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

The Responsible Designer

Rebecca J. Wirfs-Brock

"One must always be aware, to notice even though the cost of noticing is to become responsible."
—Thylias Moss

Successful software systems often live far longer than their original designers anticipated. And over their lifetime, most of those systems evolve. Developers who make modifications, fix bugs, and add new features to long-lived systems have an easier time of it if they keep the code base habitable ("Creating Sustainable Designs," Rebecca Wirfs-Brock, *IEEE Software*, May/June 2009) and preserve design integrity. But even so, maintenance can be painful when new requirements invalidate initial design assumptions. Given that we can't anticipate the design impact of every future requirement, is it enough to keep our code clean and our designs well factored? Or should we be doing something more?

Support Design Variability

Most complex systems support a fair amount of variability from the start. Some systems support those variations poorly—they're rife with cut-copy-and-paste reuse and overly complex code. Conditional checks and hard-coded constants are common. The code is as uninspired as its designers were. As a consequence, the implementation may support a high degree of variation, but the design is accidental or chaotic. The system squeaks by with inflexible, difficult-to-change code.

A more flexible design would have fewer hard-wired assumptions, fixed values, and duplicated code. It might have been refactored and revised numerous times to accommodate shifting design concerns. Most likely, it was initially designed to be testable and it continues to be tested. Initially, abstractions may have been identified, but they too have been reworked and revised. The current design is coherent because of its revisionist designers.

Flexible design is the byproduct of preparation and continued attention to detail. Where there's a lot of variability in a design problem, a flexible solution will incorporate appropriate design hooks that allow for developers to predictably add planned extensions. Once they've established ways to support specific variations, developers can follow predefined extension recipes rather than hacking in new features that are similar to existing ones.

Extremely flexible designs, however, can be unnecessarily complex. If you don't watch out, software that's too cleverly designed ends up with a pile of complexity that's rarely used but must be understood in order to maintain the system. Extending such a design can involve making many error-prone steps. Yet even elegantly crafted flexible designs can have steep learning curves. New developers need to be taught the right way to make changes and extensions. They need to learn design idioms. Coding conventions need to be enforced.

Joshua Kerievsky, in *Refactoring to Patterns*, offers advice on how to avoid overly complex solutions: "Patterns are a cornerstone of object-oriented design, while test-first programming and merciless refactoring are cornerstones of evolutionary design. To stop over- or under-engineering, balance these practices and evolve only what you need."

OK. We all know we shouldn't create elaborate solutions when simpler, adequate solutions suffice. And we should refactor our design when we're faced with changing requirements. But how can you know whether you're evolving a design along reasonable dimensions? Is agile development the answer? Agile development processes embrace the

0740-7459/09/$26.00 © 2009 IEEE

notion of making decisions at the last *responsible* moment.

On agile projects, just-in-time requirements drive incremental development. As a consequence, design activities are short and focused on the current iteration. You design to meet current requirements and don't have time to over-design.

Indeed, agile development can be a heady experience when you're fed a steady stream of right-sized requirements. If you're so lucky, you can be extremely productive. You don't fret about tomorrow's problems; instead you solve the problems of the day. You get to focus on simple design and clean code. Frequent, timely feedback gives you insights into the next problem you tackle. You work closely with others. Nobody goes off and does cowboy coding or clandestine design.

But agile development isn't always a designer's paradise, as Joe Yoder and Brian Foote pointed out in their Agile 2009 conference debate, "Big Balls of Mud: Is This the Best that Agile Can Do?" Agile practices, if followed naively, can lead to system decay (also known as big balls of mud; see their original big balls of mud pattern at www.laputan.org/mud). Lack of any upfront design can lead to muddled design. Continuously evolving a software architecture can prove expensive. Piecemeal growth can be haphazard and lead to less than optimal designs. Late changes to requirements can cause significant design churn.

It seems obvious, but it's worth stating: no development process can eliminate disruptions caused by significant requirements changes. When your design context changes, of course you need to rethink earlier decisions. One myth that should be dispelled about agile design is that it always happens just in time at the keyboard with no upfront planning or thought about design. That may make for a sensational story, but it's only common when you've already figured out the hard bits. If your development tasks are predictable and routine, well then, you don't need to spend a lot of time thinking about your design. Your primary job is to produce working code and keep it clean. But when tackling a new and challenging design problem, even agile designers take time to explore their options—especially if there's great risk or uncertainty. It's the responsible thing to do.

Flexibly Support Core Variations

But given the shifting design context that many of us live with, whether we work on agile projects or not, what can we do to better support variability that's inherent in most software? James Coplien wrote about commonality-variability analysis in his PhD thesis and in "Commonality and Variability in Software Engineering" (*IEEE Software*, Nov./Dec. 1998).

The first three steps of commonality-variability analysis aren't strictly about design. Instead they characterize the variability that your software needs to support. These activities set you up to make more informed design decisions. Ideally, you should do this for core areas of software functionality that have significant impact. You start by asking what functions will change over time or work differently because of certain known conditions. A list of points of variation, or *hot spots*, can focus your efforts. Each hot spot becomes a separate design problem.

Coplien outlines these steps for analyzing and then solving the design for a hot spot:

1. Establish the scope of the variation—how much of the design will you consider?
2. Identify what's common and what varies.
3. Bound the degree of variability the design solution will support. Place specific limits on how much variation it can support. Explain those limits.
4. Exploit commonalities in a design solution; while
5. Accommodating the variability.

Ideally, you should do this analysis collaboratively with business decision-makers having relevant domain expertise. This is especially important if you anticipate elaborate or costly design solutions. Experienced designers often spot potential variations that business experts might not. If we software designers work collaboratively with business decision-makers, we can jointly determine whether potential points of variability are real and meaningful to the business, or just byproducts of our highly tuned abstraction skills. If they're important, then we should pay extra attention.

James Taylor, in "Using Business Rules in Stable, Core Processes" (http://jtonedm.com/2009/09/03/using-busines-rules-in-stable-core-processes) distinguishes between core business processes that are stable and predictable, from those at the business's edges where change is the norm and business processes and rules seemingly never settle down. Whether you design IT software or embedded systems, it makes sense to focus on crafting flexible solutions for the known, stable parts. The other parts may be important, but their design constraints will keep changing and evolving. It may be good enough to keep the code habitable.

Fortunately, you can apply everything you've learned about good design to creating a flexible design solution. However, it requires extra effort. You may need to:

- refactor, isolate, encapsulate, and redistribute responsibilities;
- design places where behavior can be "tuned" or replaced;
- use factoring strategies and known patterns;
- design configurable behavior and data;
- apply inheritance when variations are relatively few and static, or choose composition when variations are many or need to change at runtime;
- define interfaces that can be implemented by different classes; and
- enable dynamic configuration of system behavior.

When you propose a flexible design solution, it's reasonable to state your design's limitations. Flexibility always has limits. A well-designed, flexible solution encapsulates known variable aspects and provides mechanisms that adequately support expected changes and adaptations.

Identifying "just enough" flexibility is a key design skill. And at the end of the day, a responsible designer should strive to make informed decisions about what's an appropriate amount of design flexibility.

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