## **Complex Software Is Just Too Complicated!**

## (And How to Save Your Project with the 1-5-10 Rule)

Chuck Connell, BeautifulSoftware.com

Many software projects collapse under the weight of their own complexity. During the collapse, these projects go through familiar stages as they die a painful death. First come minor schedule slips, then a missed milestone, then an adjustment to the master schedule extending the end-date a bit, then 80-hour weeks for the programmers, then the altered schedule does not work either, then management looses touch with reality and insists all is fine, and then the end-date comes and goes with no working software. Then the finger pointing begins to assess blame.

Why does this happen again and again, to competent organizations that are able to complete other types of projects efficiently?

The problem is that many of the world's large software projects are just too darn complicated. I have written about this problem before in my article titled *Why Bad Things Happen to Big Software Projects* and in several chapters of my book, *Beautiful Software*. Of course, some large software projects fail from normal incompetence, and a smarter team could have pulled them off. But in many cases, especially for large projects, software engineers and designers create pie-in-the-sky architectures that are beyond the limits of human comprehension. The software becomes so complicated that no one is capable of putting it together.

So what is the solution? Never take on a large software project?

The answer is that the larger the program, the simpler it should be. The reason for this is obvious, although often overlooked. As a software system becomes large by any measure (number of servers, number of users, size of data set, number of transactions, etc) the greater its inherent complexity. To counteract this built-in problem, the software must mitigate the complexity by becoming even simpler. To say it another way, small programs can be complicated and still work, but large programs must be simple if they have any chance of working.

## **Example from Hardware Design**

Computer CPUs provide a good example of internal simplicity counteracting overall complexity.

Consider a large computer system such as the IBM zEnterprise 196 mainframe. This physical machine can host thousands of virtual servers at the same time, with each virtual server running hundreds of simultaneous processes. Each of the processes can be interacting with other processes within the mainframe, so the number of input/output scenarios is huge. The overall complexity of the entire computing workload is mind boggling. How does all of this work correctly without descending into chaos?

A mainframe works because the internal design of the CPU is rather simple. The z196 chip has relatively few hardware instructions, each of which performs a single, small operation. The hardware development team tests each CPU instruction very carefully to be sure it does its function properly. If each instruction works right, a long sequence of instructions will be correct, and overlapping instruction streams will be correct also. Seen from the outside, a running mainframe computer is truly incomprehensible, but it works because the core is simple and there is nothing else being done other than many, many repetitions of the core instructions.

## 1-5-10 Rule

The CPU analogy shows the kind of simple core that large software systems need, but, unfortunately, often do not have. The innate complexity of real-world software problems lull us into designing complicated solutions, which then fail. Instead, the bigger the project, the harder we need to work to beat down the intricacy of our approach.

To this end, I introduce the 1-5-10 Rule of Software Design:

- The goal of the software is clearly described in a one-minute talk.
- The overall architecture is clearly described in five more minutes.
- *Key technical details are clearly described in ten more minutes.*

We need large software systems that are so clean and simple their main design points can be communicated in a 16 minute talk. Systems that cannot be so described are likely heading for trouble as we try to implement them.

Some guidelines for the 1-5-10 Rule:

- Each talk must make sense to a listener with good general computer knowledge, not just to someone already on the project team.
- Diagrams and pictures are fine, but they should be simple and there should be no more than one for the first talk, two for the second and three for the third.
- If the presentation is written instead of spoken, the guidelines are 100 words, 500 words and 1000 words respectively.

The first part of the talk (one minute) explains why you should care about this software; what good it will do for the world. The talk includes no technical jargon, except for terminology in the problem domain. For example, it is OK to say, "This software will perform DNA sequencing, determining the order of the nucleotide bases—adenine, guanine, cytosine, and thymine—in a molecule of DNA." This jargon is acceptable because it describes *what the software will do*. But it is not OK to say, "This software will consist of abstract generic classes conforming to the Java J2SE 5.0 spec." This jargon is not acceptable because it describes *how the software will work*.

The first part of the talk should also pass the laugh test. So, it is OK to say "This software will perform the check-in/check-out procedure at the main New York City Public Library." But it is a comical recipe for disaster to say, "This software will tie together all of the libraries in the world with a common check-in/check-out procedure."

The second part of the talk (five minutes) *does* describe how the software will work. It covers the key user interfaces, most important files, message passing mechanism, major algorithms, etc. At the end of this talk, a knowledgeable listener should have a broad understanding of how the software does what it does. If it is not possible to explain this material in five minutes, this is a good indication that the software design is too complex.

The third part of the talk (ten minutes) dives into the most important aspects of the software design, giving a listener with adequate background a strong understanding of the software's operation. An example is the "Carrier Sense Multiple Access with Collision Detection" algorithm of the Ethernet protocol. If someone were proposing the CSMA/CD protocol for the first time, this topic would be worth an explanation during the ten minute talk. Another example is a new virtual memory paging routine in an operating system. Since the performance of this code is of central importance to the overall operating system, it is worth some explanation here. Of course, a programmer could not join the project team and start contributing code immediately after hearing this ten minute talk. But the talk would be a good start in this direction.

I suspect that some readers will say, "This is impossible. My software is too complicated to describe in 16 minutes!" This is exactly the point. Software designs that cannot be quickly explained, and grasped by an appropriate listener, are often too complex to work correctly.

Simple is good. Simple is beautiful. Simple is better.

Chuck Connell knows software projects from all sides. He has been a hands-on programmer, a software product architect, a manager of programmers, a university teacher of software engineering and a consultant. His consulting practice <u>BeautifulSoftware.com</u> helps organizations assess the health of software projects, manage ongoing projects and turn around troubled projects.

For more information....

www.chc-3.com/pub/bad\_things\_big\_software.pdf, my related article www.amazon.com/dp/1456438786/, my book *Beautiful Software* www.ibm.com/systems/z/hardware/zenterprise/z196.html, z196 system description publibz.boulder.ibm.com/cgi-bin/bookmgr\_OS390/download/A2278325.pdf, z196 CPU