Others might not share my values or design goals. Many design choices are subtle and nuanced. One solution might be better, and several others will likely be reasonable. We might not know whether a design will hold up until we've implemented it and tried to make some changes (with increasing difficulty). When should I argue for one solution over another?

Refactoring

Scott Bain suggests that we consider the cost of refactoring as part of the decision to support one option over another. Assess the consequences of waiting until later to rework your design in order to bring it back to a state where it more readily accommodates known, tangible changes. If you think rework will be inevitable and extremely difficult, hold your ground.

Some refactorings involve simple, local decisions. These have little impact on other developers or other parts of the software. I consider simple refactorings as a matter of course whenever I repair or extend overly complex code. That's part of my ongoing responsibility to keep my design clean and ready to absorb change. I find www.refactoring.com a good

 0740-7459/08/$25.00 © 2008 IEEE

online tickler list of potential refactorings. Its catalog of refactoring patterns is more current than that in *Refactoring: Improving the Design of Existing Code* (M. Fowler et al., Addison-Wesley, 1999).

Simple refactorings supported by automated tools are easy. Others require a series or combination of manual changes. Inexpensive refactorings often improve design clarity and are worth considering whenever you have to revise code.

For example, Extract Method moves a cohesive chunk of code from a larger method into a new method. Performed repeatedly, this approach lets you deconstruct a lengthy code and separate implementation details into extracted methods that are called by code in the slimmed-down controlling method. In the process, you can also give these extracted methods intention-revealing names. The relatively straightforward refactoring results in cleaner code and a clearer expression of design intent.

Some refactorings involve significant recoding and changes to tests. That doesn't mean that they aren't warranted; it just means they cost more and require more justification. For example, to perform Replace Conditional with Polymorphism, you create new subclasses with methods that encapsulate chunks of variable behavior previously embedded in conditional logic. I wouldn't perform this refactoring unless I thought that logic was overly complex and I needed to make additional changes that could be embodied in these new, simpler, and more focused classes. Doing so enables me to make certain extensions without modifying existing code. It also prevents me from heading down the path of tangling even more decision-making logic throughout that class as I support more variation.

When refactoring becomes risky

Last year I spent time with two developers reworking a bulky, hard-to-maintain class. We cleaned it up by performing a number of interface-preserving transformations. One goal we had was to break up particularly long methods. Using Extract Method, we created a simpler method that invoked helper methods. We also created new helper classes. The developers felt confident in making these refactorings because these changes had minimal impact on regression tests.

They stopped short of making another change I suggested—a design consolidation, really—because it would require changing code in two class hierarchies. I wanted to push through this change because it would remove redundancies. If each hierarchy used a common strategy class, then complex logic would only need to be refactored once. But because the two class hierarchies operated on different types, this would require us to invent an abstraction shared by both hierarchies. Not hard to do, but still more rework than they wanted to take on. It also would have forced them to critically examine hundreds of lines of code to see whether coding differences in these hierarchies were meaningful or merely gratuitous. Given their code's complexity, this wasn't a trivial exercise. My suggested refactoring would have introduced more work than they or their management were comfortable with.

I couldn't help wondering what bugs lurked in the code we didn't touch. We had an opportunity to flush out several by cleaning up the design and making sure that inconsistencies were intentional instead of accidental (and adding appropriate comments). Cut-and-paste-then-modify reuse enables you to quickly wedge in functionality but can have long-term consequences. At the moment the decision was made to not make similar code consistent, the design's integrity drifted.

History is important

What held them back from making my suggested change is that they didn't know whether code differences were significant. Nothing in the code gave them a clue. And the original designer no longer worked there. Brian Marick makes the intriguing connection between technical debt and the need for history (www.exampler.com/blog/2008/06/22/technical-debt-paying-it-down). He suggests that debt-free implementations don't need history.

Although I think this position is rather extreme, it's true that during development many refactorings take place. If these refactorings are reasonable, the design often needs little explanation other than how it works and why a particular path was chosen. However, the history behind fundamental design decisions and why they were made is important to communicate. Wouldn't it be great if this commentary were readily accessible, rather than hidden in someone else's head?

When there's technical debt, history and explanations become even more important. For example, you need to explain that "there's duplication here because when we finished adding these classes, we didn't have time to go back and rework them to use a common strategy." These explanations shouldn't get lost or dismissed as rationalizations. Brian suggests, "Maybe embedding history in the code (somehow) is a way of increasing the debt load the team is capable of supporting in perpetuity."

It's important that design teams continue to work effectively and make changes to code that has some technical debt. We aren't perfect, and any complex system is bound to have a certain amount of technical debt. Implementing our design ideas gives us feedback, causing us to adjust our initial solutions. Designs change and evolve. Sometimes good solutions turn out to need tweaking—it's a constant learning process. But over time, the costs of making change increase. If there's too much debt, making changes will be difficult and costly. Tangled code is going to be difficult to change even with explanatory notes.

Enabling continued, steady change requires that we acknowledge design corrections and adjustments as a natural part of development. Evolving designs must be cleaned up regularly so that technical debt won't overwhelm the design. If we did, then absorbing change wouldn't be so difficult.

Enabling continued, steady change requires that we acknowledge design corrections and adjustments as a natural part of development.

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