IEEE Software
www.computer.org/software

Valuing Design Repair

Rebecca J. Wirfs-Brock

Vol. 25, No. 1
January/February 2008

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

Valuing Design Repair

Rebecca J. Wirfs-Brock

The time to repair the roof is when the sun is shining. —John F. Kennedy

One of my favorite activities in any of the architecture or design courses I teach is to discuss antipatterns—design ideas hatched with good intentions that prove problematic over time. The few books on antipatterns focus primarily on introducing problems and straightforward solutions, which makes them hard to distinguish from better-known books that present design or programming guidelines or refactoring advice.

However, there's a slight but significant difference between antipatterns and style guidance. A style guide typically covers good practices—what to do and what to avoid. An antipattern is somewhat more ambitious. It seeks to explain how good intentions can go awry and suggest meaningful ways to repair broken systems. The point isn't so much to say "do this" or "avoid doing that" as to suggest ways to prevent a problem or to skillfully apply a set of corrective actions.

Deconstructing antipatterns

Because of antipatterns' prevalence and impact on system integrity, I ask students to share their encounters with an antipattern of their choosing. I start by asking what, if anything, they could have done to prevent the problem and what, if anything, they did to rectify the situation. Students have been incredibly inventive, presenting their own aptly named antipattern case studies.

For instance, the Frankenstein antipattern came about when too many people contributed to a critical component's design. Because of time pressures and disagreements, the component's design grew haphazardly. As project pressures increased, all design discussions stopped, some of the initial contributors moved on to other tasks, and the implementation continued to grow in fits and starts. Functionality continuously was hacked in without any thought to design integrity. Over time, the implementation grew into a poorly understood monster.

On many projects, design integrity—even when stated as a goal—is sacrificed in the name of delivering functionality. It's all too easy to halt design discussions and plunge into frantic coding, especially when team members continually argue and can't resolve their differences.

Fortunately, this antipattern story had a happy ending. The developer assigned to ongoing support finally declared "Enough!" and asked for time to clean up the implementation before adding more functionality. After reviewing the implementation, he requested a redesign and complete rewrite. He also stopped accepting code written by marketing folks and convinced them to write pseudocode specs instead. Frankenstein was deconstructed when the person in charge of maintenance took responsibility for fixing the design.

Solutions to antipatterns aren't always easy or satisfying. Another antipattern, Rocky Road, exhibits complexities that make it especially difficult to solve. Similar to the Lava Flow antipattern—where blobs of unused code are hanging around—a Rocky Road compounds the situation by tossing into the mix poorly designed data. You not only stub your toes on unused code but also trip on complicated data with overloaded, tangled, or forgotten and un-

 0740-7459/08/$25.00 © 2008 IEEE

used encodings. The initial design intentions might have been good—keep using the same database fields to support more functionality without schema redesign. Unfortunately, failure to rework the database design means that, over time, the data fields use increasingly arcane and obtuse encodings. Programs that use the data become unnecessarily complex because complex logic is required to decipher the data before it can be used by the program and to encode the data before it can be stored.

What I particularly like about this antipattern is its name's dual meaning—a tasty ice cream flavor *and* a travel hazard. What started out as a seemingly sweet, quick fix (overloading data field definitions) turned into a development landscape that's difficult to navigate owing to an overused encoding technique.

Navigating a Rocky Road becomes especially treacherous when a business abandons data encodings. The program code must still support these encodings because no one can be certain they were really abandoned. The Rocky Roads I've encountered have proven extremely difficult to repair. My student's case study didn't have a happy ending, either. After living with the Rocky Road for several years (it was already entrenched when he starting working on the system), he moved to a new job.

The only perfect ending to a Rocky Road story I know of was when the project leader took the time and effort to banish the unused data and code. After explaining for the umpteenth time why several hundred bytes of data weren't used, he finally decided to excise the offending patch of Rocky Road to simplify ongoing support. There was no time allocated in the schedule—he just did it. Fortunately, although that patch of Rocky Road passed through several different applications, it was under his control.

Planning for repair and redesign

Usually, a Rocky Road receives attention only after severe consequences or extreme difficulty in adding new functionality. Making an informed diagnosis before making repairs usually involves several individuals assessing any programming, reporting, and data dependencies. Ferreting out the costs and impact of such revisions can be daunting. Furthermore, this task can be compounded when data becomes retargeted for other uses by other distant and poorly understood systems.

So, development teams typically make only modest, safe, and limited improvements. Rarely do they significantly redesign database schemas for the sake of simplifying software programs or overall system integrity. Usually, there must be compelling reasons to make deep and significant changes (think Y2K or a forced migration to a new technology platform). Modest, compromised repairs don't improve the system's overall habitability that much. Those who make the repairs often feel that they've merely delayed disaster for a while.

Recently, I wrote a blog entry on antipatterns, asking others to share their experiences in successfully tackling them. Instead of glowing success stories, I got a healthy dose of reality. One person stated,

> *I see a trend in business lately, especially big businesses, where projects are done in phases. Instead of sticking with a design process and trying to fix the bugs within it, executives get concerned that the problem is within the development methodology, and discount what should have been done at the beginning of the project which is adequate planning. …All the AntiPatterns in the world won't get you there if you don't leave the time for thought and proper planning on the front end.*

Prevention is preferable to repair, but upfront thinking and planning can't avoid all future problems. What appeared to be a solid design decision might in hindsight seem incredibly naïve. During an incremental development process, a system's design context rarely stays constant.

To preserve design integrity while supporting change and evolution, some software thought leaders propose that we constantly refactor our code and data. Several popular refactoring books present simple techniques for making relatively localized improvements: *Refactoring: Improving the Design of Existing Code*, by Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts (Addison-Wesley, 1999); *Refactoring to Patterns*, by Joshua Kerievsky (Addison-Wesley, 2004); and *Refactoring Databases: Evolutionary Database Design*, by Scott Ambler and Pramodkumar Sadalage (Addison-Wesley, 2006). We'd be well advised to follow their advice.

However, on many projects, developers and their management become inured to ever-growing complexity and don't stop to make timely, simple repairs. They accept increasing complexity as a natural consequence of supporting new functionality. And project pressures never let up; there isn't time in busy schedules to merely clean things up and make the system more maintainable. So developers make repairs at a later date, when problems have become widespread, invasive, and more difficult to correct.

Michael Feathers' *Working Effectively with Legacy Code* (Prentice Hall, 2004) presents techniques for cracking open and improving ugly code in systems. But refactorings, while useful, won't solve more systemic problems. Often, significantly improving an ailing system requires more than a few, safe refactorings. It warrants significant redesign and rework.

Perhaps repair, rework, and redesign should be more central to any iterative development process. As a community, we need to adopt better practices for repairing and reworking complex systems and for determining when a redesign is required.

Repairs and redesigns are often more complicated than the design and implementation of a greenfield project. Careful analysis might be required before launching into any major repair effort. Even rework often introduces a series of related changes with unpredictable outcomes. So developers must test the changes to assess whether they've had the desired effect. Things might get better, but the developers can't guarantee miracle cures. This makes for a tough sell—"trust us, with a little time, we know we can make things somewhat better."

Because we can't prevent certain antipatterns, we need to better plan for and perform necessary repairs. The optimal time to remedy an antipattern isn't when a project is in crisis but when there's a slight lull in the action, yet an urgent need to make improvements.

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