The illusion of simplicity

by Grady Booch
Chief Scientist
Rational Software
IBM Software Group

I recently met with the CTO of a large company who was charged with providing the technology infrastructure upon which his organization's developers could craft their applications. I spent much of our meeting commiserating with him about the many challenges he faced in trying to weigh the tradeoffs between Microsoft's .NET and IBM's WebSphere while at the same time keeping a large legacy platform alive. One thing that he said stuck with me all day: Ultimately, he said, he was rewarded for making things simple.

Simplicity is an elusive thing. In a software-intensive system that might consist of hundreds of thousand of lines of custom code on top of several million lines of middleware code on top of several million lines of operating system code, there is an essential complexity that cannot be eliminated. From the perspective of its end users, simplicity manifests itself in terms of a user experience made up of a small set of concepts that can be manipulated predictably, logically, and consistently. From the perspective of those who deploy that system, simplicity manifests itself in terms of an installation process that addresses the most common path directly while at the same time making alternative paths accessible and intuitive. From the perspective of the developers who build that system, simplicity manifests itself in terms of an architecture that is shaped by a manageable set of patterns that act upon a self-consistent, regular, and logical model of the domain. From the perspective of the developers who maintain that system, simplicity manifests itself in the principle of least astonishment, namely, the ability to touch one part of the system without causing other distant parts to fall off.
Simplicity is most often expressed in terms of Occam's Razor\(^1\). William Occam, a 14th century logician and Franciscan friar stated, "Entities should not be multiplied unnecessarily." Isaac Newton projected Occam's work into physics by noting, "We are to admit no more causes of natural things than as are both true and sufficient to explain their appearances." Put in contemporary terms, physicists often observe, "When you have two competing theories that make exactly the same predictions, the one that is simpler is the better." Finally, Albert Einstein declared that "Everything should be made as simple as possible, but not simpler."

As software developers, we cannot measure simplicity, but we do know it when we see it. As C.A. R. Hoare observed, "There are two ways of constructing a software design; one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

As such, you’ll often hear programmers talk about "elegance" and "beauty," both of which are projections of simplicity. Don Knuth’s work on literate programming -- wherein code reads like a well-written novel -- attempts to bring beauty to code. Dick Gabriel's work on the "quality with no name" which builds upon the architect Christopher Alexander's work, also seeks to bring beauty and elegance to systems. In fact, the very essence of the patterns movement, from the Gang of Four and beyond, encourages simplicity in the presence of overwhelming complexity by the application of common solutions to common problems.

Simplicity in software design is important because we as humans have a limited capacity for dealing with complexity. Indeed, I often know that my designs are getting simpler when the implementation gets smaller, primarily due to refactoring that extracts these common solutions. For example, as we were working on the original UML metamodel, we’d see the number of classes in the model grow (as we added semantics) and then shrink (as we discovered some underlying simplicity that was previously hidden). This regular breathing of our metamodel was a good measure as to the health of our design; too rapid a rhythm and it was clear we were churning, but a steady, slowing rhythm told us that we were on track.

The entire history of software engineering can perhaps be told by the languages, methods, and tools that help us raise the level of abstraction within our systems, for abstraction is the primary means whereby we can engineer the illusion of simplicity. At the level of our programming languages, we seek idioms that codify beautiful writing. At the level of our designs, we seek good classes, and in turn good design patterns, that yield a good separation of concerns and a balanced distribution of responsibilities. At the level of our systems, we seek architectural mechanisms that regulate societies of these classes and patterns.

Buckminster Fuller once said, "When I am working on a problem, I never think about beauty. But when I have finished, if the solution is not beautiful, I know it is wrong." In short, beauty in software is an important after-effect of simplicity in design.
Notes

1 In particular, see http://math.ucr.edu/home/baez/physics/occam.html

This article was originally published on Rational Developer Network, the learning and support channel for the IBM Rational customer community. If you are an IBM Rational customer and have not already registered for your free membership, please go to www.rational.net

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!