A Gram of Prevention

by Chuck Allison

At the June 2008 Better Software Conference & EXPO in Las Vegas, I presented a tutorial on the principles and patterns of software design. Much of the discussion focused on how understanding and adhering to sound design principles lead to better code. Most design principles are based, it seems, on the notion of *separation*: separating things that change from things that don't in a given context, separating interface from implementation, separating object creation from object use, etc. While studying the virtues of separating command invocation from command internals, a tester in the audience observed that the Command pattern makes program functionality easier to test since functions can be tested independently of their calling contexts. Being somewhat of a focused (read *myopic*) developer, I never considered how design patterns could improve code testability.

Similarly, it seems that thinking before coding is also a Good Idea. Back when the turnaround time of program builds was much longer and more expensive than it is now, developers had to be very careful crafting code, or else they quickly ran out of time or budget or both. While modern IDEs and agile processes encourage a healthy project velocity, they seem to have begotten a generation of quick-draw programmers who are all too willing to let their tools do the programming. But following an *I-click-therefore-I-program* methodology does not lead to quality software. Agile methodologies, in part a reaction to the extremism of Big Up-Front Design, may not only have helped small-to-medium-sized projects break free from analysis paralysis but also may have encouraged a knee-jerk abandonment of almost *all* up-front design. Consequently, the sorry state of software quality throughout the industry has seen little measurable improvement [1,2].

Famed software visionary Grady Booch recently commented on how fast-and-loose code slinging compromises architectural integrity:

> In other disciplines, engineering in particular, there exist treatises on architecture. This is not the current case in software, which has evolved organically over only the past few decades. All software-intensive systems have an architecture, but most of the time it's accidental, not intentional. This has led to the condition of most software programming knowledge being tribal and existing more in the heads of its programmers than in some reference manual or publicly available resource … If I don't have a sense of the architecture, and I keep piling on code, it becomes a fetid mess [3].

> "Good code can and should evolve from clear, up-front descriptions of the solution to the problem at hand."

Some up-front design must be performed. Due diligence early on finesses much of the needless complexity that inevitably ensues with unarchitected code. According to programming pioneer Per Brinch Hansen, "Once you appreciate the value of description as an early warning signal of unnecessary complexity, it becomes self-evident that program structures should be described (without detail) before they are built and should be described by the designer (and nobody else). Programming is the art of writing essays in crystal clear prose and making them executable [4]."

If we can't manage our plans and descriptions, we won't be able to keep the code that follows under control.

This is all very old news, by the way, as you can see by the date of the previous quote. In his Turing Award Lecture a few years earlier, the legendary Edsger Dijkstra asserted that:

> Those who want really reliable software will discover that they must find means of avoiding the majority of bugs to start with, and as a result the programming process will become cheaper. If you want more effective programmers, you will discover that they should not waste their time debugging; they should not introduce the bugs to start with [5].

When I read this to an audience of testers at STAR*WEST* 2001, I had to wait for the laughter to subside before I continued, but Dijkstra knew what he was talking about. It pays to put in the effort to develop an architecture, to design for maximal cohesion and minimal coupling, to use or develop languages close to the problem domain, to create abstractions according to sound design principles—the list goes on. Testers will not lose their jobs if developers can deliver higher quality to begin with. But software just might be raised from its dismal state. **{end}**

References:

1] "The Economic Impacts of Inadequate Infrastructure for Software Testing." NIST Report, May 2002. www.nist.gov/director/prog-ofc/report02-3.pdf.

2] Frye,Colleen. "The State of Software Quality, Part 1." Software Quality News, February 16, 2007. searchsoftwarequality.techtarget.com/news/article/0,289142,sid92_gci1243311,00.html.

3] Booch, Grady. "Software's Dirty Little Secret." Scientific American, June 17, 2008. www.sciam.com/article.cfm?id=softwares-dirty-little-secret.

4] Brinch Hansen, Per. *The Architecture of Concurrent Programs*, Prentice-Hall, 1977.

5] Dijkstra, Edsger. "The Humble Programmer." ACM Turing Award Lecture, 1972.