Toward Design Simplicity

Rebecca J. Wirfs-Brock

Vol. 24, No. 2
March/April 2007

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

Toward Design Simplicity

Rebecca J. Wirfs-Brock

Our responsibility is to do what we can, learn what we can, improve the solutions, and pass them on. —Richard P. Feynman

When designing I tend to go with the flow. I don't think deeply about why I'm making any particular decision. I'm an intuitive designer. I do have a set of principles that drive my decision making. But I don't ponder what principle to apply at each moment or worry about violating any specific principle when I do make a decision. My design process is fluid, dynamic, and somewhat messy. I'm comfortable with tossing out partial solutions when better ideas come along. Rework and revision seem natural and necessary.

A sense of design aesthetics drives my decision making as much as anything. Clean designs are better than messy ones. Alan Perlis observes that "Simplicity does not precede complexity, but follows it" ("Epigrams on Programming," Sept. 1982 *ACM SIGPLAN Notices*). Get it working, then get it working better seems to track with my experience. But not every comprehensive solution can be simplified.

You can't force simplicity

I remember a colleague who struggled with complexity. He was redesigning a loan approval application that, given certain parameters, would answer whether a personal loan would be approved and for what amount. At the problem's core were rules that lent themselves to carefully designed decision tables. At first blush, his redesign seemed simple and elegant. But as he continued implementing all of the previous system's functionality, his elegant design was thwarted by prickly exception cases that could only be handled by many extra boundary checks.

He was dismayed that his final solution didn't exploit object technology or seem particularly elegant—even though his implementation avoided hard coding values and allowed for extensions and new rules in a fairly elegant manner (by inventing a small domain language to describe constraints). He considered his complex solution to be somewhat of a failure. I disagreed. To me, it seemed that his solution's structure reflected the business constraints' inherent, messy complexity.

If the nature of your problem lends itself to a simple solution and you're willing to put the time and effort into redesigning and streamlining your current solution, your next redesign might be simpler. But simplicity can't be forced.

Robert Martin, in *Agile Software Development: Principles, Patterns, and Practices* (Prentice Hall, 2002) writes,

> *Every software module has three functions. First, there is the function it performs while executing. This function is the reason for the module's existence. The second function of a module is to afford change. Almost all modules will change in the course of their lives, and it is the responsibility of the developers to make sure that such changes are as simple as possible to make. A module that is hard to change is broken and needs fixing, even though it works. The third function of a module is to communicate to readers. Developers unfamiliar with the module should be able to read and understand it without undue mental gymnastics. A module that does not communicate is broken and needs to be fixed.*

Sure, code that reads likes a page turner is certainly better than code that's as dense as a Thomas Pynchon novel. But is a hard-to-change, hard-to-understand module a product of bad design? Or does it merely reflect the problem's inherent complexity? Few design experts talk about how to live with inherent complexity. But software developers do this every day. Designers don't start out intending to create unnecessarily complex software. But many live in systems riddled with complex interfaces and tortured logic. As a developer, you have to decide whether to tackle unnecessary, accidental complexity or leave it alone, adopting the attitude that if it works it isn't broken.

Good intentions gone bad

Anti-patterns name and characterize problematic software that might have started out with good intentions but ended up being overly complex or unfit for use. Over the past few years, I've informally polled colleagues and designers about which anti-patterns they've seen in systems they've worked on and which they find the most annoying (for an extensive list of general development anti-patterns, see http://en.wikipedia.org/wiki/Anti-pattern#General_design_anti-patterns). By far and away the most irritating and commonly mentioned anti-pattern is the *Lava Flow*.

A Lava Flow is characterized by "flows" of previous design fragments strewn about the working system, hardened into immovable, generally useless bits of implementation (perhaps commented out, perhaps not). No one can remember much, if anything, about the flow, but it typically looks so complicated that it seems important, even if no one can really explain what it does or why it exists. Everyone shies away from Lava Flows. When newcomers stumble into a Lava Flow fragment, they often stub a proverbial toe and need to ask a wise team elder to explain (or rationalize) the Lava Flow and help them go around it.

Requirements shift. New code is added but old code is left in place, just in case. I once talked to a software architect whose team purposely left unused code hanging around in a system heavily influenced by state and federal regulations. These regulations changed from year to year and then frequently changed back again. The team could have tucked unused code into a source code management system, but they chose not to. They chose to live with unnecessary, familiar complexity just to keep it in mind.

Creating and decreasing complexity

Accidental complexity has many causes. Most boil down to not cleaning up your design once the software is working. Cleanup time rarely gets officially scheduled. Managers who control schedules and resources typically don't perceive a business justification for such cleanup. Most developers I know always have a long list of things they'd like to tidy up. But most don't get sanctioned time to putter with a working system to improve it. They end up reworking existing code (if they do) as an "off the clock" activity.

Complexity can accrue in more innocuous ways. The original designer leaves, and someone else makes a change. Code is revisited after too long a stretch away from it. Not fully understanding (or remembering) the original design intent, programmers implement new functionality or fix a bug in an overly complex way.

Software that continues to evolve and add new functionality can get quite complex. And until it gets overbearingly so, developers tend to tolerate increased complexity (whether accidental or necessary). You can make complex solutions more manageable by spot-reducing the complexity of any single design element: Simplify overly long methods. Break them into smaller, more understandable ones. Refactor complex behaviors into new classes. Redesign interfaces to make them simpler.

One principle that designers can apply is the Single-Responsibility Principle. Simply stated by Robert Martin, "A class should have only one reason to change." When a class must support a new variation, don't just wedge it in. Instead, bust up the class into two parts: one that supports the varying behavior and one that invokes it.

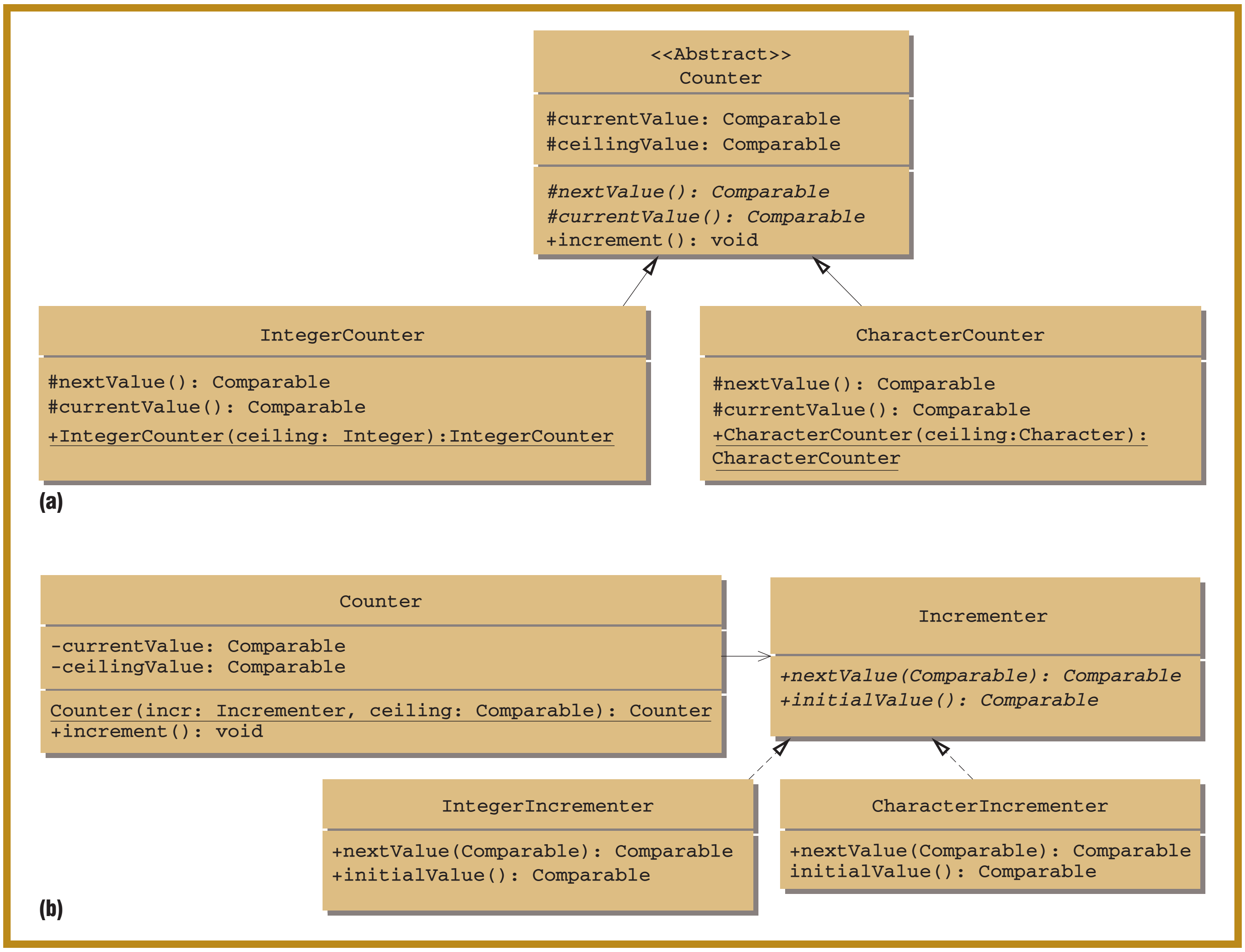
Inheritance vs. composition

Dividing a class involves a trade-off. Complexity shifts. I'm not so sure that composition is inherently simple (the more parts you have to compose, the more you have to understand about how to properly make them work together). Rather than looking in one place, you must look at several.

Is refactoring into two different classes an overall design simplification? In my design class, I illustrate two different solutions to the design of an integer and a character rollover counter. One solution uses inheritance (see figure 1a), the other composition (see figure 1b). I ask students which they prefer. Those who prefer inheritance like it because the behavior seems "in the same place." Those that prefer composition aren't bothered by having to plug in a very simple class to increment the counter. The incrementer is so tiny it seems hard to break (and easy to get right).

Richard Gabriel, in *Patterns of Software* (Oxford Univ. Press, 1996), talks of inheritance as a form of compression. Compression happens when subclasses base much of their meaning on superclass definitions. Compression can be useful. You can understand something unfamiliar just by knowing its surrounding context. As Richard Gabriel illustrates, you can understand the sentence, "My horse was hungry, so I filled a morat with oats and put it on him to eat" because of text compression. You can derive the meaning of morat from its larger context.

Compression lets you extend a class with less effort than having to build new behavior from scratch. That's why I like inheritance when it's carefully done. But you've got to study the hierarchy before you can extend it. The deeper the hierarchy, the more difficult mastering it can be. I suspect my design students who liked the inheritance-based solution felt they got real leverage out of the compression. The steps to add a new counter subclass were simple, and the hierarchy was extremely shallow.



<<Abstract>>
Counter
#currentValue: Comparable
#ceilingValue: Comparable
#nextValue(): Comparable
#currentValue(): Comparable
+increment(): void
IntegerCounter
#nextValue(): Comparable
#currentValue(): Comparable
+IntegerCounter(ceiling: Integer):IntegerCounter
CharacterCounter
#nextValue(): Comparable
#currentValue(): Comparable
+CharacterCounter(ceiling:Character): CharacterCounter
(a)
Counter
-currentValue: Comparable
-ceilingValue: Comparable
Counter(incr: Incrementer, ceiling: Comparable): Counter
+increment(): void
Incrementer
+nextValue(Comparable): Comparable
+initialValue(): Comparable
IntegerIncrementer
+nextValue(Comparable): Comparable
+initialValue(): Comparable
CharacterIncrementer
+nextValue(Comparable): Comparable
initialValue(): Comparable
(b)

Figure 1. A counter implemented using (a) inheritance and (b) composition.

Gabriel also cautions that inheritance compression can be tricky. Designing a deep hierarchy requires great care, and maintaining one requires much skill. Subclasses could break when a superclass changes undocumented invariants. To understand the complete context often requires good documentation and source code. The deeper the hierarchy, the harder that hierarchy is to know (and the easier it is for the class hierarchy designer to build in implementation constraints that hamper unanticipated extensions).

I suspect that those students who preferred the inheritance-based counter solution weren't worried about whether the abstract counter class might change out from under them. My example was too simple. But I use that simple example to illustrate that design preferences can be a matter of differing personal aesthetics. One solution might appear simpler to you because it seems natural, intuitive, or elegant. Your colleague might have a different opinion.

Simplifying a familiar design can require a fresh perspective. Robert Martin advises, "An axis of change is an axis of change only if the changes actually occur. It is not wise to apply the Single-Responsibility Principle, or any other principle, if there is no symptom." Simplify when you need to support a variation and not before. Designing incrementally, keeping it clean as you go, can help you avoid accidental complexity. But doing this takes discipline and design familiarity.

Richard Gabriel observes that "Piecemeal growth is the norm, and programs grow, change, and are repaired. Therefore, the perfectly beautiful program [or beautiful design] is possible only for very small programs, like the ones in books on programming or in programming classes." Gabriel challenges us to strive for software that is comfortable to live with and that supports embellishment, modification, and improvement through repair and extension. Just adding new functionality isn't enough. Rework, repair, redesign, and simplification must be ongoing when you do so, or your design's crispness will erode over time as it's passed on to future maintainers.

Rebecca J. Wirfs-Brock is president of Wirfs-Brock Associates and an adjunct professor at Oregon Health & Science University. Contact her at rebecca@wirfs-bock.com.