**74** Chapter 5: Design in Construction

# 5.1 Design Challenges

The phrase “software design” means the conception, invention, or contrivance of a scheme for turning a specification for computer software into operational software. Design is the activity that links requirements to coding and debugging. A good top- level design provides a structure that can safely contain multiple lower-level designs. Good design is useful on small projects and indispensable on large projects.

Design is also marked by numerous challenges, which are outlined in this section.

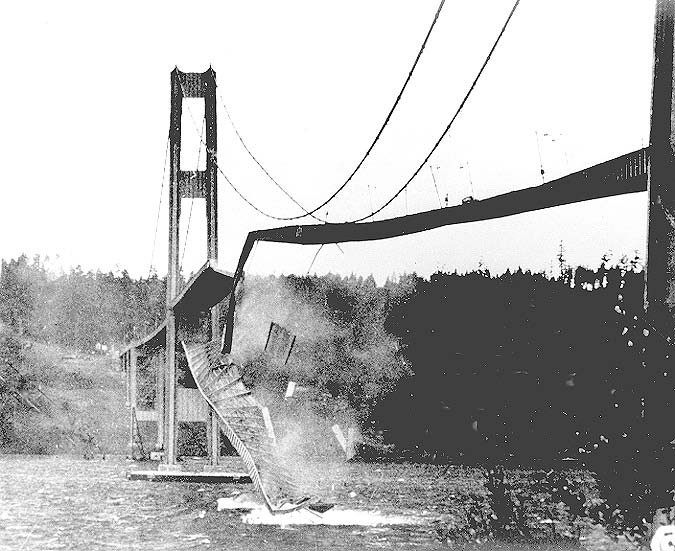
## Design Is a Wicked Problem

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 has been motherhood and apple pie in software development for decades (Peters and Tripp 1976).

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 con- sideration in designing a bridge was that it be strong 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, as shown in Figure 5-1.

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 addi- tional consideration in the problem that allowed them to build another bridge that still stands.

* 1. Design Challenges **75**



**Figure 5-1** The Tacoma Narrows bridge—an example of a wicked problem.

Morning News Tribune

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 tar and feather 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 pro- gram. But that very process is an everyday reality in professional programming.

## Design Is a Sloppy Process (Even If it Produces a Tidy Result)

The finished software design should look well organized and clean, but the process used to develop the design isn’t nearly as tidy as the end result.

Design is sloppy because you take many false steps and go down many blind alleys— you make a lot of mistakes. Indeed, making mistakes is the point of design—it’s cheaper to make mistakes and correct designs than it would be to make the same mis- takes, recognize them after coding, and have to correct full-blown code. Design is sloppy because a good solution is often only subtly different from a poor one.

Design is also sloppy because it’s hard to know when your design is “good enough.” How much detail is enough? How much design should be done with a formal design notation, and how much should be left to be done at the keyboard? When are you done? Since design is open-ended, the most common answer to that question is “When you’re out of time.”

## Design Is About Tradeoffs and Priorities

In an ideal world, every system could run instantly, consume zero storage space, use zero network bandwidth, never contain any errors, and cost nothing to build. In the real world, a key part of the designer’s job is to weigh competing design characteristics and strike a balance among those characteristics. If a fast response rate is more important than minimizing development time, a designer will choose one design. If minimizing development time is more important, a good designer will craft a different design.

## Design Involves Restrictions

The point of design is partly to create possibilities and partly to *restrict possibilities*. If people had infinite time, resources, and space to build physical structures, you would see incredible sprawling buildings with one room for each shoe and hundreds of rooms. This is how software can turn out without deliberately imposed restrictions. The con- straints of limited resources for constructing buildings force simplifications of the solu- tion that ultimately improve the solution. The goal in software design is the same.

## Design Is Nondeterministic

If you send three people away to design the same program, they can easily return with three vastly different designs, each of which could be perfectly acceptable. There might be more than one way to skin a cat, but there are usually dozens of ways to design a computer program.

## Design Is a Heuristic Process

Because design is nondeterministic, design techniques tend to be heuristics—“rules of thumb” or “things to try that sometimes work”—rather than repeatable processes that are guaranteed to produce predictable results. Design involves trial and error. A

design tool or technique that worked well on one job or on one aspect of a job might not work as well on the next project. No tool is right for everything.

## Design Is Emergent

A tidy way of summarizing these attributes of design is to say that design is “emergent.” Designs don’t spring fully formed directly from someone’s brain. They evolve and improve through design reviews, informal discussions, experience writing the code itself, and experience revising the code.

Virtually all systems undergo some degree of design changes during their initial devel- opment, and then they typically change to a greater extent as they’re extended into later versions. The degree to which change is beneficial or acceptable depends on the nature of the software being built.

# Key Design Concepts

Good design depends on understanding a handful of key concepts. This section dis- cusses the role of complexity, desirable characteristics of designs, and levels of design.

## Software’s Primary Technical Imperative: Managing Complexity

To understand the importance of managing complexity, it’s useful to refer to Fred Brooks’s landmark paper, “No Silver Bullets: Essence and Accidents of Software Engi- neering” (1987).

### Accidental and Essential Difficulties

Brooks argues that software development is made difficult because of two different classes of problems—the *essential* and the *accidental*. In referring to these two terms, Brooks draws on a philosophical tradition going back to Aristotle. In philosophy, the essential properties are the properties that a thing must have in order to be that thing. A car must have an engine, wheels, and doors to be a car. If it doesn’t have any of those essential properties, it isn’t really a car.

Accidental properties are the properties a thing just happens to have, properties that don’t really bear on whether the thing is what it is. A car could have a V8, a turbo- charged 4-cylinder, or some other kind of engine and be a car regardless of that detail. A car could have two doors or four; it could have skinny wheels or mag wheels. All those details are accidental properties. You could also think of accidental properties as *incidental*, *discretionary*, *optional*, and *happenstance*.

Brooks observes that the major accidental difficulties in software were addressed long ago. For example, accidental difficulties related to clumsy language syntaxes were largely eliminated in the evolution from assembly language to third-generation lan- guages and have declined in significance incrementally since then. Accidental difficul- ties related to noninteractive computers were resolved when time-share operating systems replaced batch-mode systems. Integrated programming environments fur- ther eliminated inefficiencies in programming work arising from tools that worked poorly together.

Brooks argues that progress on software’s remaining *essential* difficulties is bound to be slower. The reason is that, at its essence, software development consists of working out all the details of a highly intricate, interlocking set of concepts. The essential difficulties arise from the necessity of interfacing with the complex, disorderly real world; accurately and completely identifying the dependencies and exception cases; designing solutions that can’t be just approximately correct but that must be exactly correct; and so on. Even if we could invent a programming language that used the same terminology as the real-world problem we’re trying to solve, programming would still be difficult because of the challenge in determining precisely how the real world works. As software addresses ever-larger real-world problems, the interactions among the real-world entities become increasingly intricate, and that in turn increases the essential difficulty of the software solutions.

The root of all these essential difficulties is complexity—both accidental and essential.

### Importance of Managing Complexity

When software-project surveys report causes of project failure, they rarely identify technical reasons as the primary causes of project failure. Projects fail most often because of poor requirements, poor planning, or poor management. But when projects do fail for reasons that are primarily technical, the reason is often uncon- trolled complexity. The software is allowed to grow so complex that no one really knows what it does. When a project reaches the point at which no one completely understands the impact that code changes in one area will have on other areas, progress grinds to a halt.

Managing complexity is the most important technical topic in software development. In my view, it’s so important that Software’s Primary Technical Imperative has to be *managing complexity*.

Complexity is not a new feature of software development. Computing pioneer Edsger Dijkstra pointed out that computing is the only profession in which a single mind is obliged to span the distance from a bit to a few hundred megabytes, a ratio of 1 to 109, or nine orders of magnitude (Dijkstra 1989). This gigantic ratio is staggering. Dijkstra put it this way: “Compared to that number of semantic levels, the average mathemati- cal 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.” Of course software has become even more complex since 1989, and Dijkstra’s ratio of 1 to 109 could easily be more like 1 to 1015 today.

Dijkstra pointed out that no one’s skull is really big enough to contain a modern com- puter program (Dijkstra 1972), which means that we as software developers shouldn’t try to cram whole programs into our skulls at once; we should try to orga- nize our programs in such a way that we can safely focus on one part of it at a time. The goal is to minimize the amount of a program you have to think about at any one time. You might think of this as mental juggling—the more mental balls the program requires you to keep in the air at once, the more likely you’ll drop one of the balls, leading to a design or coding error.

At the software-architecture level, the complexity of a problem is reduced by dividing the system into subsystems. Humans have an easier time comprehending several sim- ple pieces of information than one complicated piece. The goal of all software-design techniques is to break a complicated problem into simple pieces. The more indepen- dent the subsystems are, the more you make it safe to focus on one bit of complexity at a time. Carefully defined objects separate concerns so that you can focus on one thing at a time. Packages provide the same benefit at a higher level of aggregation.

Keeping routines short helps reduce your mental workload. Writing programs in terms of the problem domain, rather than in terms of low-level implementation details, and working at the highest level of abstraction reduce the load on your brain.

The bottom line is that programmers who compensate for inherent human limita- tions write code that’s easier for themselves and others to understand and that has fewer errors.

### How to Attack Complexity

Overly costly, ineffective designs arise from three sources:

* A complex solution to a simple problem
* A simple, incorrect solution to a complex problem
* An inappropriate, complex solution to a complex problem

As Dijkstra pointed out, modern software is inherently complex, and no matter how hard you try, you’ll eventually bump into some level of complexity that’s inherent in the real-world problem itself. This suggests a two-prong approach to managing complexity:

* Minimize the amount of essential complexity that anyone’s brain has to deal with at any one time.
* Keep accidental complexity from needlessly proliferating.

Once you understand that all other technical goals in software are secondary to man- aging complexity, many design considerations become straightforward.

## Desirable Characteristics of a Design

A high-quality design has several general characteristics. If you could achieve all these goals, your design would be very good indeed. Some goals contradict other goals, but that’s the challenge of design—creating a good set of tradeoffs from competing objectives. Some characteristics of design quality are also characteristics of a good program: reliability, performance, and so on. Others are internal characteristics of the design.

Here’s a list of internal design characteristics:

***Minimal complexity*** The primary goal of design should be to minimize complexity for all the reasons just described. Avoid making “clever” designs. Clever designs are usually hard to understand. Instead make “simple” and “easy-to-understand” designs. If your design doesn’t let you safely ignore most other parts of the program when you’re immersed in one specific part, the design isn’t doing its job.

***Ease of maintenance*** Ease of maintenance means designing for the maintenance programmer. Continually imagine the questions a maintenance programmer would ask about the code you’re writing. Think of the maintenance programmer as your audience, and then design the system to be self-explanatory.

***Loose coupling*** Loose coupling means designing so that you hold connections among different parts of a program to a minimum. Use the principles of good abstrac- tions in class interfaces, encapsulation, and information hiding to design classes with as few interconnections as possible. Minimal connectedness minimizes work during integration, testing, and maintenance.

***Extensibility*** Extensibility means that you can enhance a system without causing violence to the underlying structure. You can change a piece of a system without affecting other pieces. The most likely changes cause the system the least trauma.

***Reusability*** Reusability means designing the system so that you can reuse pieces of it in other systems.

***High fan-in*** High fan-in refers to having a high number of classes that use a given class. High fan-in implies that a system has been designed to make good use of utility classes at the lower levels in the system.

* 1. Key Design Concepts **81**

## Levels of Design 20k

Design is needed at several different levels of detail in a software system. Some design tech- niques apply at all levels, and some apply at only one or two. Figure 5-2 illustrates the levels.

**Figure 5-2** The levels of design in a program. The system (1) is first organized into sub- systems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5).

### Level 1: Software System

The first level is the entire system. Some programmers jump right from the system level into designing classes, but it’s usually beneficial to think through higher level combinations of classes, such as subsystems or packages.

### Level 2: Division into Subsystems or Packages

The main product of design at this level is the identification of all major subsystems. The subsystems can be big: database, user interface, business rules, command interpreter,

report engine, and so on. The major design activity at this level is deciding how to parti- tion the program into major subsystems and defining how each subsystem is allowed to use each other subsystem. Division at this level is typically needed on any project that takes longer than a few weeks. Within each subsystem, different methods of design might be used—choosing the approach that best fits each part of the system. In Figure 5- 2, design at this level is marked with a *2*.

Of particular importance at this level are the rules about how the various subsystems can communicate. If all subsystems can communicate with all other subsystems, you lose the benefit of separating them at all. Make each subsystem meaningful by restrict- ing communications.

Suppose for example that you define a system with six subsystems, as shown in Fig- ure 5-3. When there are no rules, the second law of thermodynamics will come into play and the entropy of the system will increase. One way in which entropy increases is that, without any restrictions on communications among subsystems, communica- tion will occur in an unrestricted way, as in Figure 5-4.

**Figure 5-3** An example of a system with six subsystems.

**Figure 5-4** An example of what happens with no restrictions on intersubsystem communications.

As you can see, every subsystem ends up communicating directly with every other subsystem, which raises some important questions:

* + How many different parts of the system does a developer need to understand at least a little bit to change something in the graphics subsystem?
  + What happens when you try to use the business rules in another system?
  + What happens when you want to put a new user interface on the system, per- haps a command-line UI for test purposes?
  + What happens when you want to put data storage on a remote machine?

You might think of the lines between subsystems as being hoses with water running through them. If you want to reach in and pull out a subsystem, that subsystem is going to have some hoses attached to it. The more hoses you have to disconnect and reconnect, the more wet you’re going to get. You want to architect your system so that if you pull out a subsystem to use elsewhere, you won’t have many hoses to reconnect and those hoses will reconnect easily.

With forethought, all of these issues can be addressed with little extra work. Allow communication between subsystems only on a “need to know” basis—and it had bet- ter be a *good* reason. If in doubt, it’s easier to restrict communication early and relax it later than it is to relax it early and then try to tighten it up after you’ve coded several hundred intersubsystem calls. Figure 5-5 shows how a few communication guidelines could change the system depicted in Figure 5-4.

**Figure 5-5** With a few communication rules, you can simplify subsystem interactions sig- nificantly.

To keep the connections easy to understand and maintain, err on the side of simple intersubsystem relations. The simplest relationship is to have one subsystem call rou- tines in another. A more involved relationship is to have one subsystem contain classes from another. The most involved relationship is to have classes in one sub- system inherit from classes in another.

A good general rule is that a system-level diagram like Figure 5-5 should be an acyclic graph. In other words, a program shouldn’t contain any circular relationships in which Class A uses Class B, Class B uses Class C, and Class C uses Class A.

On large programs and families of programs, design at the subsystem level makes a difference. If you believe that your program is small enough to skip subsystem-level design, at least make the decision to skip that level of design a conscious one.

**Common Subsystems** Some kinds of subsystems appear again and again in different systems. Here are some of the usual suspects.

***Business rules*** Business rules are the laws, regulations, policies, and procedures that you encode into a computer system. If you’re writing a payroll system, you might encode rules from the IRS about the number of allowable withholdings and the estimated tax rate. Additional rules for a payroll system might come from a union contract specifying overtime rates, vacation and holiday pay, and so on. If you’re writing a program to quote automobile insurance rates, rules might come from government regulations on required liability coverages, actuarial rate tables, or underwriting restrictions

***User interface*** Create a subsystem to isolate user-interface components so that the user interface can evolve without damaging the rest of the program. In most cases, a user-interface subsystem uses several subordinate subsystems or classes for the GUI interface, command line interface, menu operations, window management, help sys- tem, and so forth.

***Database access*** You can hide the implementation details of accessing a database so that most of the program doesn’t need to worry about the messy details of manipulat- ing low-level structures and can deal with the data in terms of how it’s used at the business-problem level. Subsystems that hide implementation details provide a valu- able level of abstraction that reduces a program’s complexity. They centralize data- base operations in one place and reduce the chance of errors in working with the data. They make it easy to change the database design structure without changing most of the program.

***System dependencies*** Package operating-system dependencies into a subsystem for the same reason you package hardware dependencies. If you’re developing a pro- gram for Microsoft Windows, for example, why limit yourself to the Windows envi- ronment? Isolate the Windows calls in a Windows-interface subsystem. If you later want to move your program to Mac OS or Linux, all you’ll have to change is the interface subsystem. An interface subsystem can be too extensive for you to imple- ment on your own, but such subsystems are readily available in any of several com- mercial code libraries.

### Level 3: Division into Classes

Design at this level includes identifying all classes in the system. For example, a data- base-interface subsystem might be further partitioned into data access classes and persistence framework classes as well as database metadata. Figure 5-2, Level 3, shows how one of Level 2’s subsystems might be divided into classes, and it implies that the other three subsystems shown at Level 2 are also decomposed into classes.

Details of the ways in which each class interacts with the rest of the system are also specified as the classes are specified. In particular, the class’s interface is defined. Overall, the major design activity at this level is making sure that all the subsystems have been decomposed to a level of detail fine enough that you can implement their parts as individual classes.

The division of subsystems into classes is typically needed on any project that takes longer than a few days. If the project is large, the division is clearly distinct from the program partitioning of Level 2. If the project is very small, you might move directly from the whole-system view of Level 1 to the classes view of Level 3.

**Classes vs. Objects** A key concept in object-oriented design is the differentiation between objects and classes. An object is any specific entity that exists in your program at run time. A class is the static thing you look at in the program listing. An object is the dynamic thing with specific values and attributes you see when you run the program. For example, you could declare a class *Person* that had attributes of name, age, gender, and so on. At run time you would have the objects *nancy*, *hank*, *diane*, *tony*, and so on—that is, specific instances of the class. If you’re familiar with database terms, it’s the same as the distinction between “schema” and “instance.” You could think of the class as the cookie cutter and the object as the cookie. This book uses the terms informally and generally refers to classes and objects more or less inter- changeably.