CODE COMPLETE
by Steve McConnell

At the point at which you've identified the object interfaces and you are designing the code to support them, you usually switch to structured design. If you're working in an object-oriented language, it's hard to say you're not still doing object-oriented design because you're still working in terms of methods or messages or other object-oriented constructs. If you're working in a more traditional procedural language, it's easier to say that you're doing plain old structured design, which is appropriate at that point.

Object-oriented design is applicable to any system that acts as objects in the real world do. Examples of such systems include highly interactive programs that use windows, dialog boxes, buttons, and so on; object-oriented databases (by definition); and event-driven systems that require specific responses to randomly ordered events.

Much of the research being done on object-oriented techniques is focused on successful implementation of systems from 100,000 to over a million lines of code. Structured techniques have too often failed on such large projects, and object-oriented techniques seem to be a better solution. However, the object techniques are still useful on any but the largest projects, and the superiority of object-oriented design for smaller problems has yet to be proven.

7.5 Round-Trip Design

KEY POINT

It's possible to combine the major design approaches, making the most of their strengths and minimizing their weaknesses. Each of the design approaches is a tool in the programmer's toolbox, and different design tools are appropriate for different jobs. You'll benefit from exploiting the heuristic power of any or all of the approaches.

The following subsections describe some of the reasons that software design is hard, how to make it easier, and how to combine structured design, object-oriented design, and other design approaches.

What's a Round Trip?

You might have had an experience in which you learned so much from writing a program that you wished you could write it again, knowing what you learned from writing it. The same phenomenon applies to design, but in design the cycles are shorter and the effects downstream are bigger, so you can afford to whirl through the design loop a few times.

KEY POINT

The term "round-trip design" captures the idea that design is an iterative process: You don't usually go from point A just to point B; you go from point A to point B and back to point A. The term is inspired by a similar term in *Object-Oriented Design: With Applications* (Booch 1991).

As you cycle through candidate designs and try different approaches, you'll look at both high-level and low-level views. The big picture you get from working with high-level issues will help you to put the low-level details in perspective. The details you get from working with low-level issues will provide a foundation in solid reality for the high-level decisions. The tug and pull between top-level and bottom-level considerations is a healthy dynamic; it creates a stressed structure that is more stable than one built wholly from the top down or the bottom up.

Many programmers—many people, for that matter—have trouble ranging between high-level and low-level considerations. Switching from one view of a system to another is mentally strenuous, but it's essential to effective design. For entertaining exercises to enhance your mental flexibility, read *Conceptual Blockbusting* (Adams 1980), described in the "Further Reading" section at the end of the chapter.

Design Is a Sloppy Process

J. P. Morgan said that every person has two reasons for doing things: the one that sounds good and the real reason. In design, the finished product usually looks well organized and clean, as if the designers had never taken a wrong turn. The process used to develop the design is rarely as tidy as the end result.

FURTHER READING For a fuller exploration of this viewpoint, see "A Rational Design Process: How and Why to Fake It" (Parnas and Clements 1986).

Design is a sloppy process. It's sloppy because the right answer is often hard to distinguish from the wrong one. If you send three people away to design the same program, they might easily return with three vastly different designs, each of which is perfectly acceptable. It's sloppy because you take many false steps and go down many blind alleys—you make a lot of mistakes. Design is also sloppy because it's hard to know when your design is "good enough." When are you done? Since design is open-ended, the answer to that question is usually "When you're out of time."

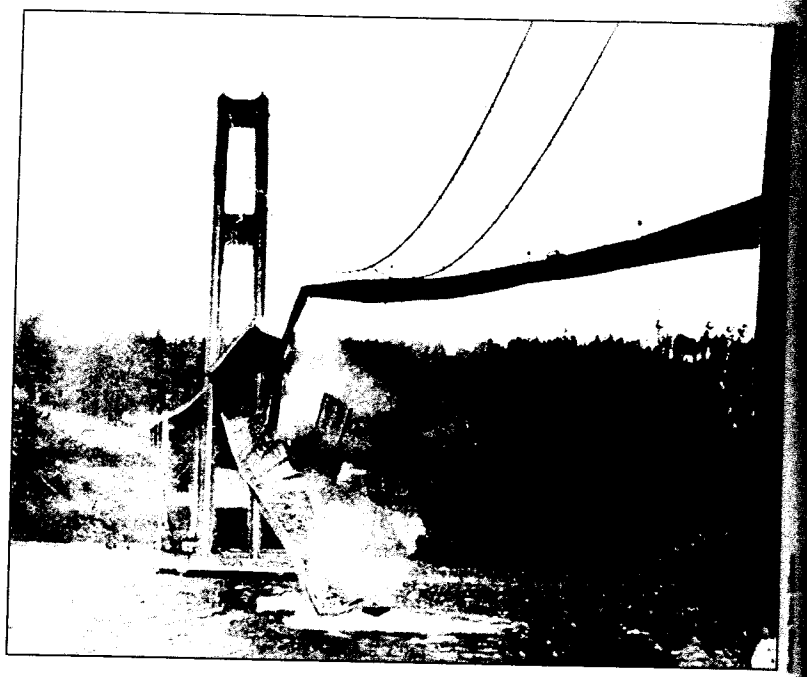
Design Is a Wicked Problem

Horst Rittel and Melvin Webber defined a "wicked" problem as one that could be clearly defined only by solving it, or by solving part of it (1973). This paradox implies, essentially, that you have to "solve" the problem once in order to clearly define it and then solve it again to create a solution that works. This process is almost motherhood and apple pie in software development.

In my part of the world, a dramatic example of such a wicked problem was the design of the original Tacoma Narrows bridge. At the time the bridge was built, the main consideration in designing a bridge was that it be strong

ıe picture of the ftware designer ıg his design in onal, error-free om a statement equirements is unrealistic. No ı has ever been ıed in that way, probably none r will. Even the program devel- ıents shown in ɔks and papers eal. They have en revised and ed until the au- has shown us ıt he wishes he lone, not what ly did happen.

vid Parnas and Paul Clements



The Tacoma Narrows bridge—an example of a wicked problem.

enough to support its planned load. In the case of the Tacoma Narrows bridge, wind created an unexpected, side-to-side harmonic ripple. One blustery day in 1940, the ripple grew uncontrollably until the bridge collapsed.

This is a good example of a wicked problem because until the bridge collapsed, its engineers didn't know that aerodynamics needed to be considered to such an extent. Only by building the bridge (solving the problem) could they learn about the additional consideration in the problem that allowed them to build another bridge that still stands.

One of the main differences between programs you develop in school and those you develop as a professional is that the design problems solved by school programs are rarely, if ever, wicked. Programming assignments in school are devised to move you in a beeline from beginning to end. You'd probably want to lynch a teacher who gave you a programming assignment, then changed the assignment as soon as you finished the design, and then changed it again just as you were about to turn in the completed program. But that very process is an everyday reality in professional programming.

Design Is a Heuristic Process

KEY POINT

A key to effective design is recognizing that it's a heuristic process. Design always involves some trial and error. The round-trip design concept accounts for the fact that design is heuristic by treating all design methodologies as tools in an intellectual toolbox. One tool works well on one job or on one phase or aspect of a job; other tools work well on others. No tool is right for everything, and it's useful to have several tools at your disposal.

When in doubt, use brute force.

Butler Lampson

One powerful heuristic tool is brute force. Don't underestimate it. A brute-force solution that works is better than an elegant solution that doesn't work. It can take a long time to get an elegant solution to work. In describing the history of searching algorithms, for example, Donald Knuth pointed out that even though the first description of a binary search algorithm was published in 1946, it took another 16 years for someone to publish an algorithm that correctly searched lists of all sizes (1973b).

Diagrams are another powerful heuristic tool. A picture is worth 1000 words—kind of. You actually want to leave out most of the 1000 words because one point of using a picture is that a picture can represent the problem at a higher level of abstraction. Sometimes you want to deal with the problem in detail, but other times you want to be able to work with it at a more general level.

More alarming, the same programmer is quite capable of doing the same task himself in two or three ways, sometimes unconsciously, but quite often simply for a change, or to provide elegant variation, or to find a way that will take less core or time.

A. R. Brown and W. A. Sampson

An additional aspect of the heuristic power of round-trip design is that you can leave some details unresolved during early design cycles. You don't have to decide everything at once. Remember that a point needs to be decided, but recognize that you don't yet have enough information to resolve that specific issue. Why fight your way through the last 10 percent of the design when it will drop into place easily the next time through? Why make bad decisions based on limited experience with the design when you can make good decisions based on more experience with it later? Some people are uncomfortable if they don't come to closure after a design cycle, but after you have created a few designs without resolving issues prematurely, it will seem natural to leave issues unresolved until you have more information (Zahniser 1992).

One of the most effective guidelines is not to get stuck on a single approach. I writing the program in PDL isn't working, make a picture. Write it in English Write a short test program. Try a completely different approach. Think of a brute-force solution. Keep outlining and sketching with your pencil, and you brain will follow. If all else fails, walk away from the problem. Literally go fo a walk, or think about something else before returning to the problem. I you've given it your best and are getting nowhere, putting it out of your mind for a time often produces results more quickly than sheer persistence can.