Keep It Simple

Prospecting for programmer's gold.

IF ALEXANDER THE GREAT COULD conquer the known world by the time he was 18, you would think adults could conquer the bits of complexity contained in the taupe-colored boxes on their desks.

Unfortunately, these "bits of complexity" aren't as simple as some people assume. Computing is the only profession in which a single mind is obliged to span the intellectual distance from a bit to a few hundred megabytes, a ratio of 10^9, or nine orders of magnitude. The immensity of this ratio is staggering. As Edsger Dijkstra says, "Compared to that number of semantic levels, the average mathematical theory is almost flat. By evoking the need for deep conceptual hierarchies, the automatic computer confronts us with a radically new intellectual challenge that has no precedent in our history" ("On the Cruelty of Really Teaching Computer Science," *Communications of the ACM*, Dec. 1989).

At the 1972 Turing Award Lecture, Dijkstra argued that most programming is an attempt to compensate for the limited size of our skulls—to manage the enormous complexity associated with modern software systems. Some software complexity is inherent in the problems we try to solve, but a large part depends as much on the solution as the problem. The best solutions are those created by people who realize just how small their skulls are and tailor their solutions accordingly.

HIERARCHIES AND ABSTRACTIONS. Hierarchies and abstractions are two of the most effective ways to manage complexity. A hierarchy is a tiered, structured organization in which a problem space is divided into levels that are ordered and ranked. In a hierarchy, you handle different details at different levels. The details don't go away completely; you simply push them to another level so that you can think about them when you want to rather than all at the same time. Hierarchies come into play most obviously in the module hierarchy of a functional design, but also in inheritance hierarchies in object-oriented design, nested data structures, and many other cases.

Using hierarchies comes naturally to most people. When we draw a complex object such as a house, for example, we tend to draw it as a hierarchy. As Herbert Simon points out in *The Sciences of the Artificial*, we first draw the house's outline, then the windows and doors, then additional details. We don't draw the house brick by brick, shingle by shingle, or nail by nail.

Abstraction is another way of reducing complexity by handling different details at different levels. Any time you work with an aggregate entity, you're working with an abstraction. If you refer to an object as a "house" rather than as a combination of glass, wood, and nails, you're making an abstraction. If you refer to a collection of houses as a "town," you're making another abstraction. Abstraction is a more general concept than hierarchy. It can reduce complexity by spreading details across a loose network of components, for example, rather than among a hierarchy's strictly tiered levels.

Programming productivity has advanced largely through increasing the abstractness of program components. According to Fred Brooks ("No Silver Bullets—Essence and Accidents of Software Engineering," *Computer*, April 1987), the move from machine language to higher-level languages produced the single biggest productivity gain ever made in software development. That move freed programmers from worrying about the detailed quirks of individual pieces of hardware and allowed them to focus on programming.

Programming productivity has advanced largely through increasing the abstractness of program components.

More recently, the advent of visual programming environments has greatly reduced the complexity associated with creating GUI applications.

Continued on page 127

Editor:
Steve McConnell
Construx Software Builders
PO Box 6922
Bellevue, WA 98008
smcconn@aol.com

Copyright © 1996 Steven C. McConnell. All Rights Reserved.

Continued from page 128

Visual programming environments allow programmers to work at an abstraction level at which they can forget about many GUI-related housekeeping details and focus on application particulars.

Neither hierarchies nor abstractions reduce the number of details in a program; they might actually increase them. Their benefit arises from organizing details so that fewer details have to be considered at any one time.

DESIGN GUIDANCE. Focusing on minimizing complexity yields valuable design guidance.

Subsystem design. At the software architecture level, you can simplify a problem by dividing it into subsystems. The more independent you make the subsystems—the more strictly you separate their concerns—the more you reduce complexity, and the more you enable programmers to focus on one thing at a time.

Classes and modules. Without classes or modules, the traditional advice to keep individual routines short becomes a double-edged sword. It helps readers understand each routine, but it tends to multiply the number of routines systemwide, which makes the system as a whole harder to understand.

Classes and modules, and for that matter subsystems, are helpful complexity-reduction tools because they provide an intermediate level of aggregation between individual routines and entire systems. With classes and modules, you can keep routines short but combine them into meaningful groups to keep complexity from exploding at the whole-system level.

Cohesion and coupling. The structured design guideline to build programs with strong cohesion and loose coupling arises from the need to manage complexity. The more loosely coupled two routines or classes are, the fewer interactions are possible and the less complex their relationship will be. The stronger a routine's cohesion, the neater a mental package it fits into and the less your brain has to remember and account for in the operation of its code.

Fan-out. The classic advice to limit *fan-out* (the number of routines a routine calls) might seem arbitrary until you realize that the underlying motivation for the advice is to limit the complexity that a programmer has to contend with at any one time. The computer can handle virtually any degree of fan-out; it's human software developers with small skulls who need a limit on the possibilities they have to consider simultaneously.

Information hiding. Information hiding is the practice of hiding design and implementation details behind abstract routine, module, and class interfaces. From a complexity viewpoint, information hiding is perhaps the most powerful design heuristic because it explicitly focuses on *hiding details*, which ipso facto reduces a program's complexity when viewed from any particular point of view.

CODING GUIDANCE. A focus on reducing complexity also helps cut through many historically nettlesome coding issues.

Global data. Global data lets virtually any part of a program interact with any other part of the program through their operations on the same data. Even a few global variables dramatically increase the complexity that a human reader has to deal with when trying to understand a program; for that reason global data compromises the programmer's primary objective of keeping complexity to a minimum.

Gotos. What guidance does complexity reduction provide for the historically controversial goto debate? Because gotos don't necessarily follow any specific pattern, your brain can't simplify their operation in any standard way. Gotos introduce flexibility that dramatically increases a program's complexity and therefore should be avoided.

By the same reasoning, if you need to use gotos to compensate for weaknesses in the programming language, do so—if such use serves to reduce a program's complexity from both the local and global viewpoints.

Coding standards. The complexity lens brings the purpose of coding standards into focus. From a complexity reduction viewpoint, the particular details of a coding standard almost don't matter. The primary benefit of a coding standard is that it reduces the complexity of having to revisit formatting, documentation, and naming decisions with every line of code you write. When you standardize such decisions, you free up mental resources for more challenging aspects of the programming problem.

One of the reasons that coding standards are often controversial is that the choice among many candidate standards is essentially arbitrary. Standards are most useful when they spare you the trouble of making and defending arbitrary decisions. They're less valuable when they impose restrictions in more meaningful areas.

LITMUS TEST. When programming is seen predominately as an attempt to manage complexity, the litmus test for any design or implementation approach becomes clear: Does the approach increase or decrease overall system complexity? If a design seems simple and yet accounts for all possible cases, it is a good design. If an implementation results in easy-to-read code that is more simple than clever, it is a good implementation.

Standards are most useful when they spare you the trouble of making and defending arbitrary decisions.

Our brains might not be capable of fully encompassing the mind-numbing details associated with creating a modern software system. But, paradoxically, if we approach software problems with a keen awareness that our skulls are smaller than we would like and tailor our approaches accordingly, we just might be able to conquer that world of details after all. ◆