IEEE Software
www.computer.org/software

Principles in Practice

Rebecca J. Wirfs-Brock

Vol. 26, No. 4
July/August 2009

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

Principles in Practice

Rebecca J. Wirfs-Brock

Whenever two good people argue over principles, they are both right.—Marie von Ebner-Eschenbach

On what do you base your design decisions—established conventions, past experience, or principles? Most software designers are pragmatic: if an approach suits the problem at hand, we use it. Standards and conventions come and go and change. When our team adopts them, we try to follow them. Although committed to design quality, we can't always articulate guiding principles that underlie what we accept as good practice. We aren't die-hard fanatics who live or die by a rigid set of principles. Yet we'd like to know reasonable design heuristics and when to apply them.

When I first started programming in Smalltalk, I was awestruck by the programming prowess of those who seemed to have perfected object thinking. I felt at a distinct disadvantage, having just spent two years programming in assembly language. I observed experienced Smalltalk programmers making major revisions quickly. How did they manage to make change seem so effortless? What tricks would I have to learn to become a good Smalltalk designer?

After much thinking and observation, and some experimentation, I came up with three principles that seemed to underlie many of those experts' decisions:

- Distribute behavior among objects (rather than concentrate it into a single controlling object).
- Preserve design flexibility by hiding implementation details.
- Define abstractions and interfaces first (before focusing on data and coding details).

They were also doing many other things, but those principles seemed fundamental. But they were fairly general, too. They didn't tell me when a particular design choice was better than another so much as suggest what to consider as I worked through my design. They also gave me a set of criteria to evaluate my work: Is this class doing too much? Are its implementation details encapsulated? Does it represent a cohesive set of behaviors, rather than a grab bag of functionality?

Putting Principles to the Test

Robert Martin, in the best-selling book *Agile Software Development: Principles, Patterns, and Practices* (Prentice Hall, 2003), collected and named what he considers five fundamental design principles:

- The single responsibility principle (SRP): A class should have only one reason to change.
- The open-closed principle (OCP): Extending a class shouldn't require modifying that class.
- The Liskov substitution principle (LSP): Derived classes should be substitutable for their superclasses.
- The interface segregation principle (ISP): A class's clients shouldn't be forced to depend on interfaces they don't use.
- The dependency-inversion principle (DIP):

0740-7459/09/$25.00 © 2009 IEEE

> Abstractions shouldn't depend on details. Details should depend on abstractions.

According to Robert, the idea for defining software design principles came from his appreciation of science: "I had spent a lot of time studying physics and astronomy, and in those disciplines you find principles like the Heisenberg uncertainty principle or the polyexclusion principle, and so I rather like the idea of three words, with the last word being 'principle.'" (The podcast with this quote is at www.hanselminutes.com/dafault.aspx?showID-163; the transcript is at http://perseus.franklins.net/hanselminutes_0145.pdf.)

One definition of "principle" I like is, "an adopted rule or method for application in action" (http://dictionary.reference.com). A good design principle should help generate ideas and enable you to think through design implications. Most software design principles and practices tend to be rules of thumb rather than hard-and-fast rules. And therein lies the challenge: to find them useful, you must try them out.

Questioning the SRP

When I looked more closely, most of Robert's principles rang true with my own experience. Sure, I found cases where they simply didn't seem to apply. But it's up to me to see where (and whether) any principle fits. A principle isn't a rigid design rule. Like a design pattern, it's something that you apply in context.

But the principle I initially had the most trouble finding use for was the SRP. I think of a class as having a single purpose, collecting together a set of related responsibilities. In the context of the SRP, responsibility represents a "reason for a class to change." That is quite different from responsibility-driven design, where a responsibility is an obligation to perform a task or know certain information. Robert and I were meaning totally different things when we used the word "responsibility." Once I understood that difference, I could go on to explore all that this principle implied.

A class should have only one reason to change. Classes, if designed right, support a certain degree of variability. Instances don't have to have identical behavior. They support the same responsibilities, but depending on their current state, they might react quite differently. So what kind of change would force me to refactor some behavior into a new class?

Should something that varies always be factored into another distinct abstraction? My first worry was that, if carried to extremes, the SRP would produce a design filled with classes that represented too many tiny variations on a common theme. I don't consider that good design. But Robert didn't imply that was good, either. He took care in his writing to explain that you should factor responsibilities into different classes only if there are requirements causing them to change. He didn't recommend pulling out variations into tiny classes (although someone, without thinking through the consequences, might do just that).

Instead, he suggested you invent new abstractions to represent the thing that varies and declare an interface so that any class can implement that responsibility, regardless of inheritance. Robert also cautioned, "An axis of change is an axis of change only if the changes actually occur. It is not wise to apply the SRP, or any other principle for that matter, if there is no symptom." Generally, I agree.

It's a Judgment Call

So, the SRP provides a hint about when to create an abstraction. Actually, it's just another way of saying, "Keep a class's behaviors cohesive." I don't always form new classes to support individual behaviors. Nor should I be forced to. The SRP isn't a prescription, just general advice on what to do when a class's behavior isn't cohesive enough. I should balance the cost of creating that abstraction with supporting it more simply.

Michael Feathers, in *Working Effectively with Legacy Code* (Prentice Hall, 2005), points out two ways the SRP can be violated:

- at the interface level, when a class presents an interface that makes it appear responsible for many things, and
- at the implementation level, when it really does implement many things.

If a class implements responsibilities by delegating work to other classes, it's a façade—providing an interface to a number of smaller classes. This might be okay and exactly what you intend—to hide those classes performing specific responsibilities from others.

On the other hand, if a class directly implements many diverse responsibilities, this is an implementation violation of the SRP that you should resolve by modifying your design. Also, a class that provides an interface to a related set of responsibilities is well designed only if its implementation isn't overly complex or tangled. Only by looking at the code and variable definitions will you know whether there aren't other problems. Both the interface and the implementation need examining.

So what makes for a good software design principle? Echoing the sentiments of the military strategist Carl von Clausewitz, "Principles and rules are intended to provide a thinking man [or woman, in my case] with a frame of reference." I find it refreshing to occasionally step back to deeply examine why one design option seems better than another. I get uneasy when tribal knowledge about "the way things work around here" or vague, hard-to-express sentiments are the only reasons for a particular decision. I guarantee that if you discuss with your colleagues the nuanced reasons for making a particular design choice, you'll learn more about putting design principles into practice.

A good design principle should help generate ideas and enable you to think through design implications.

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