Reducing Software Complexity

Revision 0.0 © 2011 John F. Beetem
This document is Chapter 1 of *The XXICC Anthology*.

> *Everything should be made as simple as possible, but not simpler.*
> [Albert Einstein]

> *Simplicity, when it removes encumbering details, makes for beauty in music, in art, and in living. It clears the springs of life and permits wholesome mirth and gladness to bubble up; it cleans the windows of life and lets joy radiate.*
> [Quaker Philadelphia Yearly Meeting, *Faith & Practice*, 1961]

1. Prologue

In the early 1980's a controversial idea called RISC (Reduced Instruction Set Computer) started to take root in computer architecture. The basis behind RISC is that even though it had become possible and inexpensive to build computers with an extremely rich collection of opcodes and addressing modes, it was not actually a good idea to do so. Before RISC, computer architectures proudly proclaimed the number of instructions, native data types, and addressing modes, asserting that "more is better". The RISC movement noted several problems with this assertion:

1. Writing compilers that can take advantage of a huge number of opcodes and addressing modes is very difficult, especially if the instruction set is irregular. Complex compilers are very hard to debug and update. In fact, most compilers only used a subset of the available opcodes and addressing modes.

2. The presence of rarely-used opcodes and addressing modes slows down the execution of frequently-used opcodes and addressing modes. Simple decoding logic is faster than complex decoding logic.

3. Any irregularity in the instruction set makes pipelining difficult.

With RISC, each opcode and addressing mode must justify its existence. Adding an opcode or addressing mode always slows down the CPU cycle time, but if adding it results in fewer instructions so that the overall software performance improves, then doing so is worth it. On the other hand, if emulating the opcode or addression mode using the existing instruction set has better overall performance, then do not add the new feature no matter how tempting it is to do so.

RISC was treated with scepticism in the 1980's [B&M 85] but it has mostly triumphed over CISC (Complex Instruction Set Computer) architectures. The principal holdout is the x86 Intel Architecture, which achieves high performance by using huge numbers of high-power transistors cooled by increasingly sophisticated schemes. In contrast, the most successful RISC architecture is ARM (Advanced RISC Machines), which is almost universally used in cell phones because of superior performance per watt.

So what does this have to do with software? Isn't software free, or essentially free because the cost per MB of Flash and DRAM has become so ridiculously low that people have Flash cards that programmers would have killed for back in 1980 filled with inane snapshots? Well, that's true in the same sense that in 1980 transistors became so cheap that adding new opcodes, addressing formats, native data types, and instruction

formats was a no-brainer... until you calculated the impact on compiler complexity and decode logic performance, at which point it became clear that RISC was better. So let's see if adding that new software feature is really free after all.

2. Introduction

The last few decades has seen an explosion in software complexity, fueled by massive increases in computer performance coupled with a staggering drop in cost of memory. Modern desktop computers have gigabytes of DRAM and hundreds of gigabytes of disk. Because computers are so fast (to be revisited later) and memory is so cheap, software complexity has practically no direct cost. However, indirect costs are present and increasing.

Let's first see where we've come from.

One of the most popular computers of the 1970s was the DEC PDP-11. It had a small but extremely well thought-out instruction set which was more pleasant to program in that most of the high-level languages available at the time. One of the key reference books was the *PDP-11 Peripherals Handbook* [DEC 73], which provided everything a programmer needed to know to interface to disk drives, printers, and every other DEC product that could interface to a PDP-11.

The *PDP-11 Peripherals Handbook* has a very nice chapter called "Programming", which describes -- with assembly-language examples -- how to write device-driver code for a typical PDP-11 peripheral, using both wait loops and interrupts. This chapter is *eight pages long*. Its simplicity invites the programmer to write some code and try it out. Compare this to device driver reference books with hundreds of pages for Windows and Unix, which seem to have the goal of preventing programmers from getting anywhere near their computers.

Well, nowadays the PDP-11 is considered a relic from the past, a 16-bit computer that could not directly access more than 64KB of memory. The argument today goes that we need multi-GB to run things like Windows and Linux with their snazzy GUIs. Of course, that argument forgets that the original Unix was developed on a PDP-11, as was the C programming language. That argument also forgets that GUIs worked very nicely on the original Apple Macintosh, which had 128KB of RAM. Now scaling up screen sizes does require a few extra megabytes, but gigabytes? Is a 1GB Windows machine today really 16,000 times better than a PDP-11 with 64K? Or is it only 100 times better, with rest pure waste?

Now that we've seen where we came from, let's look at the indirect costs of software complexity.

- Learning to use the software: As more features get added to software, getting started as a new user gets increasingly difficult. It used to be that software came with a user's guide, but nowdays software is supposed to be "intuitive" and there is no user's guide. There's on-line help, but that is usually far from adequate. An amusing example of this is how Microsoft guru Charles Simonyi couldn't figure out how to switch off Microsoft Office's infernal "Clippy" even though he managed Office development [SR 07, p. 38].

- Developing and maintaining the software: As software increases in size, the number of possible interactions rises even faster. This makes adding features more expensive, since they may affect more code. The effect on bugs gets even worse: more interactions results in more possible bugs and they get more subtle and difficult to find and reproduce.

  Here is an even more damaging concern: Once software hits a certain level of complexity, no one person understands it any more. Programmers then become reluctant to make large changes that may

improve program structure and maintainability because they fear they may break code they do not understand. Instead, they add patches upon patches, until the orginal software structure vanishes within a tangled mess of patched-together features. According to Alan Perlis, "Every program eventually becomes rococo, and then rubble."

An excellent example of this is Macintosh OS version 7. Prior to version 7, an individual programmer could read *Inside Machintosh* volumes 1-5 and learn everything anyone needed to know to program the Macintosh. With version 7, everything became much more complex (in this author's opinion) and you needed a team to do any significant programming. No one person understood the entire program any more, and the path to rococo (and then rubble) became assured.

- Loading and running the software: One could argue that computers are so fast today that software complexity does not have a significant effect on performance as far as users are concerned. This argument fails to hold water as each faster computer comes with a new version of the operating system that takes even longer to boot. It is patently absurd that a Radio Shack TRS-80 with a Z80 microprocessor can boot instantantly while it takes a minute for a 1GHz 64-bit processor to do so. This is clearly an example of overly complex software. Yes, it is possible for users to switch off features to make their computers boot faster, but most cannot because the system is so complex and fragile that they do not know which ones can safely be turned off.

  While CPU performance and memory size and performance have increased, the I/O bandwidth between them has not grown anywhere as quickly. It certainly has not kept up with the increase in program size, which is why it takes longer to load and run so many programs. Installing large programs takes much longer as well. Yes, CD-ROMs are much faster than floppy disks, but not when you have to install hundreds of megabytes.

  The effect is particularly dramatic with embedded computers like cell phones, which have less memory and run processors slower to conserve power. There is a reason cell phones do not run full implementations of Windows or Linux.

- Power consumption: Intel has shown us that it is possible to get high performance from bloated software by using lots of hot transistors. Similarly, memory bandwidth has increased significantly using DDR schemes. However, all this takes considerable electrical power, which adds up over the millions of computers in operation. How much of this power is actually being put to good use?

CAR Hoare has an excellent paper called "The Emperor's Old Clothes" [CH 81] which discusses software complexity at length. The title comes from a story an emperor who acquired a fine new set of clothes each year, but insisted on draping the new set of clothes on top of the existing ones. Eventually there was a huge, immovable mound of beautiful clothes but nobody could see the emperor any more.

This is often the effect of attempting to maintain legacy compatibility by adding patches and layers to emulate previous versions. At some point it becomes far better to discard the old code and start over.

3. How to Reduce Software Complexity

3.1 Omit Unnecessary Features

Following Einstein's and Occam's advice, we must avoid the temptation continually to add unnecessary features to software. The author's own experience is that new versions usually do not add any new features he really needs, and often screw up existing features that he does need. In many cases, it seems that the new features are there solely so there is a reason "on paper" to upgrade to the new version. Technology

magazines are complicit in this, since they tend to celebrate the new features without considering whether they are useful or merely "newsworthy".

Each software feature inherently reduces usability and productivity by making the software more complex to the user. As with RISC opcodes and addressing modes, each new software feature should have to justify its existence by proving that its presence improves usability and productivity more than the inherent losses. If users can accomplish the same function with existing features, is it really necessary to add the new feature? If it's used often enough to improve productivity significantly, it's worth it. Otherwise, it's probably not.

A fine example is Microsoft Office's infernal "Clippy", mentioned earlier. While the author has met people who find Clippy entertaining and impressive animation, he has yet to meet anyone who actually finds Clippy useful. Most have to either put up with Clippy's nonsense or have taken the time to figure out how to turn Clippy off. In any case, the large amount of development time that went into Clippy seems to be a total waste from the author's point of view.

In general, one should minimize the number of ways of doing something. For example, having keyboard equivalents of menu selections is good, but tabbing between check boxes and groups of check boxes compilicates the code for dialog boxes and makes it harder to get right. While one can argue that users can choose to ignore the feature if desired, the reality is that a misplaced keystroke can trigger the feature unexpectedly, resulting in user confusion.

A good way to control proliferation of features is to require authors to document all features before implementing them. Since many programmers hate to document anything, this limits new features to ones that those programmers want really badly. It also improves the quality of software documentation.

3.2 Encapsulate Complexity

Once a software feature has justified its existence, adding it inevitably increases software complexity both for users and developers. Modular design can reduce this effect significantly by encapsulating complexity. For example, putting "advanced options" is a separate dialog from "basic options" presents a simplified view to new users while allowing sophisticated users to have full control. Similarly, APIs benefit from having basic and extended versions of system calls.

Developers can reduce complexity by encapsulating advanced features in separate program modules so that basic functionality is simplified. The key to making this work is well-defined and well-documented interfaces between modules. Documenting what functions do is far more important than writing the code of a function, particularly as the software is maintained and improved.

Abstraction layers can also be a powerful tool to encapsulate complexity. One of the most successful examples of this is networking, which uses the Open Systems Interconnection (OSI) Model to layer the functionality at the physical, link, network, transport, and other layers. All physical layer considerations such as mechanical connection and electronic components are encapsulated so that the link layer and above do not need to consider them. Similarly, the phyical layer does not care what data or control is being set over it. The link layer only considers immediate connections between nodes, so it does not have to deal with alternate routing paths and other higher level considerations. The networking layer such as IP does not care what underlying technology (Ethernet, USB) is used to transfer data between adjacent nodes, so it can concentrate on routing. In IP, the networking layer makes a "best effort" to transfer data reliably, but could lose packets. However, it does not need to recover from this, which keeps IP simpler. The transport layer deals with reliable data transfer with a protocol such as TCP.

As long as layers have well-defined and well-documented interfaces, it is relatively easy to replace a layer

with a different underlying technology, or to provide alternate technologies for a given layer.

Operating systems should also be designed in layers, so that one can easily substitute lower layers to run on different machines or higher layers to provide the infrastructure for applications. This has only been moderately successful, as evident by the fact that so many applications are Windows-only or Linux-only. There is no operating-system equivalent to the OSI model. POSIX, X Windows, and TrollTech's Qt are notable steps in the right direction as they provide an adaptation layer to permit applications to run on multiple systems.

Abstraction layers are attractive, but programmers and users must beware of the *Law of Leaky Abstractions* [SR 07, SR 08]. The problem is that no matter how carefully one defines abstraction layers, there is always the possibility that bugs will cause a lower-level effect to show up at a higher level. A classic example of this is the buffer overflow problem in C. C arrays have defined sizes, but the object code does not enforce array limits so it is possible for an erroneous program to read or write data past the end (or before the beginning) of an array, clobbering other data or even clobbering return addresses on the stack. This is a well-known way of introducing worms into unsuspecting computers. C is often called a "high level" language, so naïve C programmers may expect that writing to an array cannot affect anything other than the array. In fact, C should be thought of as a "portable assembly language", which uses high-level notations to write portable machine language. Once C object code is running, its abstraction level is machine language. A correct program behaves as if it is a high-level abstraction. But erroneous programs can cause fail in ways that can only be understood at the machine language level.

3.3 Reuse Modules

An important way to reduce complexity is to reuse software modules whenever possible. This reduces program size and improves test coverage (and therefore software quality) since modules are executed more frequently. A single bug fix can fix many programs at once. Dynamically linked sharable libraries make this easy to do, so there is really no excuse.

One of the most glaring examples of where this is *not* done is the Microsoft Office suite. Microsoft Office began as a set of independent tools acquired from various companies with minimal interaction between them. After decades, Microsoft Office is *still* a set of independent tools, even though there is no technical need for them to be independent tools. So while you can create a table in Word, you cannot fill the table with dynamically calculated expressions like Excel. While you can create text in PowerPoint, you do not have the same capabilities as Word for adding special characters. You would think that after decades the user interfaces would have become completely common between the tools, or better yet, Microsoft would have developed a generalized tool that covers all the functionality. The author can only assume that the internal complexity of the tools is so high that no one person knows how they all work, so everyone is afraid to do more than maintain the code, adding rococo features until the software eventually becomes rubble.

Reusing software modules may require adding parameters for different contexts. This adds complexity, and at a certain point it no longer makes sense to reuse a module since doing so would complicate its use elsewhere. When this occurs, consider that:

- It may not be appropriate to reuse the module in the given context. However, it may be possible to reuse components of the module.

- It may be time to rethink the module interface entirely. Perhaps there is a generalization of the module that would simplify its use everywhere.

3.4 Generalize

Generalizing software features and modules is probably the most difficult of all software development tasks. It is relatively easy to take a well-written set of specifications and implement it as code. Taking several bodies of code that were created for different purposes and finding the common elements so as to create a general form is challenging, particularly with deadlines looming. Yet, it is usually the best way to achieve large reductions in complexity and if successful will save huge amounts of effort in the long run.

A useful analogy here is Plato's *Allegory of the Cave* [Wiki 1] in which prisoners are held from birth in a cave in which their only life experience is shadows projected onto the walls. The shadows become their reality, and they are unable to conceive of the three-dimensional objects that cast those shadows. Similarly, particular implementations of software are projections of abstract objects into a specific programming language. A generalized object, if it exists, may project into many different implementations. If a programmer can conceive of the generalized object, then each of the specific implementations are simply special cases of the generalized object with general parameters given fixed values.

The challenge is that the generalized object is currently abstract, which makes it very difficult to think about and write about. Object-oriented programming styles have tried to realize this goal with limited results. Some day there will be programming languages that achieve this, making C and C++ look like assembly language in comparison.

An excellent example of generalization and reuse is the PDP-11 register set. Previous DEC computers had an accumulator. The PDP-11 has a set of eight "general-purpose" registers, which could be used for all opcodes and addressing modes, with a few minor restrictions. Two of these registers are not quite general-purpose: R6 is the stack pointer and R7 is the program counter. However, by using the standard opcodes and addressing modes with R6 and R7 the PDP-11 gets stack operations, immediate operands, and PC-relative addressing for free.

Similarly, some RISC machines designate register 0 to always have the value 0. This allows the RISC machine to omit instructions like "negate", since it can be done by subtracting from 0. It also allows the machine to implement direct addressing using base-displacement addressing with register 0 as the base register. This simplifies the set of opcodes and speeds up decoding.

3.5 Start Over

In the hurry to ship software there is a tendency to program incrementally by making small changes to existing programs. While this makes sense for bug fixes and minor tweaks, at some point adding one feature after another makes the software so complex that it collapses under its complexity: "rococo, then rubble". It then makes sense to "throw everything away" and take what one has learned to build an entirely new system.

The PDP-11 is once again a good example. It was created as a totally new architecture, not at all based on the popular PDP-8 or PDP-10. Best of all, DEC formed a team of both computer designers and system programmers who worked together to create an extremly elegant architecture. On the other hand, one could argue that the PDP-11's beauty and simplicity was largely because of economic considerations and the available gate density available at the time. This is true to some extent, but the reality is that the PDP-11 was much nicer than competing minicomputers of the time with the same number of gates.

Whenever creating a new system, one makes many decisions based on the constraints at that time. As time progresses, many of those constraints vanish yet the system continues as if they were still there. For example, most programming languages use monospaced ASCII characters, typically 80 column lines. This is a throwback to 80-column punch cards, which most programmers have not seen in decades, if at all. Yet the

monospaced ASCII practice continues, even though proper mathematical notations have long been supported by displays and printers.

The great supercomputer designer Seymour Cray was a big proponent of throwing things away and starting over from scratch. He felt that he could not develop the next world's fastest computer by just reworking current machines. According to legend, he also applied this same philosophy to his hobby of building sailboats. Each winter he would build a new sailboat in his basement, bring it out in the spring, use it all summer, and at the end of the season he would burn it in a kind of Viking funeral. Next year's sailboat would always be better since he had learned from the previous one, and it did not have to carry along the mistakes of the previous years.

4. General References

Here are some general references on the subject of excessive program size and its implications.

1. Ed Perratore, Tom Thompson, Jon Udell, Rich Malloy, "Fighting Fatware", *Byte*, April 1993.

2. Niklaus Wirth, "A Plea for Lean Software", *IEEE Computer*, February 1995.

3. Robert Capps, "The Good Enough Revolution: When Cheap and Simple is Just Fine", *Wired*, August 2009.

5. Specific References

[B&M 85] Clifford Barney and Tom Manuel, "RISC: Is it a Good Idea, or just another Hype?", *Electronics*, May 5, 1986.

[CH 81] C.A.R. Hoare, "The Emperor's Old Clothes", *Communications of the ACM*, 1981.

[DEC 73] Digital Equipment Corporation, *PDP-11 Peripherals Handbook*, 1973.

[SR 07] Scott Rosenberg, "Anything you can do, I can do Meta", *Technology Review*, Jan/Feb 2007.

[SR 08] Scott Rosenberg, *Dreaming in Code: Two Dozen Programmers, Three Years, 4,732 Bugs, and One Quest for Transcendent Software*, Three Rivers Press, 2008.

[Wiki 1] http://en.wikipedia.org/wiki/Allegory_of_the_cave.