1

# A Dialogue on the Book

**Professor:** *Welcome to this book! It's called* ***Operating Systems in Three Easy Pieces****, and I am here to teach you the things you need to know about operating systems. I am called "Professor"; who are you?*

**Student:** *Hi Professor! I am called "Student", as you might have guessed. And I am here and ready to learn!*

**Professor:** *Sounds good. Any questions?*

**Student:** *Sure! Why is it called "Three Easy Pieces"?*

**Professor:** *That's an easy one. Well, you see, there are these great lectures on Physics by Richard Feynman...*

**Student:** *Oh! The guy who wrote "Surely You're Joking, Mr. Feynman", right? Great book! Is this going to be hilarious like that book was?*

**Professor:** *Um... well, no. That book was great, and I'm glad you've read it. Hopefully this book is more like his notes on Physics. Some of the basics were summed up in a book called "Six Easy Pieces". He was talking about Physics; we're going to do Three Easy Pieces on the fine topic of Operating Systems. This is appropriate, as Operating Systems are about half as hard as Physics.*

**Student:** *Well, I liked physics, so that is probably good. What are those pieces?*

**Professor:** *They are the three key ideas we're going to learn about:* ***virtualization****,* ***concurrency****, and* ***persistence****. In learning about these ideas, we'll learn all about how an operating system works, including how it decides what program to run next on a CPU, how it handles memory overload in a virtual memory system, how virtual machine monitors work, how to manage information on disks, and even a little about how to build a distributed system that works when parts have failed. That sort of stuff.*

**Student:** *I have no idea what you're talking about, really.*

**Professor:** *Good! That means you are in the right class.*

**Student:** *I have another question: what's the best way to learn this stuff?*

**Professor:** *Excellent query! Well, each person needs to figure this out on their* *own, of course, but here is what I would do: go to class, to hear the professor introduce the material. Then, at the end of every week, read these notes, to help the ideas sink into your head a bit better. Of course, some time later (hint: before the exam!), read the notes again to firm up your knowledge. Of course, your professor will no doubt assign some homeworks and projects, so you should do those; in particular, doing projects where you write real code to solve real problems is the best way to put the ideas within these notes into action. As Confucius said...*

**Student:** *Oh, I know! 'I hear and I forget. I see and I remember. I do and I understand.' Or something like that.*

**Professor:** *(surprised) How did you know what I was going to say?!*

**Student:** *It seemed to follow. Also, I am a big fan of Confucius, and an even bigger fan of Xunzi, who actually is a better source for this quote*[1].

**Professor:** *(stunned) Well, I think we are going to get along just fine! Just fine indeed.*

**Student:** *Professor – just one more question, if I may. What are these dialogues for? I mean, isn't this just supposed to be a book? Why not present the material directly?*

**Professor:** *Ah, good question, good question! Well, I think it is sometimes useful to pull yourself outside of a narrative and think a bit; these dialogues are those times. So you and I are going to work together to make sense of all of these pretty complex ideas. Are you up for it?*

**Student:** *So we have to think? Well, I'm up for that. I mean, what else do I have to do anyhow? It's not like I have much of a life outside of this book.*

**Professor:** *Me neither, sadly. So let's get to work!*

[1] According to this website (http://www.barrypopik.com/index.php/new_york_city/entry/tell_me_and_i_forget_teach_me_and_i_may_remember_involve_me_and_i_will_lear/), Confucian philosopher Xunzi said "Not having heard something is not as good as having heard it; having heard it is not as good as having seen it; having seen it is not as good as knowing it; knowing it is not as good as putting it into practice." Later on, the wisdom got attached to Confucius for some reason. Thanks to Jiao Dong (Rutgers) for telling us!

# 2

# Introduction to Operating Systems

If you are taking an undergraduate operating systems course, you should already have some idea of what a computer program does when it runs. If not, this book (and the corresponding course) is going to be difficult — so you should probably stop reading this book, or run to the nearest bookstore and quickly consume the necessary background material before continuing (both Patt/Patel [PP03] and particularly Bryant/O'Hallaron [BOH10] are pretty great books).

So what happens when a program runs?

Well, a running program does one very simple thing: it executes instructions. Many millions (and these days, even billions) of times every second, the processor **fetches** an instruction from memory, **decodes** it (i.e., figures out which instruction this is), and **executes** it (i.e., it does the thing that it is supposed to do, like add two numbers together, access memory, check a condition, jump to a function, and so forth). After it is done with this instruction, the processor moves on to the next instruction, and so on, and so on, until the program finally completes[1].

Thus, we have just described the basics of the **Von Neumann** model of computing[2]. Sounds simple, right? But in this class, we will be learning that while a program runs, a lot of other wild things are going on with the primary goal of making the system **easy to use**.

There is a body of software, in fact, that is responsible for making it easy to run programs (even allowing you to seemingly run many at the same time), allowing programs to share memory, enabling programs to interact with devices, and other fun stuff like that. That body of software

[1]Of course, modern processors do many bizarre and frightening things underneath the hood to make programs run faster, e.g., executing multiple instructions at once, and even issuing and completing them out of order! But that is not our concern here; we are just concerned with the simple model most programs assume: that instructions seemingly execute one at a time, in an orderly and sequential fashion.

[2]Von Neumann was one of the early pioneers of computing systems. He also did pioneering work on game theory and atomic bombs, and played in the NBA for six years. OK, one of those things isn't true.

THE CRUX OF THE PROBLEM:
HOW TO VIRTUALIZE RESOURCES

One central question we will answer in this book is quite simple: how does the operating system virtualize resources? This is the crux of our problem. *Why* the OS does this is not the main question, as the answer should be obvious: it makes the system easier to use. Thus, we focus on the *how*: what mechanisms and policies are implemented by the OS to attain virtualization? How does the OS do so efficiently? What hardware support is needed?

We will use the "crux of the problem", in shaded boxes such as this one, as a way to call out specific problems we are trying to solve in building an operating system. Thus, within a note on a particular topic, you may find one or more *cruces* (yes, this is the proper plural) which highlight the problem. The details within the chapter, of course, present the solution, or at least the basic parameters of a solution.

is called the **operating system** (**OS**)[3], as it is in charge of making sure the system operates correctly and efficiently in an easy-to-use manner.

The primary way the OS does this is through a general technique that we call **virtualization**. That is, the OS takes a **physical** resource (such as the processor, or memory, or a disk) and transforms it into a more general, powerful, and easy-to-use **virtual** form of itself. Thus, we sometimes refer to the operating system as a **virtual machine**.

Of course, in order to allow users to tell the OS what to do and thus make use of the features of the virtual machine (such as running a program, or allocating memory, or accessing a file), the OS also provides some interfaces (APIs) that you can call. A typical OS, in fact, exports a few hundred **system calls** that are available to applications. Because the OS provides these calls to run programs, access memory and devices, and other related actions, we also sometimes say that the OS provides a **standard library** to applications.

Finally, because virtualization allows many programs to run (thus sharing the CPU), and many programs to concurrently access their own instructions and data (thus sharing memory), and many programs to access devices (thus sharing disks and so forth), the OS is sometimes known as a **resource manager**. Each of the CPU, memory, and disk is a **resource** of the system; it is thus the operating system's role to **manage** those resources, doing so efficiently or fairly or indeed with many other possible goals in mind. To understand the role of the OS a little bit better, let's take a look at some examples.

[3]Another early name for the OS was the **supervisor** or even the **master control program**. Apparently, the latter sounded a little overzealous (see the movie Tron for details) and thus, thankfully, "operating system" caught on instead.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];
    while (1) {
        Spin(1);
        printf("%s\n", str);
    }
    return 0;
}
```

Figure 2.1: **Simple Example: Code That Loops and Prints (cpu.c)**

## 2.1 Virtualizing the CPU

Figure 2.1 depicts our first program. It doesn't do much. In fact, all it does is call `Spin()`, a function that repeatedly checks the time and returns once it has run for a second. Then, it prints out the string that the user passed in on the command line, and repeats, forever.

Let's say we save this file as `cpu.c` and decide to compile and run it on a system with a single processor (or **CPU** as we will sometimes call it). Here is what we will see:

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

Not too interesting of a run — the system begins running the program, which repeatedly checks the time until a second has elapsed. Once a second has passed, the code prints the input string passed in by the user (in this example, the letter "A"), and continues. Note the program will run forever; only by pressing "Control-c" (which on UNIX-based systems will terminate the program running in the foreground) can we halt the program.

Now, let's do the same thing, but this time, let's run many different instances of this same program. Figure 2.2 shows the results of this slightly more complicated example.

```
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...
```

Figure 2.2: **Running Many Programs At Once**

Well, now things are getting a little more interesting. Even though we have only one processor, somehow all four of these programs seem to be running at the same time! How does this magic happen?[4]

It turns out that the operating system, with some help from the hardware, is in charge of this **illusion**, i.e., the illusion that the system has a very large number of virtual CPUs. Turning a single CPU (or small set of them) into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once is what we call **virtualizing the CPU**, the focus of the first major part of this book.

Of course, to run programs, and stop them, and otherwise tell the OS which programs to run, there need to be some interfaces (APIs) that you can use to communicate your desires to the OS. We'll talk about these APIs throughout this book; indeed, they are the major way in which most users interact with operating systems.

You might also notice that the ability to run multiple programs at once raises all sorts of new questions. For example, if two programs want to run at a particular time, which *should* run? This question is answered by a **policy** of the OS; policies are used in many different places within an OS to answer these types of questions, and thus we will study them as we learn about the basic **mechanisms** that operating systems implement (such as the ability to run multiple programs at once). Hence the role of the OS as a **resource manager**.

[4]Note how we ran four processes at the same time, by using the `&` symbol. Doing so runs a job in the background in the `tcsh` shell, which means that the user is able to immediately issue their next command, which in this case is another program to run. The semi-colon between commands allows us to run multiple programs at the same time in `tcsh`. If you're using a different shell (e.g., `bash`), it works slightly differently; read documentation online for details.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    int *p = malloc(sizeof(int));                    // a1
    assert(p != NULL);
    printf("(%d) address pointed to by p: %p\n",
           getpid(), p);                             // a2
    *p = 0;                                          // a3
    while (1) {
        Spin(1);
        *p = *p + 1;
        printf("(%d) p: %d\n", getpid(), *p);        // a4
    }
    return 0;
}
```

Figure 2.3: **A Program that Accesses Memory (`mem.c`)**

## 2.2 Virtualizing Memory

Now let's consider memory. The model of **physical memory** presented by modern machines is very simple. Memory is just an array of bytes; to **read** memory, one must specify an **address** to be able to access the data stored there; to **write** (or **update**) memory, one must also specify the data to be written to the given address.

Memory is accessed all the time when a program is running. A program keeps all of its data structures in memory, and accesses them through various instructions, like loads and stores or other explicit instructions that access memory in doing their work. Don't forget that each instruction of the program is in memory too; thus memory is accessed on each instruction fetch.

Let's take a look at a program (in Figure 2.3) that allocates some memory by calling `malloc()`. The output of this program can be found here:

```
prompt> ./mem
(2134) memory address of p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

The program does a couple of things. First, it allocates some memory (line a1). Then, it prints out the address of the memory (a2), and then puts the number zero into the first slot of the newly allocated memory (a3). Finally, it loops, delaying for a second and incrementing the value stored at the address held in `p`. With every print statement, it also prints out what is called the process identifier (the PID) of the running program. This PID is unique per running process.

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) memory address of p: 0x200000
(24114) memory address of p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

Figure 2.4: **Running The Memory Program Multiple Times**

Again, this first result is not too interesting. The newly allocated memory is at address `0x200000`. As the program runs, it slowly updates the value and prints out the result.

Now, we again run multiple instances of this same program to see what happens (Figure 2.4). We see from the example that each running program has allocated memory at the same address (`0x200000`), and yet each seems to be updating the value at `0x200000` independently! It is as if each running program has its own private memory, instead of sharing the same physical memory with other running programs[5].

Indeed, that is exactly what is happening here as the OS is **virtualizing memory**. Each process accesses its own private **virtual address space** (sometimes just called its **address space**), which the OS somehow maps onto the physical memory of the machine. A memory reference within one running program does not affect the address space of other processes (or the OS itself); as far as the running program is concerned, it has physical memory all to itself. The reality, however, is that physical memory is a shared resource, managed by the operating system. Exactly how all of this is accomplished is also the subject of the first part of this book, on the topic of **virtualization**.

## 2.3 Concurrency

Another main theme of this book is **concurrency**. We use this conceptual term to refer to a host of problems that arise, and must be addressed, when working on many things at once (i.e., concurrently) in the same program. The problems of concurrency arose first within the operating system itself; as you can see in the examples above on virtualization, the OS is juggling many things at once, first running one process, then another, and so forth. As it turns out, doing so leads to some deep and interesting problems.

[5]For this example to work, you need to make sure address-space randomization is disabled; randomization, as it turns out, can be a good defense against certain kinds of security flaws. Read more about it on your own, especially if you want to learn how to break into computer systems via stack-smashing attacks. Not that we would recommend such a thing...

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

Figure 2.5: **A Multi-threaded Program (`threads.c`)**

Unfortunately, the problems of concurrency are no longer limited just to the OS itself. Indeed, modern **multi-threaded** programs exhibit the same problems. Let us demonstrate with an example of a **multi-threaded** program (Figure 2.5).

Although you might not understand this example fully at the moment (and we'll learn a lot more about it in later chapters, in the section of the book on concurrency), the basic idea is simple. The main program creates two **threads** using `Pthread_create()`[6]. You can think of a thread as a function running within the same memory space as other functions, with more than one of them active at a time. In this example, each thread starts running in a routine called `worker()`, in which it simply increments a counter in a loop for `loops` number of times.

Below is a transcript of what happens when we run this program with the input value for the variable `loops` set to 1000. The value of `loops`

[6]The actual call should be to lower-case `pthread_create()`; the upper-case version is our own wrapper that calls `pthread_create()` and makes sure that the return code indicates that the call succeeded. See the code for details.

THE CRUX OF THE PROBLEM:
HOW TO BUILD CORRECT CONCURRENT PROGRAMS

When there are many concurrently executing threads within the same memory space, how can we build a correctly working program? What primitives are needed from the OS? What mechanisms should be provided by the hardware? How can we use them to solve the problems of concurrency?

determines how many times each of the two workers will increment the shared counter in a loop. When the program is run with the value of `loops` set to 1000, what do you expect the final value of `counter` to be?

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000
```

As you probably guessed, when the two threads are finished, the final value of the counter is 2000, as each thread incremented the counter 1000 times. Indeed, when the input value of `loops` is set to $N$, we would expect the final output of the program to be $2N$. But life is not so simple, as it turns out. Let's run the same program, but with higher values for `loops`, and see what happens:

```
prompt> ./thread 100000
Initial value : 0
Final value   : 143012    // huh??
prompt> ./thread 100000
Initial value : 0
Final value   : 137298    // what the??
```

In this run, when we gave an input value of 100,000, instead of getting a final value of 200,000, we instead first get 143,012. Then, when we run the program a second time, we not only again get the *wrong* value, but also a *different* value than the last time. In fact, if you run the program over and over with high values of `loops`, you may find that sometimes you even get the right answer! So why is this happening?

As it turns out, the reason for these odd and unusual outcomes relate to how instructions are executed, which is one at a time. Unfortunately, a key part of the program above, where the shared counter is incremented, takes three instructions: one to load the value of the counter from memory into a register, one to increment it, and one to store it back into memory. Because these three instructions do not execute **atomically** (all at once), strange things can happen. It is this problem of **concurrency** that we will address in great detail in the second part of this book.

```c
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/types.h>

int
main(int argc, char *argv[])
{
    int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
    assert(fd > -1);
    int rc = write(fd, "hello world\n", 13);
    assert(rc == 13);
    close(fd);
    return 0;
}
```

Figure 2.6: **A Program That Does I/O (`io.c`)**

## 2.4 Persistence

The third major theme of the course is **persistence**. In system memory, data can be easily lost, as devices such as DRAM store values in a **volatile** manner; when power goes away or the system crashes, any data in memory is lost. Thus, we need hardware and software to be able to store data **persistently**; such storage is thus critical to any system as users care a great deal about their data.

The hardware comes in the form of some kind of **input/output** or **I/O** device; in modern systems, a **hard drive** is a common repository for long-lived information, although **solid-state drives** (**SSDs**) are making headway in this arena as well.

The software in the operating system that usually manages the disk is called the **file system**; it is thus responsible for storing any **files** the user creates in a reliable and efficient manner on the disks of the system.

Unlike the abstractions provided by the OS for the CPU and memory, the OS does not create a private, virtualized disk for each application. Rather, it is assumed that often times, users will want to **share** information that is in files. For example, when writing a C program, you might first use an editor (e.g., Emacs[7]) to create and edit the C file (`emacs -nw main.c`). Once done, you might use the compiler to turn the source code into an executable (e.g., `gcc -o main main.c`). When you're finished, you might run the new executable (e.g., `./main`). Thus, you can see how files are shared across different processes. First, Emacs creates a file that serves as input to the compiler; the compiler uses that input file to create a new executable file (in many steps — take a compiler course for details); finally, the new executable is then run. And thus a new program is born!

To understand this better, let's look at some code. Figure 2.6 presents code to create a file (`/tmp/file`) that contains the string "hello world".

[7]You should be using Emacs. If you are using vi, there is probably something wrong with you. If you are using something that is not a real code editor, that is even worse.

THE CRUX OF THE PROBLEM:
HOW TO STORE DATA PERSISTENTLY

The file system is the part of the OS in charge of managing persistent data. What techniques are needed to do so correctly? What mechanisms and policies are required to do so with high performance? How is reliability achieved, in the face of failures in hardware and software?

To accomplish this task, the program makes three calls into the operating system. The first, a call to `open()`, opens the file and creates it; the second, `write()`, writes some data to the file; the third, `close()`, simply closes the file thus indicating the program won't be writing any more data to it. These **system calls** are routed to the part of the operating system called the **file system**, which then handles the requests and returns some kind of error code to the user.

You might be wondering what the OS does in order to actually write to disk. We would show you but you'd have to promise to close your eyes first; it is that unpleasant. The file system has to do a fair bit of work: first figuring out where on disk this new data will reside, and then keeping track of it in various structures the file system maintains. Doing so requires issuing I/O requests to the underlying storage device, to either read existing structures or update (write) them. As anyone who has written a **device driver**[8] knows, getting a device to do something on your behalf is an intricate and detailed process. It requires a deep knowledge of the low-level device interface and its exact semantics. Fortunately, the OS provides a standard and simple way to access devices through its system calls. Thus, the OS is sometimes seen as a **standard library**.

Of course, there are many more details in how devices are accessed, and how file systems manage data persistently atop said devices. For performance reasons, most file systems first delay such writes for a while, hoping to batch them into larger groups. To handle the problems of system crashes during writes, most file systems incorporate some kind of intricate write protocol, such as **journaling** or **copy-on-write**, carefully ordering writes to disk to ensure that if a failure occurs during the write sequence, the system can recover to reasonable state afterwards. To make different common operations efficient, file systems employ many different data structures and access methods, from simple lists to complex b-trees. If all of this doesn't make sense yet, good! We'll be talking about all of this quite a bit more in the third part of this book on **persistence**, where we'll discuss devices and I/O in general, and then disks, RAIDs, and file systems in great detail.

[8]A device driver is some code in the operating system that knows how to deal with a specific device. We will talk more about devices and device drivers later.

## 2.5 Design Goals

So now you have some idea of what an OS actually does: it takes physical **resources**, such as a CPU, memory, or disk, and **virtualizes** them. It handles tough and tricky issues related to **concurrency**. And it stores files **persistently**, thus making them safe over the long-term. Given that we want to build such a system, we want to have some goals in mind to help focus our design and implementation and make trade-offs as necessary; finding the right set of trade-offs is a key to building systems.

One of the most basic goals is to build up some **abstractions** in order to make the system convenient and easy to use. Abstractions are fundamental to everything we do in computer science. Abstraction makes it possible to write a large program by dividing it into small and understandable pieces, to write such a program in a high-level language like C[9] without thinking about assembly, to write code in assembly without thinking about logic gates, and to build a processor out of gates without thinking too much about transistors. Abstraction is so fundamental that sometimes we forget its importance, but we won't here; thus, in each section, we'll discuss some of the major abstractions that have developed over time, giving you a way to think about pieces of the OS.

One goal in designing and implementing an operating system is to provide high **performance**; another way to say this is our goal is to **minimize the overheads** of the OS. Virtualization and making the system easy to use are well worth it, but not at any cost; thus, we must strive to provide virtualization and other OS features without excessive overheads. These overheads arise in a number of forms: extra time (more instructions) and extra space (in memory or on disk). We'll seek solutions that minimize one or the other or both, if possible. Perfection, however, is not always attainable, something we will learn to notice and (where appropriate) tolerate.

Another goal will be to provide **protection** between applications, as well as between the OS and applications. Because we wish to allow many programs to run at the same time, we want to make sure that the malicious or accidental bad behavior of one does not harm others; we certainly don't want an application to be able to harm the OS itself (as that would affect *all* programs running on the system). Protection is at the heart of one of the main principles underlying an operating system, which is that of **isolation**; isolating processes from one another is the key to protection and thus underlies much of what an OS must do.

The operating system must also run non-stop; when it fails, *all* applications running on the system fail as well. Because of this dependence, operating systems often strive to provide a high degree of **reliability**. As operating systems grow evermore complex (sometimes containing millions of lines of code), building a reliable operating system is quite a chal-

[9]Some of you might object to calling C a high-level language. Remember this is an OS course, though, where we're simply happy not to have to code in assembly all the time!

lenge — and indeed, much of the on-going research in the field (including some of our own work [BS+09, SS+10]) focuses on this exact problem.

Other goals make sense: **energy-efficiency** is important in our increasingly green world; **security** (an extension of protection, really) against malicious applications is critical, especially in these highly-networked times; **mobility** is increasingly important as OSes are run on smaller and smaller devices. Depending on how the system is used, the OS will have different goals and thus likely be implemented in at least slightly different ways. However, as we will see, many of the principles we will present on how to build an OS are useful on a range of different devices.

## 2.6 Some History

Before closing this introduction, let us present a brief history of how operating systems developed. Like any system built by humans, good ideas accumulated in operating systems over time, as engineers learned what was important in their design. Here, we discuss a few major developments. For a richer treatment, see Brinch Hansen's excellent history of operating systems [BH00].

### Early Operating Systems: Just Libraries

In the beginning, the operating system didn't do too much. Basically, it was just a set of libraries of commonly-used functions; for example, instead of having each programmer of the system write low-level I/O handling code, the "OS" would provide such APIs, and thus make life easier for the developer.

Usually, on these old mainframe systems, one program ran at a time, as controlled by a human operator. Much of what you think a modern OS would do (e.g., deciding what order to run jobs in) was performed by this operator. If you were a smart developer, you would be nice to this operator, so that they might move your job to the front of the queue.

This mode of computing was known as **batch** processing, as a number of jobs were set up and then run in a "batch" by the operator. Computers, as of that point, were not used in an interactive manner, because of cost: it was simply too expensive to let a user sit in front of the computer and use it, as most of the time it would just sit idle then, costing the facility hundreds of thousands of dollars per hour [BH00].

### Beyond Libraries: Protection

In moving beyond being a simple library of commonly-used services, operating systems took on a more central role in managing machines. One important aspect of this was the realization that code run on behalf of the OS was special; it had control of devices and thus should be treated differently than normal application code. Why is this? Well, imagine if you allowed any application to read from anywhere on the disk; the notion of privacy goes out the window, as any program could read any file. Thus, implementing a **file system** (to manage your files) as a library makes little sense. Instead, something else was needed.

Thus, the idea of a **system call** was invented, pioneered by the Atlas computing system [K+61,L78]. Instead of providing OS routines as a library (where you just make a **procedure call** to access them), the idea here was to add a special pair of hardware instructions and hardware state to make the transition into the OS a more formal, controlled process.

The key difference between a system call and a procedure call is that a system call transfers control (i.e., jumps) into the OS while simultaneously raising the **hardware privilege level**. User applications run in what is referred to as **user mode** which means the hardware restricts what applications can do; for example, an application running in user mode can't typically initiate an I/O request to the disk, access any physical memory page, or send a packet on the network. When a system call is initiated (usually through a special hardware instruction called a **trap**), the hardware transfers control to a pre-specified **trap handler** (that the OS set up previously) and simultaneously raises the privilege level to **kernel mode**. In kernel mode, the OS has full access to the hardware of the system and thus can do things like initiate an I/O request or make more memory available to a program. When the OS is done servicing the request, it passes control back to the user via a special **return-from-trap** instruction, which reverts to user mode while simultaneously passing control back to where the application left off.

## The Era of Multiprogramming

Where operating systems really took off was in the era of computing beyond the mainframe, that of the **minicomputer**. Classic machines like the PDP family from Digital Equipment made computers hugely more affordable; thus, instead of having one mainframe per large organization, now a smaller collection of people within an organization could likely have their own computer. Not surprisingly, one of the major impacts of this drop in cost was an increase in developer activity; more smart people got their hands on computers and thus made computer systems do more interesting and beautiful things.

In particular, **multiprogramming** became commonplace due to the desire to make better use of machine resources. Instead of just running one job at a time, the OS would load a number of jobs into memory and switch rapidly between them, thus improving CPU utilization. This switching was particularly important because I/O devices were slow; having a program wait on the CPU while its I/O was being serviced was a waste of CPU time. Instead, why not switch to another job and run it for a while?

The desire to support multiprogramming and overlap in the presence of I/O and interrupts forced innovation in the conceptual development of operating systems along a number of directions. Issues such as **memory**

**protection** became important; we wouldn't want one program to be able to access the memory of another program. Understanding how to deal with the **concurrency** issues introduced by multiprogramming was also critical; making sure the OS was behaving correctly despite the presence of interrupts is a great challenge. We will study these issues and related topics later in the book.

One of the major practical advances of the time was the introduction of the UNIX operating system, primarily thanks to Ken Thompson (and Dennis Ritchie) at Bell Labs (yes, the phone company). UNIX took many good ideas from different operating systems (particularly from Multics [O72], and some from systems like TENEX [B+72] and the Berkeley Time-Sharing System [S+68]), but made them simpler and easier to use. Soon this team was shipping tapes containing UNIX source code to people around the world, many of whom then got involved and added to the system themselves; see the **Aside** (next page) for more detail[10].

## The Modern Era

Beyond the minicomputer came a new type of machine, cheaper, faster, and for the masses: the **personal computer**, or **PC** as we call it today. Led by Apple's early machines (e.g., the Apple II) and the IBM PC, this new breed of machine would soon become the dominant force in computing, as their low-cost enabled one machine per desktop instead of a shared minicomputer per workgroup.

Unfortunately, for operating systems, the PC at first represented a great leap backwards, as early systems forgot (or never knew of) the lessons learned in the era of minicomputers. For example, early operating systems such as **DOS** (the **Disk Operating System**, from **Microsoft**) didn't think memory protection was important; thus, a malicious (or perhaps just a poorly-programmed) application could scribble all over memory. The first generations of the **Mac OS** (v9 and earlier) took a cooperative approach to job scheduling; thus, a thread that accidentally got stuck in an infinite loop could take over the entire system, forcing a reboot. The painful list of OS features missing in this generation of systems is long, too long for a full discussion here.

Fortunately, after some years of suffering, the old features of minicomputer operating systems started to find their way onto the desktop. For example, Mac OS X has UNIX at its core, including all of the features one would expect from such a mature system. Windows has similarly adopted many of the great ideas in computing history, starting in particular with Windows NT, a great leap forward in Microsoft OS technology. Even today's cell phones run operating systems (such as Linux) that are much more like what a minicomputer ran in the 1970s than what a PC

[10]We'll use asides and other related text boxes to call attention to various items that don't quite fit the main flow of the text. Sometimes, we'll even use them just to make a joke, because why not have a little fun along the way? Yes, many of the jokes are bad.

ASIDE: **THE IMPORTANCE OF UNIX**

It is difficult to overstate the importance of UNIX in the history of operating systems. Influenced by earlier systems (in particular, the famous **Multics** system from MIT), UNIX brought together many great ideas and made a system that was both simple and powerful.

Underlying the original "Bell Labs" UNIX was the unifying principle of building small powerful programs that could be connected together to form larger workflows. The **shell**, where you type commands, provided primitives such as **pipes** to enable such meta-level programming, and thus it became easy to string together programs to accomplish a bigger task. For example, to find lines of a text file that have the word "foo" in them, and then to count how many such lines exist, you would type: `grep foo file.txt|wc -l`, thus using the `grep` and `wc` (word count) programs to achieve your task.

The UNIX environment was friendly for programmers and developers alike, also providing a compiler for the new **C programming language**. Making it easy for programmers to write their own programs, as well as share them, made UNIX enormously popular. And it probably helped a lot that the authors gave out copies for free to anyone who asked, an early form of **open-source software**.

Also of critical importance was the accessibility and readability of the code. Having a beautiful, small kernel written in C invited others to play with the kernel, adding new and cool features. For example, an enterprising group at Berkeley, led by **Bill Joy**, made a wonderful distribution (the **Berkeley Systems Distribution**, or **BSD**) which had some advanced virtual memory, file system, and networking subsystems. Joy later co-founded **Sun Microsystems**.

Unfortunately, the spread of UNIX was slowed a bit as companies tried to assert ownership and profit from it, an unfortunate (but common) result of lawyers getting involved. Many companies had their own variants: **SunOS** from Sun Microsystems, **AIX** from IBM, **HPUX** (a.k.a. "H-Pucks") from HP, and **IRIX** from SGI. The legal wrangling among AT&T/Bell Labs and these other players cast a dark cloud over UNIX, and many wondered if it would survive, especially as Windows was introduced and took over much of the PC market...

ran in the 1980s (thank goodness); it is good to see that the good ideas developed in the heyday of OS development have found their way into the modern world. Even better is that these ideas continue to develop, providing more features and making modern systems even better for users and applications.

ASIDE: **AND THEN CAME LINUX**

Fortunately for UNIX, a young Finnish hacker named **Linus Torvalds** decided to write his own version of UNIX which borrowed heavily on the principles and ideas behind the original system, but not from the code base, thus avoiding issues of legality. He enlisted help from many others around the world, and soon **Linux** was born (as well as the modern open-source software movement).

As the internet era came into place, most companies (such as Google, Amazon, Facebook, and others) chose to run Linux, as it was free and could be readily modified to suit their needs; indeed, it is hard to imagine the success of these new companies had such a system not existed. As smart phones became a dominant user-facing platform, Linux found a stronghold there too (via Android), for many of the same reasons. And Steve Jobs took his UNIX-based **NeXTStep** operating environment with him to Apple, thus making UNIX popular on desktops (though many users of Apple technology are probably not even aware of this fact). And thus UNIX lives on, more important today than ever before. The computing gods, if you believe in them, should be thanked for this wonderful outcome.

## 2.7 Summary

Thus, we have an introduction to the OS. Today's operating systems make systems relatively easy to use, and virtually all operating systems you use today have been influenced by the developments we will discuss throughout the book.

Unfortunately, due to time constraints, there are a number of parts of the OS we won't cover in the book. For example, there is a lot of **networking** code in the operating system; we leave it to you to take the networking class to learn more about that. Similarly, **graphics** devices are particularly important; take the graphics course to expand your knowledge in that direction. Finally, some operating system books talk a great deal about **security**; we will do so in the sense that the OS must provide protection between running programs and give users the ability to protect their files, but we won't delve into deeper security issues that one might find in a security course.

However, there are many important topics that we will cover, including the basics of virtualization of the CPU and memory, concurrency, and persistence via devices and file systems. Don't worry! While there is a lot of ground to cover, most of it is quite cool, and at the end of the road, you'll have a new appreciation for how computer systems really work. Now get to work!

## References

[BS+09] "Tolerating File-System Mistakes with EnvyFS"
Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
USENIX '09, San Diego, CA, June 2009
*A fun paper about using multiple file systems at once to tolerate a mistake in any one of them.*

[BH00] "The Evolution of Operating Systems"
P. Brinch Hansen
In Classic Operating Systems: From Batch Processing to Distributed Systems
Springer-Verlag, New York, 2000
*This essay provides an intro to a wonderful collection of papers about historically significant systems.*

[B+72] "TENEX, A Paged Time Sharing System for the PDP-10"
Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, Raymond S. Tomlinson
CACM, Volume 15, Number 3, March 1972
*TENEX has much of the machinery found in modern operating systems; read more about it to see how much innovation was already in place in the early 1970's.*

[B75] "The Mythical Man-Month"
Fred Brooks
Addison-Wesley, 1975
*A classic text on software engineering; well worth the read.*

[BOH10] "Computer Systems: A Programmer's Perspective"
Randal E. Bryant and David R. O'Hallaron
Addison-Wesley, 2010
*Another great intro to how computer systems work. Has a little bit of overlap with this book — so if you'd like, you can skip the last few chapters of that book, or simply read them to get a different perspective on some of the same material. After all, one good way to build up your own knowledge is to hear as many other perspectives as possible, and then develop your own opinion and thoughts on the matter. You know, by thinking!*

[K+61] "One-Level Storage System"
T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner
IRE Transactions on Electronic Computers, April 1962
*The Atlas pioneered much of what you see in modern systems. However, this paper is not the best read. If you were to only read one, you might try the historical perspective below [L78].*

[L78] "The Manchester Mark I and Atlas: A Historical Perspective"
S. H. Lavington
Communications of the ACM archive
Volume 21, Issue 1 (January 1978), pages 4-12
*A nice piece of history on the early development of computer systems and the pioneering efforts of the Atlas. Of course, one could go back and read the Atlas papers themselves, but this paper provides a great overview and adds some historical perspective.*

[O72] "The Multics System: An Examination of its Structure"
Elliott Organick, 1972
*A great overview of Multics. So many good ideas, and yet it was an over-designed system, shooting for too much, and thus never really worked as expected. A classic example of what Fred Brooks would call the "second-system effect" [B75].*

[PP03] "Introduction to Computing Systems:
From Bits and Gates to C and Beyond"
Yale N. Patt and Sanjay J. Patel
McGraw-Hill, 2003
*One of our favorite intro to computing systems books. Starts at transistors and gets you all the way up to C; the early material is particularly great.*

[RT74] "The UNIX Time-Sharing System"
Dennis M. Ritchie and Ken Thompson
CACM, Volume 17, Number 7, July 1974, pages 365-375
*A great summary of UNIX written as it was taking over the world of computing, by the people who wrote it.*

[S68] "SDS 940 Time-Sharing System"
Scientific Data Systems Inc.
TECHNICAL MANUAL, SDS 90 11168 August 1968
Available: http://goo.gl/EN0Zrn
*Yes, a technical manual was the best we could find. But it is fascinating to read these old system documents, and see how much was already in place in the late 1960's. One of the minds behind the Berkeley Time-Sharing System (which eventually became the SDS system) was Butler Lampson, who later won a Turing award for his contributions in systems.*

[SS+10] "Membrane: Operating System Support for Restartable File Systems"
Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift
FAST '10, San Jose, CA, February 2010
*The great thing about writing your own class notes: you can advertise your own research. But this paper is actually pretty neat — when a file system hits a bug and crashes, Membrane auto-magically restarts it, all without applications or the rest of the system being affected.*

CHAPTER

# Systems

1

## CHAPTER CONTENTS

DOI: 10.1016/B978-0-12-374957-4.00010-4

## OVERVIEW

This book is about computer systems, and this chapter introduces some of the vocabulary and concepts used in designing computer systems. It also introduces "systems perspective", a way of thinking about systems that is global and encompassing rather than focused on particular issues. A full appreciation of this way of thinking can't really be captured in a short summary, so this chapter is actually just a preview of ideas that will be developed in depth in succeeding chapters.

The usual course of study of computer science and engineering begins with linguistic constructs for describing computations (software) and physical constructs for realizing computations (hardware). It then branches, focusing, for example, on the theory of computation, artificial intelligence, or the design of systems, which itself is usually divided into specialities: operating systems, transaction and database systems, computer architecture, software engineering, compilers, computer networks, security, and reliability. Rather than immediately tackling one of those specialties, we assume that the reader has completed the introductory courses on software and hardware, and we begin a broad study of computer systems that supports the entire range of systems specialties.

Many interesting applications of computers require

- fault tolerance
- coordination of concurrent activities
- geographically separated but linked data
- vast quantities of stored information
- protection from mistakes and intentional attacks
- interactions with many people

To develop applications that have these requirements, the designer must look beyond the software and hardware and view the computer system as a whole. In doing so, the designer encounters many new problems—so many that the limit on the scope of computer systems generally arises neither from laws of physics nor from theoretical impossibility, but rather from limitations of human understanding.

Some of these same problems have counterparts, or at least analogs, in other systems that have, at most, only incidental involvement of computers. The study of systems is one place where computer engineering can take advantage of knowledge from other engineering areas: civil engineering (bridges and skyscrapers), urban planning (the

Much wisdom about systems that has accumulated over the centuries is passed along in the form of folklore, maxims, aphorisms and quotations. Some of that wisdom is captured in the boxes at the bottom of these pages.

Everything should be made as simple as possible, but no simpler.

**— commonly attributed to Albert Einstein; it is actually a paraphrase of a comment he made in a 1933 lecture at Oxford.**

design of cities), mechanical engineering (automobiles and air conditioning), aviation and space flight, electrical engineering, and even ecology and political science. We start by looking at some of those common problems. Then we will examine two ways in which computer systems pose problems that are quite different. Don't worry if some of the examples are of things you have never encountered or are only dimly aware of. The sole purpose of the examples is to illustrate the range of considerations and similarities across different kinds of systems.

As we proceed in this chapter and throughout the book, we shall point out a series of *system design principles*, which are rules of thumb that usually apply to a diverse range of situations. Design principles are not immutable laws, but rather guidelines that capture wisdom and experience and that can help a designer avoid making mistakes. The astute reader will quickly realize that sometimes a tension, even to the point of contradiction, exists between different design principles. Nevertheless, if a designer finds that he or she is violating a design principle, it is a good idea to review the situation carefully.

At the first encounter of a design principle, the text displays it prominently. Here is an example, found on page 16.

**Avoid excessive generality**

*If it's good for everything, it's good for nothing.*

Each design principle thus has a formal title ("Avoid excessive generality") and a brief informal description ("If it's good for . . ."), which are intended to help recall the principle. Most design principles will show up several times, in different contexts, which is one reason why they are useful. The text highlights later encounters of a principle such as: *avoid excessive generality*. A list of all of the design principles in the book can be found on the inside front cover and also in the index, under "Design principles".

The remaining sections of this chapter discuss common problems of systems, the sources of those problems, and techniques for coping with them.

## 1.1 SYSTEMS AND COMPLEXITY

### 1.1.1 Common Problems of Systems in Many Fields

The problems one encounters in these many kinds of systems can usefully be divided into four categories: *emergent properties, propagation of effects, incommensurate scaling*, and *trade-offs*.

Seek simplicity and distrust it.

— **Alfred North Whitehead, *The Concept of Nature* (1920)**

#### *1.1.1.1 Emergent Properties*

*Emergent properties* are properties that are not evident in the individual components of a system, but show up when combining those components, so they might also be called surprises. Emergent properties abound in most systems, although there can always be a (fruitless) argument about whether or not careful enough prior analysis of the components might have allowed prediction of the surprise. It is wise to avoid this argument and instead focus on an unalterable fact of life: some things turn up only when a system is built.

Some examples of emergent properties are well known. The behavior of a committee or a jury often surprises outside observers. The group develops a way of thinking that could not have been predicted from knowledge about the individuals. (The concept of—and the label for—emergent properties originated in sociology.) When the Millennium Bridge for pedestrians over the River Thames in London opened, its designers had to close it after only a few days. They were surprised to discover that pedestrians synchronize their footsteps when the bridge sways, causing it to sway even more. Interconnection of several electric power companies to allow load sharing helps reduce the frequency of power failures, but when a failure finally occurs it may take down the entire interconnected structure. The political surprise is that the number of customers affected may be large enough to attract the unwanted attention of government regulators.

#### *1.1.1.2 Propagation of Effects*

The electric power inter-tie also illustrates the second category of system problems—*propagation of effects*—when a tree falling on a power line in Oregon leads to the lights going out in New Mexico, 1000 miles away. What looks at first to be a small disruption or a local change can have effects that reach from one end of a system to the other. An important requirement in most system designs is to limit the impact of failures. As another example of propagation of effects, consider an automobile designer's decision to change the tire size on a production model car from 13 to 15 inches. The reason for making the change might have been to improve the ride. On further analysis, this change leads to many other changes: redesigning the wheel wells, enlarging the spare tire space, rearranging the trunk that holds the spare tire, and moving the back seat forward slightly to accommodate the trunk redesign. The seat change makes knee room in the back seat too small, so the backs of the seats must be made thinner, which in turn reduces the comfort that was the original reason for changing the tire size, and it may also reduce safety in a collision. The extra weight of the trunk and rear seat design means that stiffer rear springs are now needed. The rear axle ratio must be modified to keep the force delivered to the road by the wheels correct, and the speedometer gearing must be changed to agree with the new tire size and axle ratio.

Those effects are the obvious ones. In complicated systems, as the analysis continues, more distant and subtle effects normally appear. As a typical example, the

> Our life is frittered away by detail . . . simplicity, simplicity, simplicity!
>
> **— Henry David Thoreau, *Walden; or, Life in the Woods* (1854)**

automobile manufacturer may find that the statewide purchasing office for Texas does not currently have a certified supplier for replacement tires of the larger size. Thus there will probably be no sales of cars to the Texas government for two years, which is the length of time it takes to add a supplier onto the certified list. Folk wisdom characterizes propagation of effects as: "There are no small changes in a large system".

#### *1.1.1.3 Incommensurate Scaling*

The third characteristic problem encountered in the study of systems is *incommensurate scaling*: as a system increases in size or speed, not all parts of it follow the same scaling rules, so things stop working. The mathematical description of this problem is that different parts of the system exhibit different orders of growth. Some examples:

- Galileo observed that "nature cannot produce a . . . giant ten times taller than an ordinary man unless by . . . greatly altering the proportions of his limbs and especially of his bones, which would have to be considerably enlarged over the ordinary" [*Discourses and Mathematical Demonstrations on Two New Sciences*, second day, Leiden, 1638]. In a classic 1928 paper, "On being the right size" [see Suggestions for Further Reading 1.4.1], J. B. S. Haldane uses the example of a mouse, which, if scaled up to the size of an elephant, would collapse of its own weight. For both examples, the reason is that weight grows with volume, which is proportional to the cube of linear size, but bone strength, which depends primarily on cross-sectional area, grows only with the square of linear size. Thus a real elephant requires a skeletal arrangement that is quite different from that of a scaled-up mouse.
- The Egyptian architect Sneferu tried to build larger and larger pyramids. Unfortunately, the facing fell off the pyramid at Meidum, and the ceiling of the burial chamber of the pyramid at Dashur cracked. He later figured out that he could escalate a pyramid to the size of the pyramids at Giza by lowering the ratio of the pyramid's height to its width. The reason this solution worked has apparently never been completely analyzed, but it seems likely that incommensurate scaling was involved—the weight of a pyramid increases with the cube of its linear size, while the strength of the rock used to create the ceiling of a burial chamber increases only with the area of its cross-section, which grows with the square.
- The captain of a modern oil supertanker finds that the ship is so massive that when underway at full speed it takes 12 miles to bring it to a straight line stop—but 12 miles is beyond the horizon as viewed from the ship's bridge (see Sidebar 1.1 for the details).
- The height of a skyscraper is limited by the area of lower floors that must be devoted to providing access to the floors above. The amount of access area

By undue profundity we perplex and enfeeble thought.

**— Edgar Allan Poe, "The Murders in the Rue Morgue" (1841)**

**Sidebar 1.1 Stopping a Supertanker** A little geometry reveals that the distance to the visual horizon is proportional to the square root of the height of the bridge. That height (presumably) grows with the first power of the supertanker's linear dimension. The energy required to stop or turn a supertanker is proportional to its mass, which grows with the third power of its linear dimensions. The time required to deliver the stopping or turning energy is less clear, but pushing on the rudder and reversing the propellers are the only tools available, and both of those have surface area that grows with the square of the linear dimension.

Here is the bottom line: if we double the tanker's linear dimensions, the momentum goes up by a factor of 8, and the ability to deliver stopping or turning energy goes up by only a factor of 4, so we need to see twice as far ahead. Unfortunately, the horizon will be only 1.414 times as far away. Inevitably, there is some size for which visual navigation must fail.

required (for example, for elevators and stairs) is proportional to the number of people who have offices on higher floors. That number is in turn proportional to the number of higher floors multiplied by the usable area of each floor. If all floors have the same area, and the number of floors increases, at some point the bottom floor will be completely used up providing access to higher floors, so the bottom floor provides no added value (apart from being able to brag about the building's height). In practice, the economics of office real estate dictate that no more than 25% of the lowest floor be devoted to access.

Incommensurate scaling shows up in most systems. It is usually the factor that limits the size or speed range that a single system design can handle. On the other hand, one must be cautious with scaling arguments. They were used at the beginning of the twentieth century to support the claim that it was a waste of time to build airplanes (see Sidebar 1.2).

#### 1.1.1.4 Trade-offs

The fourth problem of system design is that many constraints present themselves as *trade-offs*. The general model of a trade-off begins with the observation that there is a limited amount of some form of goodness in the universe, and the design challenge is first to maximize that goodness, second to avoid wasting it, and third to allocate it to the places where it will help the most. One common form of trade-off is sometimes called the *waterbed effect*: pushing down on a problem at one point causes another problem to pop up somewhere else. For example, one can typically push a hardware circuit to run at a higher clock rate, but that change increases both power consumption and the risk of timing errors. It may be possible to reduce the risk of timing errors by making the circuit physically smaller, but then less area will be available to dissipate the heat caused

KISS: Keep It Simple, Stupid.

**— traditional management folklore; source lost in the mists of time**

**Sidebar 1.2 Why Airplanes can't Fly** The weight of an airplane grows with the third power of its linear dimension, but the lift, which is proportional to surface area, can grow only with the second power. Even if a small plane can be built, a larger one will never get off the ground.

This line of reasoning was used around 1900 by both physicists and engineers to argue that it was a waste of time to build heavier-than-air machines. Alexander Graham Bell proved that this argument wasn't the whole story by flying box kites in Maine in the summer of 1902. In his experiments he attached two box kites side by side, a configuration that doubled the lifting surface area, but also allowed removal of the redundant material and supports where the two kites touched. Thus, the lift-to-weight ratio actually improved as the scale increased. Bell published his results in "The tetrahedral principle in kite structure" [see Suggestions for Further Reading 1.4.2].

by the increased power consumption. Another common form of trade-off appears in *binary classification*, which arises, for example, in the design of smoke detectors, spam (unwanted commercial e-mail message) filters, database queries, and authentication devices. The general model of binary classification is that we wish to classify a set of things into two categories based on the presence or absence of some property, but we lack a direct measure of that property. We therefore instead identify and use some indirect measure, known as a *proxy*. Occasionally, this scheme misclassifies something. By adjusting parameters of the proxy, the designer may be able to reduce one class of mistakes (in the case of a smoke detector, unnoticed fires; for a spam filter, legitimate messages marked as spam), but only at the cost of increasing some other class of mistakes (for the smoke detector, false alarms; for the spam filter, spam marked as legitimate messages). Appendix A explores the binary classification trade-off in more detail. Much of a system designer's intellectual effort goes into evaluating various kinds of trade-offs.

Emergent properties, propagation of effects, incommensurate scaling, and trade-offs are issues that the designer must deal with in every system. The question is how to build useful computer systems in the face of such problems. Ideally, we would like to describe a constructive theory, one that allows the designer systematically to synthesize a system from its specifications and to make necessary trade-offs with precision, just as there are constructive theories in such fields as communications systems, linear control systems, and (to a certain extent) the design of bridges and skyscrapers. Unfortunately, in the case of computer systems, we find that we were apparently born too soon. Although our early arrival on the scene offers the challenge to develop the missing theory, the problem is quickly apparent—we work almost entirely by analyzing *ad hoc* examples rather than by synthesizing.

Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.

**— Alan J. Perlis, "Epigrams in Programming" (1982)**

So, in place of a well-organized theory, we use case studies. For each subtopic in this book, we shall begin by identifying requirements with the apparent intent of deriving the system structure from the requirements. Then, almost immediately we switch to case studies and work backwards to see how real, in-the-field systems meet the requirements that we have set. Along the way we point out where systematic approaches to synthesizing a system from its requirements are beginning to emerge, and we introduce representations, abstractions, and design principles that have proven useful in describing and building systems. The intended result of this study is insight into how designers create real systems.

### 1.1.2 Systems, Components, Interfaces, and Environments

*Webster's Third New International Dictionary, Unabridged*, defines a system as "a complex unity formed of many often diverse parts subject to a common plan or serving a common purpose." Although this definition will do for casual use of the word, engineers usually prefer something a bit more concrete. We identify the "many often diverse parts" by naming them *components*. We identify the "unity" and "common plan" with the *interconnections* of the components, and we perceive the "common purpose" of a system to be to exhibit a certain behavior across its *interface* to an *environment*. Thus we formulate our technical definition: **A *system* is a set of interconnected components that has an expected behavior observed at the interface with its environment**.

The underlying idea of the concept of system is to divide all the things in the world into two groups: those under discussion and those not under discussion. Those things under discussion are part of the system—those that are not are part of the *environment*. For example, we might define the solar system as consisting of the sun, planets, asteroids, and comets. The environment of the solar system is the rest of the universe. (Indeed, the word "universe" is a synonym for environment.)

There are always interactions between a system and its environment; these interactions are the *interface* between the system and the environment. The interface between the solar system and the rest of the universe includes gravitational attraction for the nearest stars and the exchange of electromagnetic radiation. The primary interfaces of a personal computer typically include things such as a display, keyboard, speaker, network connection, and power cord, but there are also less obvious interfaces such as the atmospheric pressure, ambient temperature and humidity, and the electromagnetic noise environment.

One studies a system to predict its overall behavior, based on information about its components, their interconnections, and their individual behaviors. Identifying the components, however, depends on one's point of view, which has two aspects, *purpose* and *granularity*. One may, with different purposes in mind, look at a system quite

And simplicity is the unavoidable price we must pay for reliability.

**— Charles Anthony Richard Hoare, "Data Reliability" (1975)**

**Sidebar 1.3 Terminology: Words used to Describe System Composition** Since systems can contain component subsystems that are themselves systems from a different point of view, decomposition of systems is recursive. To avoid recursion in their writing, authors and designers have come up with a long list of synonyms, all trying to capture this same concept: *systems*, *subsystems*, *components*, *elements*, *constituents*, *objects*, *modules*, *submodules*, *assemblies*, *subassemblies*, and so on.

differently. One may also choose any of several different granularities. These choices affect one's identification of the components of the system in important ways.

To see how point of view can depend on purpose, consider two points of view of a jet aircraft as a system. The first looks at the aircraft as a flying object, in which the components of the system include the body, wings, control surfaces, and engines. The environment is the atmosphere and the earth, with interfaces consisting of gravity, engine thrust, and air drag. A second point of view looks at the aircraft as a passenger-handling system. Now, the components include seats, flight attendants, the air conditioning system, and the galley. The environment is the set of passengers, and the interfaces are the softness of the seats, the meals, and the air flowing from the air conditioning system.

In the first point of view, the aircraft as a flying object, the seats, flight attendants, and galley were present, but the designer considers them primarily as contributors of weight. Conversely, in the second point of view, as a passenger-handling system, the designer considers the engine as a source of noise and perhaps also exhaust fumes, and probably ignores the control surfaces on the wings. Thus, depending on point of view, we may choose to ignore or consolidate certain system components or interfaces.

The ability to choose granularity means that a component in one context may be an entire system in another. From an aircraft designer's point of view, a jet engine is a component that contributes weight, thrust, and perhaps drag. On the other hand, the manufacturer of the engine views it as a system in its own right, with many components—turbines, hydraulic pumps, bearings, afterburners, all of which interact in diverse ways to produce thrust—one interface with the environment of the engine. The airplane wing that supports the engine is a component of the aircraft system, but it is part of the environment of the engine system.

When a system in one context is a component in another, it is usually called a *subsystem* (but see Sidebar 1.3). The composition of systems from subsystems or decomposition of systems into subsystems can be carried on to as many levels as is useful.

In summary, then, to analyze a system one must establish a point of view to determine which things to consider as components, what the granularity of those

Pluralitas non est ponenda sine neccesitate. (Plurality should not be assumed without necessity.)

**— William of Ockham (14th century. Popularly known as "Occam's razor," though the idea itself is said to appear in writings of greater antiquity.)**

components should be, where the boundary of the system lies, and which interfaces between the system and its environment are of interest.

As we use the term, a *computer system* or an *information system* is a system intended to store, process, or communicate information under automatic control. Further, we are interested in systems that are predominantly digital. Here are some examples:

- a personal computer
- the onboard engine controller of an automobile
- the telephone system
- the Internet
- an airline ticket reservation system
- the space shuttle ground control system
- a World Wide Web site

At the same time we will sometimes find it useful to look at examples of nondigital and nonautomated information handling systems, such as the post office or library, for ideas and guidance.

### 1.1.3 Complexity

Webster's definition of "system" used the word "complex". Looking up that term, we find that *complex* means "difficult to understand". Lack of systematic understanding is the underlying feature of complexity. It follows that complexity is both a subjective and a relative concept. That is, one can argue that one system is more complex than another, but even though one can count up various things that seem to contribute to complexity, there is no unified measure. Even the argument that one system is more complex than another can be difficult to make compelling—again because of the lack of a unified measure. In place of such a measure, we can borrow a technique from medicine: describe a set of *signs* of complexity that can help confirm a diagnosis. As a corollary, we abandon hope of producing a definitive description of complexity. We must instead look for its signs, and if enough appear, argue that complexity is present. To that end, here are five signs of complexity:

1. **Large number of components.** Sheer size certainly affects our view of whether or not a system rates the description "complex".

2. **Large number of interconnections.** Even a few components may be interconnected in an unmanageably large number of ways. For example, the Sun and the known planets comprise only a few components, but every one has gravitational

Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher. (It is as if perfection be attained not when there is nothing more to add, but when there is nothing more to take away.)

**— Antoine de Saint-Exupéry, *Terre des Hommes* (1939)**

attraction for every other, which leads to a set of equations that are unsolvable (in closed form) with present mathematical techniques. Worse, a small disturbance can, after a while, lead to dramatically different orbits. Because of this sensitivity to disturbance, the solar system is technically *chaotic*. Although there is no formal definition of chaos for computer systems, that term is often informally applied.

3. **Many irregularities.** By themselves, a large number of components and interconnections may still represent a simple system, if the components are repetitive and the interconnections are regular. However, a lack of regularity, as shown by the number of exceptions or by non-repetitive interconnection arrangements, strongly suggests complexity. Put another way, exceptions complicate understanding.
4. **A long description.** Looking at the best available description of the system one finds that it consists of a long laundry list of properties rather than a short, systematic specification that explains every aspect. Theoreticians formalize this idea by measuring what they call the "Kolmogorov complexity" of a computational object as the length of its shortest specification. To a certain extent, this sign may be merely a reflection of the previous three, although it emphasizes an important aspect of complexity: it is relative to understanding. On the other hand, lack of a methodical description may also indicate that the system is constructed of ill-fitting components, is poorly organized, or may have unpredictable behavior, any of which add complexity to both design and use.
5. **A team of designers, implementers, or maintainers.** Several people are required to understand, construct, or maintain the system. A fundamental issue in any system is whether or not it is simple enough for a single person to understand all of it. If not, it is a complex system because its description, construction, or maintenance will require not just technical expertise but also coordination and communication across a team.

Again, an example can illustrate: contrast a small-town library with a large university library. There is obviously a difference in scale: the university has more books, so the first sign is present. The second sign is more subtle: where the small library may have a catalog to guide the user, the university library may have not only a catalog,

'Tis the gift to be simple, 'tis the gift to be free,
'Tis the gift to come down where we ought to be;
And when we find ourselves in the place just right,
'Twill be in the valley of love and delight.
When true simplicity is gained
To bow and to bend we shan't be ashamed;
To turn, turn will be our delight,
Till by turning, turning we come round right.

**— *Simple Gifts*, traditional Shaker hymn**

but also finding aids, readers' guides, abstracting services, journal indexes, and so on. Although these elaborations make the large library more useful (at least to the experienced user), they also complicate the task of adding a new item to the library: someone must add many interconnections (in this case, cross-references) so that the new item can be found in all the intended ways. The third sign, a large number of exceptions, is also apparent. Where the small library has only a few classifications (fiction, biography, nonfiction, and magazines) and a few exceptions (oversized books are kept over the newspaper rack), the university library is plagued with exceptions. Some books are oversized, others come on microfilm or on digital media, some books are rare or valuable and must be protected, the books that explain how to build a hydrogen bomb can be loaned only to certain patrons, some defy cataloging in any standard classification system. As for the fourth sign, any user of a large university library will confirm that there are no methodical rules for locating a piece of information and that library usage is an art, not a science.

Finally, the fifth sign of complexity, a staff of more than one person, is evident in the university library. Where many small towns do in fact have just one librarian—typically an energetic person who knows each book because at one time or another he or she has had occasion to touch it—the university library has not only many personnel, but even specialists who are familiar with only one facet of library operations, such as the microform collection.

The university library exhibits all five signs of complexity, but unanimity is not essential. On the other hand, the presence of only one or two of the signs may not make a compelling case for complexity. Systems considered in thermodynamics contain an unthinkably large number of components (elementary particles) and interactions, yet from the right point of view they do not qualify as complex because there is a simple, methodical description of their behavior. It is exactly when we lack such a simple, methodical description that we have complexity.

One objection to conceiving complexity as being based on the five signs is that all systems are indefinitely, perhaps infinitely, complex because the deeper one digs the more signs of complexity turn up. Thus, even the simplest digital computer is made of gates, which are made with transistors, which are made of silicon, which is composed of protons, neutrons, and electrons, which are composed of quarks, which some physicists suggest are describable as vibrating strings, and so on. We shall address this objection in a moment by limiting the depth of digging, a technique known as *abstraction*. The complexity that we are interested in and worried about is the complexity that remains despite the use of abstraction.

Whatever man builds . . . all of man's . . . efforts . . . invariably culminate in . . . a thing whose sole and guiding principle is . . . simplicity . . . perfection of invention touches hands with absence of invention, as if . . . [there] were a line that had not been invented but . . . [was] in the beginning . . . hidden by nature and in the end . . . found by the engineer.

**— Antoine de Saint-Exupéry, *Terre des Hommes* (1939)**

## 1.2 SOURCES OF COMPLEXITY

There are many sources of complexity, but two merit special mention. The first is in the number of requirements that the designer expects a system to meet. The second is one particular requirement: maintaining high utilization.

### 1.2.1 Cascading and Interacting Requirements

A primary source of complexity is just the list of requirements for a system. Each requirement, viewed by itself, may seem straightforward. Any particular requirement may even appear to add only easily tolerable complexity to an existing list of requirements. The problem is that the accumulation of many requirements adds not only their individual complexities but also complexities from their interactions. This interaction complexity arises from pressure for generality and exceptions that add complications, and it is made worse by change in individual requirements over time.

Most users of a personal computer have by now encountered some version of the following scenario: The vendor announces a new release of the program you use to manage your checkbook, and the new release has some feature that seems important or useful (e.g., it handles the latest on-line banking systems), so you order the program. Upon trying to install it, you discover that this new release requires a newer version of some shared library package. You track down that newer version and install it, only to find that the library package requires a newer version of the operating system, which you had not previously had any reason to install. Biting the bullet, you install the latest release of the operating system, and now the checkbook program works, but your add-on hard disk begins to act flaky. On investigation it turns out that the disk vendor's proprietary software is incompatible with the new operating system release. Unfortunately, the disk vendor is still debugging an update for the disk software, and the best thing available is a beta test version that will expire at the end of the month.

The underlying cause of this scenario is that the personal computer has been designed to meet many requirements: a well-organized file system, expandability of storage, ability to attach a variety of I/O devices, connection to a network, protection from malevolent persons elsewhere in the network, usability, reliability, low cost—the list goes on and on. Each of these requirements adds complexity of its own, and the interactions among them add still more complexity.

Similarly, the telephone system has, over the years, acquired a large number of line customizing features—call waiting, call return, call forwarding, originating and terminating call blocking, reverse billing, caller ID, caller ID blocking, anonymous call

When in doubt, make it stout, and of things you know about.
When in doubt, leave it out.

**— folklore sayings from the automobile industry**

**Sidebar 1.4 The Cast of Characters and Organizations** In concrete examples throughout this book, the reader will encounter a standard cast of characters named Alice, Bob, Charles, Dawn, Ella, and Felipe. Alice is usually the sender of a message, and Bob is its recipient. Charles is sometimes a mutual acquaintance of Alice and Bob. The others play various supporting roles, depending on the example. When we come to security, an adversarial character named Lucifer will appear. Lucifer's role is to crack the security measures and perhaps interfere with the presumably useful work of the other characters.

The book also introduces a few fictional organizations. There are two universities: Pedantic University, on the Internet at Pedantic.edu, and The Institute of Scholarly Studies, at Scholarly.edu. There are also four mythical commercial organizations on the Internet at TrustUs.com, ShopWithUs.com, Awesome.net, and Awful.net.

M.I.T. Professor Ronald Rivest introduced Alice and Bob to the literature of computer science in Suggestions for Further Reading 11.5.1. Any other resemblance to persons living or dead or organizations real or imaginary is purely coincidental.

rejection, do not disturb, vacation protection—again, the list goes on and on. These features interact in so many ways that there is a whole field of study of "feature interaction" in telephone systems. The study begins with debates over what *should* happen. For example, so-called 900 numbers have the feature called reverse billing—the called party can place a charge on the caller's bill. Alice (Alice is the first character we have encountered in our cast of characters, described in Sidebar 1.4) has a feature that blocks outgoing calls to reverse billing numbers. Alice calls Bob, whose phone is forwarded to a 900 number. Should the call go through, and if so, which party should pay for it, Bob or Alice? There are three interacting features, and at least four different possibilities: block the call, allow the call and charge it to Bob, ring Bob's phone, or add yet another feature that (for a monthly fee) lets Bob choose the outcome.

The examples suggest that there is an underlying principle at work. We call it the:

**Principle of escalating complexity**

*Adding a requirement increases complexity out of proportion.*

The principle is subjective because complexity itself is subjective—its magnitude is in the mind of the beholder. Figure 1.1 provides a graphical interpretation of the

Perfection must be reached by degrees; she requires the slow hand of time.

— **attributed to François-Marie Arouet (Voltaire)**

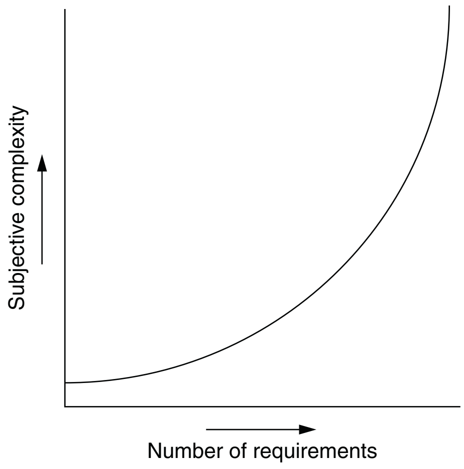

**FIGURE 1.1**

The principle of escalating complexity.

principle. Perhaps the most important thing to recognize in studying this figure is that the complexity barrier is soft: as you add features and requirements, you don't hit a solid roadblock to warn you to stop adding. It just gets worse.

As the number of requirements grows, so can the number of exceptions and thus the complications. It is the incredible number of special cases in the United States tax code that makes filling out an income tax return a complex job. The impact of any one exception may be minor, but the cumulative impact of many interacting exceptions can make a system so complex that no one can understand it. Complications also can arise from outside requirements such as insistence that a certain component must come from a particular supplier. That component may be less durable, heavier, or not as available as one from another supplier. Those properties may not prevent its use, but they add complexity to other parts of the system that have to be designed to compensate.

Meeting many requirements with a single design is sometimes expressed as a need for *generality.* Generality may be loosely defined as "applying to a variety of circumstances." Unfortunately, generality contributes to complexity, so it comes with a tradeoff, and the designer must use good judgment to decide how much of the generality is actually *wanted*. As an extreme example, an automobile with four independent steering wheels, each controlling one tire, offers some kind of ultimate in generality, almost all of which is unwanted. Here, both the aspect of unwantedness and the resulting complexity of guidance of the auto are obvious enough, but in many cases both of these aspects are more difficult to assess: How much does a proposed form of generality complicate the system, and to what extent is that generality really useful? Unwanted generality also contributes to complexity indirectly: users of a system with excessive generality will adopt styles of usage that simplify and suppress generality that they do not need. Different users may adopt different styles and then discover that they cannot easily exchange ideas with one another. Anyone who tries to use a personal computer customized by someone else will notice this problem.

Periodically, someone tries to design a vehicle that one can drive on the highway, fly, and use as a boat, but the result of such a general design does not seem to work

The best is the enemy of the good.

**— François-Marie Arouet (Voltaire), *Dictionnaire Philosophique* (1764)**

well in any of the intended modes of transport. To help counter excessive generality, experience suggests another design principle:*

**Avoid excessive generality**

*If it is good for everything, it is good for nothing.*

There is a tension between exceptions and generality. Part of the art of designing a subsystem is to make its features general enough to minimize the number of exceptions that must be handled as special cases. This area is one where the judgment of the system designer is most evident.

Counteracting the effects of incommensurate scaling can be an additional source of complexity. Haldane, in his essay "On being the right size", points out that small organisms such as insects absorb enough oxygen to survive through their skins, but larger organisms, which require an amount of oxygen proportional to the cube of their linear size, don't have enough surface area. To compensate for this incommensurate scaling, they add complexity in the form of lungs and blood vessels to absorb and deliver oxygen throughout their bodies. In the case of computers, the programmer of a 4-bit microprocessor to control a toaster can in a few days successfully write the needed code entirely with binary numbers, while the programmer of a video game with a 64-bit processor and 40 gigabytes of supporting data requires an extensive array of tools—compilers, image or video editors, special effects generators, and the like, as well as an operating system, to be able to get the job done within a lifetime. Incommensurate scaling has required employment of a far more complex set of tools.

Finally, a major source of complexity is that requirements *change*. System designs that are successful usually remain in use for a long time, during which the environment of the system changes. Improvements in hardware technology may lead the system maintainers to want to upgrade to faster, cheaper, or more reliable equipment. Meanwhile, knowledge of how to maintain the older equipment (and the supply of spare parts) may be disappearing. As users accumulate experience with the system, it becomes clearer that some additional requirements should have been part of the design and that some of the original requirements were less important than originally thought. Often a system will expand in scale, sometimes far beyond the vision of its original designers.

In each of these cases, the ground rules and assumptions that the original designers used to develop the system begin to lose their relevance. The system designers

*Computer industry consultant (and erstwhile instructor of the course for which this textbook was written) Michael Hammer suggested the informal version of this design principle.

A complex system that works is invariably found to have evolved from a simple system that works.

**— John Gall, *Systemantics* (1975)**

may have foreseen some environmental changes, but there were other changes they probably did not anticipate. As changes to meet unforeseen requirements occur, they usually add complexity. Because it can be difficult to change the architecture of a deployed system (Section 1.3 explains why), there is a powerful incentive to make changes within the existing architecture, whether or not that is the best thing to do. Propagation of effects can amplify the problems caused by change because more distant effects of a change may not be noticed until someone invokes some rarely used feature. When those distant effects finally do surface, the maintainer may again find it easiest to deal with them locally, perhaps by adding exceptions. Incommensurate scaling effects begin to dominate behavior when a later maintainer scales a system up in size or replaces the underpinnings with faster hardware. Again, the first response to these effects is usually to make local changes (sometimes called *patches*) to counteract them rather than to make fundamental changes in design that would require changing several modules or changing interfaces between modules.

A closely related problem is that as systems grow in complexity with the passage of time, even the simplest change, such as to repair a bug, has an increasing risk of introducing another bug because complexity tends to obscure the full impact of the repair. A common phenomenon in older systems is that the number of bugs introduced by a bug fix release may exceed the number of bugs fixed by that release.*

The bottom line is that as systems age, they tend to accumulate changes that make them more complex. The lifetime of a system is usually limited by the complexity that accumulates as it evolves farther and farther from its original design.

### 1.2.2 Maintaining High Utilization

One requirement by itself is frequently a specific source of complexity. It starts with a desire for high performance or high efficiency. Whenever a scarce resource is involved, an effort arises to keep its *utilization* high.

Consider, for example, a single-track railroad line running through a long, narrow canyon.† To improve the utilization of the single track, and push more traffic through, one might allow trains to run both ways at the same time by installing a switch and a short side track in a wide spot about halfway through the canyon. Then, if one is careful in scheduling, trains going in opposite directions will meet at the side track, where

*This phenomenon was documented by Laszlo A. Belady and Meir M. Lehman in "A model of large program development", *IBM Systems Journal 15*, 3 (1976), pages 225–252.

†Michael D. Schroeder suggested this example of a railroad line in a canyon.

Een schip op't droogh gezeylt, dat is een seeker baken. (A ship, sailed on to dry land, that is a certain beacon. Learn from the mistakes of others.)

**— Jacob Cats, *Mirror on Old and New Times* (1632), based on a Dutch proverb**

they can pass each other, effectively doubling the number of trains that the track can carry each day. However, the train operations are now much more complex than they used to be. If either train is delayed, the schedules of both are disrupted. A signaling system needs to be installed because human schedulers or operators may make mistakes. And—an emergent property—the trains now have a limit on their length. If two trains are to pass in the middle, at least one of them must be short enough to pull completely onto the side track.

The train in the canyon is a good illustration of how efforts to increase utilization can increase complexity. When striving for higher utilization, one usually encounters a general design principle that economists call

**The law of diminishing returns**

*The more one improves some measure of goodness, the more effort the next improvement will require.*

This phenomenon is particularly noticeable in attempts to use resources more efficiently: the more completely one tries to use a scarce resource, the greater the complexity of the strategies for use, allocation, and distribution. Thus a rarely used street intersection requires no traffic control beyond a rule that the car on the right has the right-of-way. As usage increases, one must apply progressively more complex measures: stop signs, then traffic lights, then marked turning lanes with multiphase lights, then vehicle sensors to control the lights. As traffic in and out of an airport nears the airport's capacity, measures such as stacking planes, holding them on the ground at distant airports, or coordinated scheduling among several airlines must be taken. As a general rule, the more one tries to increase utilization of a limited resource, the greater the complexity (see Figure 1.2).

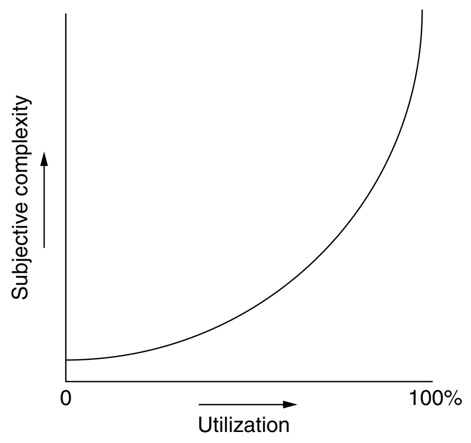

**FIGURE 1.2**

An example of diminishing returns: complexity grows with increasing utilization.

The perceptive reader will notice that Figures 1.1 and 1.2 are identical. It would

It is impossible to foresee the consequences of being clever.

— **Christopher Strachey, as reported by Roger Needham** be useful to memorize this figure because some version of it can be used to describe many different things about systems.

## 1.3 COPING WITH COMPLEXITY I

As one might expect, with many fields contributing examples of systems with common problems and sources of complexity, some common techniques for coping with complexity have emerged. These techniques can be loosely divided into four general categories: *modularity, abstraction, layering,* and *hierarchy.* The following sections sketch the general method of each of the techniques. In later chapters many examples of each technique will emerge. It is only by studying those examples that their value will become clear.

### 1.3.1 Modularity

The simplest, most important tool for reducing complexity is the divide-and-conquer technique: analyze or design the system as a collection of interacting subsystems, called *modules*. The power of this technique lies primarily in being able to consider interactions among the components within a module without simultaneously thinking about the components that are inside other modules.

To see the impact of reducing interactions, consider the debugging of a large program with, say, $N$ statements. Assume that the number of bugs in the program is proportional to its size and the bugs are randomly distributed throughout the code. The programmer compiles the program, runs it, notices a bug, finds and fixes the bug, and recompiles before looking for the next bug. Assume also that the time it takes to find a bug in a program is roughly proportional to the size of the program. We can then model the time spent debugging:

$$\begin{aligned} BugCount &\sim N \\ DebugTime &\sim N \times BugCount \\ &\sim N^2 \end{aligned}$$

Unfortunately, the debugging time grows proportional to the square of the program size.

Now suppose that the programmer divides the program into $K$ modules, each of roughly equal size, so that each module contains $N/K$ statements. To the extent that the modules implement independent features, one hopes that discovery of a bug usually will require examining only one module. The time required to debug any one module is thus reduced in two ways: the smaller module can be debugged faster, and

Plan to throw one away; you will, anyhow.

**— Frederick P. Brooks, *The Mythical Man Month* (1974)**

since there are fewer bugs in smaller programs, any one module will not need to be debugged as many times. These two effects are partially offset by the need to debug all $K$ modules. Thus our model of the time required to debug the system of $K$ modules becomes

$$\begin{aligned} DebugTime &\sim \left(\frac{N}{K}\right)^2 \times K \\ &\sim \frac{N^2}{K} \end{aligned}$$

Modularization into $K$ components thus reduces debugging time by a factor of $K$. Although the detailed mechanism by which modularity reduces effort differs from system to system, this property of modularity is universal. For this reason, one finds modularity in every large system.

The feature of modularity that we are taking advantage of here is that it is easy to replace an inferior module with an improved one, thus allowing incremental improvement of a system without completely rebuilding it. Modularity thus helps control the complexity caused by change. This feature applies not only to debugging but to all aspects of system improvement and evolution. At the same time, it is important to recognize a design principle associated with modularity, which we may call

**The unyielding foundations rule**

*It is easier to change a module than to change the modularity.*

The reason is that once an interface has been used by another module, changing the interface requires replacing at least two modules. If an interface is used by many modules, changing it requires replacing all of those modules simultaneously. For this reason, it is particularly important to get the modularity right.

Whole books have been written about modularity and the good things it brings. Sidebar 1.5 describes one of those books.

## 1.3.2 Abstraction

An important assumption in the numerical example of the effect of modularity on debugging time may not hold up in practice: that discovery of a bug should usually lead to examining just one module. For that assumption to hold true, there is a further requirement: there must be little or no propagation of effects from one module to

The purpose of computing is insight, not numbers.

**— Richard W. Hamming, *Numerical Methods for Scientists and Engineers* (1962)**

**Sidebar 1.5 How Modularity Reshaped the Computer Industry** Two Harvard Business School professors, Carliss Baldwin and Kim Clark, have written a whole book about modularity.* It discusses many things, but one of the most interesting is its explanation of a major transition in the computer business. In the 1960s, computer systems were a vertically integrated industry. That is, IBM, Burroughs, Honeywell, and several others each provided top-to-bottom systems and support, offering processors, memory, storage, operating systems, applications, sales, and maintenance; IBM even manufactured its own chips. By the 1990s, the industry had transformed into a horizontally organized one in which Intel sells processors, Micron sells memory, Seagate sells disks, Microsoft sells operating systems, Adobe sells text and image applications, Oracle sells database systems, and Gateway and Dell assemble boxes called "computers" out of components provided by the other players.

Carliss Baldwin and Kim Clark explain this transition as an example of modularity in action. The companies that created vertically integrated product lines immediately found complexity running amok, and they concluded that the only effective way to control it was to modularize their products. After a few experiments with wrong modularities (IBM originally designed different computers for business and for scientific applications), they eventually hit on effective ways of splitting things up and thereby keeping their development costs and delivery schedules under control:

- IBM developed the System/360 architecture specification, which could apply to machines of widely ranging performance. This modularity allowed any software to run on any size processor. IBM also developed a standard I/O bus and disk interface, so that any I/O device or disk manufactured by IBM could be attached to any IBM computer.
- Digital Equipment Corporation developed the PDP–11 family, which, with improving technology, could simultaneously be driven down in price toward the PDP–11/03 and up in function toward the PDP–11/70. A hardware-assisted emulation strategy for missing hardware instructions on the smaller machines allowed applications written for any machine to run on any other machine in the family. Digital also developed an I/O architecture, the UNIBUS®, that allowed any I/O device to attach to any PDP–11 model.

The long-range result was that once this modularity was defined and proven to be effective, other vendors were able to jump in and turn each module into a distinct business. The result is the computer industry since the 1990s, which is remarkably horizontal, especially considering its rather different shape only 20 years earlier.

(*Sidebar continues*)

*Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity* [see Suggestions for Further Reading 1.3.7]. Warning: the authors use the word "modularity" to mean all of modularity, abstraction, layering, and hierarchy.

Carliss Baldwin and Kim Clark also observe, more generally, that a market economy is characterized by modularity. Rather than having a self-supporting farm family that does everything for itself, a market economy has coopers, tinkers, blacksmiths, stables, dressmakers, and so on, each being more productive in a modular specialty, all selling things to one another using a universal interface—money.

another. Although there are lots of ways of dividing a system up into modules, some of these ways will prove to be better than others—"according to the natural formation, where the joint is, not breaking any part as a bad carver might" (Plato, *Phaedrus* 265e, Benjamin Jowett translation).

Thus the best divisions usually follow natural or effective boundaries. They are characterized by fewer interactions among modules and by less propagation of effects from one module to another. More generally, they are characterized by the ability of any module to treat all the others entirely on the basis of their external specifications, without need for knowledge about what goes on inside. This additional requirement on modularity is called *abstraction*. Abstraction is separation of interface from internals, of specification from implementation. Because abstraction nearly always accompanies modularity, some authors do not make any distinction between the two ideas. One sometimes sees the term *functional modularity* used to mean modularity with abstraction.

Thus one purchases a DVD player planning to view it as a device with a dozen or so buttons on the front panel and hoping never to look inside. If one had to know the details of the internal design of a television set in order to choose a compatible DVD player, no one would ever buy the player. Similarly, one turns a package over to an overnight delivery service without feeling a need to know anything about the particular kinds of vehicles or routes the service will use. Confidence that the package will be delivered tomorrow is the only concern.

In the computer world, abstraction appears in countless ways. The general ability of sequential circuits to remember state is abstracted into particular, easy-to-describe modules called *registers*. Programs are designed to hide details of their representation of complex data structures and details of which other programs they call. Users expect easy-to-use, button-pushing application interfaces such as computer

It must be remembered that there is nothing more difficult to plan, more doubtful of success, nor more dangerous to manage than the creation of a new system. For the initiator has the enmity of all who would profit by the preservation of the old institutions and merely lukewarm defenders in those who would gain by the new ones.

**— Niccolò Machiavelli, *The Prince* (1513, published 1532; Tr. by Thomas G. Bergin, Appleton-Century-Crofts, 1947)**

games, spreadsheet programs, or Web browsers that abstract incredibly complex underpinnings of memory, processor, communication, and display management.

The goal of minimizing interconnections among modules may be defeated if unintentional or accidental interconnections occur as a result of implementation errors or even well-meaning design attempts to sneak past modular boundaries in order to improve performance or meet some other requirement. Software is particularly subject to this problem because the modular boundaries provided by separately compiled subprograms are somewhat soft and easily penetrated by errors in using pointers, filling buffers, or calculating array indices. For this reason, system designers prefer techniques that enforce modularity by interposing impenetrable walls between modules. These techniques ensure that there can be no unintentional or hidden interconnections. Chapters 4 and 5 develop some of these techniques for enforcing modularity.

Well-designed and properly enforced modular abstractions are especially important in limiting the impact of faults because they control propagation of effects. As we shall see when we study fault tolerance in Chapter 8 [on-line], modules are the units of fault containment, and the definition of a failure is that a module does not meet its abstract interface specifications.

Closely related to abstraction is an important design rule that makes modularity work in practice:

**The robustness principle**

*Be tolerant of inputs and strict on outputs.*

This principle means that a module should be designed to be liberal in its interpretation of its input values, accepting them even if they are not within specified ranges, if it is still apparent how to sensibly interpret them. On the other hand, the module should construct its outputs conservatively in accordance with its specification—if possible making them even more accurate or more constrained than the specification requires. The effect of the robustness principle is to tend to suppress, rather than propagate or even amplify, noise or errors that show up in the interfaces between modules.

The robustness principle is one of the key ideas underlying modern mass production. Historically, machinists made components that were intended to mate by machining one of the components and then machining a second component to exactly fit against or into the first one, a technique known as *fitting*. The breakthrough came with the realization that if one specified *tolerances* for components and designed

We are faced with an insurmountable opportunity.

**— Pogo (Walt Kelley)**

each component to mate with any other component that was within its specified tolerance, then it would be possible to modularize and speed up manufacturing by having interchangeable parts. Apparently, this concept was first successfully applied in an 1822 contract to deliver rifles to the United States Army. By the time production lines for the Model T automobile were created, Henry Ford captured the concept in the aphorism, "In mass production there are no fitters."

The robustness principle plays a major role in computer systems. It is particularly important in human interfaces, network protocols, and fault tolerance, and, as Section 1.4 of this chapter explains, it forms the basis for digital logic. At the same time, a tension exists between the robustness principle and another important design principle:

**The safety margin principle**

*Keep track of the distance to the cliff, or you may fall over the edge.*

When inputs are not close to their specified values, that is usually an indication that something is starting to go wrong. The sooner that something going wrong can be noticed, the sooner it can be fixed. For this reason, it is important to track and report out-of-tolerance inputs, even if the robustness principle would allow them to be interpreted successfully.

Some systems implement the safety margin principle by providing two modes of operation, which might be called "shake-out" and "production". In shake-out mode, modules check every input carefully and refuse to accept anything that is even slightly out of specification, thus allowing immediate discovery of problems and of programming errors near their source. In production mode, modules accept any input that they can reasonably interpret, in accordance with the robustness principle. Carefully designed systems blend the two ideas: accept any reasonable input but report any input that is beginning to drift out of tolerance so that it may be repaired before it becomes completely unusable.

### 1.3.3 Layering

Systems that are designed using good abstractions tend to minimize the number of interconnections among their component modules. One powerful way to reduce module interconnections is to employ a particular method of module organization known as *layering*. In designing with layers, one builds on a set of mechanisms that is already complete (a lower layer) and uses them to create a different complete set of mechanisms (an upper layer). A layer may itself be implemented as several modules,

There is no such thing as a small change to a large system.

**— systems folklore, source lost in the mists of time**

but as a general rule, a module of a given layer interacts only with its peers in the same layer and with the modules of the next higher and next lower layers. That restriction can significantly reduce the number of potential intermodule interactions in a big system.

Some of the best examples of this approach are found in computer systems: an interpreter for a high-level language is implemented using a lower-level, more machine-oriented, language. Although the higher-level language doesn't allow any new programs to be expressed, it is easier to use, at least for the application for which it was designed.

Thus, nearly every computer system comprises several layers. The lowest layer consists of gates and memory cells, upon which is built a layer consisting of a processor and memory. On top of this layer is built an operating system layer, which acts as an augmentation of the processor and memory layer. Finally, an application program executes on this augmented processor and memory layer. In each layer, the functions provided by the layer below are rearranged, repackaged, reabstracted, and reinterpreted as appropriate for the convenience of the layer above. As will be seen in Chapter 7 [on-line], layers are also the primary organizing technique of data communication networks.

Layered design is not unique to computer systems and communications. A house has an inner structural layer of studs, joists, and rafters to provide shape and strength, a layer of sheathing and drywall to keep the wind out, a layer of siding, flooring and roof tiles to make it watertight, and a cosmetic layer of paint to make it look good. Much of mathematics, particularly algebra, is elegantly organized in layers (in the case of algebra, integers, rationals, complex numbers, polynomials, and polynomials with polynomial coefficients), and that organization provides a key to deep understanding.

### 1.3.4 Hierarchy

The final major technique for coping with complexity also reduces interconnections among modules but in a different, specialized way. Start with a small group of modules, and assemble them into a stable, self-contained subsystem that has a well-defined interface. Next, assemble a small group of subsystems to produce a larger subsystem. This process continues until the final system has been constructed from a small number of relatively large subsystems. The result is a tree-like structure known as a *hierarchy*. Large organizations such as corporations are nearly always set up this way, with a manager responsible for only five to ten employees, a higher-level manager responsible for five to ten managers, on up to the president of the company, who may

The first 80 percent of a project takes 80 percent of the effort.
The last 20 percent takes another 80.

**— source unknown**

supervise five to ten vice presidents. The same thinking applies to armies. Even layers can be thought of as a kind of degenerate one-dimensional hierarchy.

There are many other striking examples of hierarchy, ranging from microscopic biological systems to the assembly of Alexander's empire. A classic paper by Herbert Simon, "The architecture of complexity" [Suggestions for Further Reading 1.4.3], contains an amazing range of such examples and offers compelling arguments that, under evolution, hierarchical designs have a better chance of survival. The reason is that hierarchy constrains interactions by permitting them only among the components of a subsystem. Hierarchy constrains a system of $N$ components, which in the worst case might exhibit $N \times (N - 1)$ interactions, so that each component can interact only with members of its own subsystem, except for an interface component that also interacts with other members of the subsystem at the next higher level of hierarchy. (The interface component in a corporation is called a "manager"; in an army it is called the "commanding officer"; for a program it is called the "application programming interface".) If subsystems have a limit of, say, 10 components, this number remains constant no matter how large the system grows. There will be $N/10$ lowest level subsystems, $N/100$ next higher level subsystems, and so on, but the total number of subsystems, and thus the number of interactions, remains proportional to $N$. Analogous to the way that modularity reduces the effort of debugging, hierarchy reduces the number of potential interactions among modules from square-law to linear.

This effect is most strongly noticed by the designer of an individual module. If there are no constraints, each module should in principle be prepared to interact with every other module of the system. The advantage of a hierarchy is that the module designer can focus just on interactions with the interfaces of other members of its immediate subsystem.

## 1.3.5 Putting it Back Together: Names Make Connections

The four techniques for coping with complexity—modularity, abstraction, layering, and hierarchy—provide ways of dividing things up and placing the resulting modules in suitable relation one to another. However, we still need a way of connecting those modules. In digital systems, the primary connection method is that one module *names* another module that it intends to use. Names allow postponing of decisions, easy replacement of one module with a better one, and sharing of modules. Software uses names in an obvious way. Less obviously, hardware modules connected to a bus also use names for interconnection—addresses, including bus addresses, are a kind of name.

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

**— Douglas Hofstadter: *Gödel, Escher, Bach: An Eternal Golden Braid* (1979)**

In a modular system, one can usually find several ways to combine modules to implement a desired feature. The designer must at some point choose a specific implementation from among many that are available. Making this choice is called *binding*. Recalling that the power of modularity comes from the ability to replace an implementation with a better one, the designer usually tries to maintain maximum flexibility by delaying binding until the last possible instant, perhaps even until the first instant that the feature is actually needed.

One way to delay binding is just to name a feature rather than implementing it. Using a name allows one to design a module as if a feature of another module exists, even if that feature has not yet been implemented, and it also makes it mechanically easy to later choose a different implementation. By the time the feature is actually invoked, the name must, of course, be bound to a real implementation of the other module. Using a name to delay or allow changing a binding is called *indirection*, and it is the basis of a design principle:

**Decouple modules with indirection**

*Indirection supports replaceability.*

A folk wisdom version of this principle, attributed to computer scientist David Wheeler of the University of Cambridge, exaggerates the power of indirection by suggesting that "any problem in a computer system can be solved by adding a layer of indirection." A somewhat more plausible counterpart of this folk wisdom is the observation that any computer system can be made faster by removing a layer of indirection.

When a module has a name, several other modules can make use of it by name, thereby sharing the design effort, cost, or information contained in the first module. Because names are a cornerstone element of modularity in digital systems, Chapters 2 and 3 are largely about the design of naming schemes.

## 1.4 COMPUTER SYSTEMS ARE THE SAME BUT DIFFERENT

As we have repeatedly suggested, there is an important lesson to be drawn from the wide range of examples used up to this point to illustrate system problems. Certain common problems show up in all complex systems, whatever their field. Emergent properties, propagation of effects, incommensurate scaling, and trade-offs are considerations in activities as diverse as space station design, management of the economy,

A system is never finished being developed until it ceases to be used.

**— attributed to Gerald M. Weinberg**

the building of skyscrapers, gene-splicing, petroleum refineries, communication satellite networks, and the governing of India, as well as in the design of computer systems. Furthermore, the techniques that have been devised for coping with complexity are universal. Modularity, abstraction, layering, and hierarchy are used as tools in most fields that deal with complex systems. It is therefore useful for the computer system designer to investigate systems from other fields, both to gain additional perspective on how system problems arise and to discover specific techniques from other fields that may also apply to computer systems. Stated briefly, we conclude that *computer systems are the same as all other systems*.

But there is one problem with that conclusion: it is wrong. There are at least two significant ways in which computer systems differ from every other kind of system with which designers have experience:

- *The complexity of a computer system is not limited by physical laws.*
- *The rate of change of computer system technology is unprecedented.*

These two differences have an enormous impact on complexity and on ways of coping with it.

### 1.4.1 Computer Systems have no Nearby Bounds on Composition

Computer systems are mostly digital, and they are controlled by software. Each of these two properties separately leads to relaxations of what, in other systems, would be limits on complexity arising from physical laws.

Consider first the difference between analog and digital systems. All analog systems have the engineering limitation that each component of the system contributes noise. This noise may come from the environment in the form of, for example, vibration or electromagnetic radiation. Noise may also appear because the component's physical behavior does not precisely follow any tractable model of operation: the pile of rocks that a civil engineer specifies to go under a bridge abutment does not obey a simple deformation model; a resistor in an electronic circuit generates random noise whose level depends on the temperature. When analog components are composed into systems, the noise from individual components accumulates (if the noise sources are statistically independent, the noise may accumulate only slowly but it still accumulates). As the number of components increases, noise will at some point dominate the behavior of the system. (This analysis applies to systems designed by human engineers.

I was to learn later in life that we tend to meet any new situation by reorganisation; and what a wonderful method it can be for creating the illusion of progress while producing confusion, inefficiency and demoralisation.

— **shortened version of an observation by Charlton Ogburn, "Merrill's Marauders: The truth about an incredible adventure", *Harper's Magazine* (January 1957). Widely but improbably misattributed to Petronius Arbiter (ca. A.D. 60)**

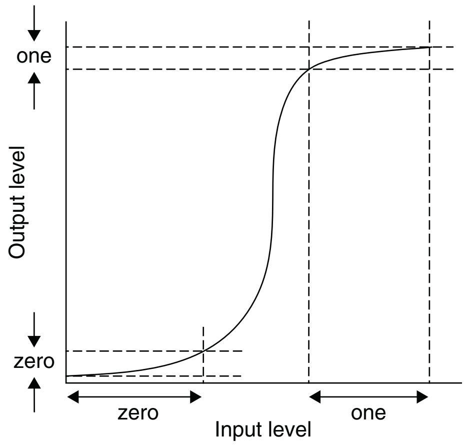

**FIGURE 1.3**

How gain and non-linearity of a digital component restore levels. The input level and output level span the same range of values, but the range of accepted inputs is much wider than the range of generated outputs.

Natural biological, thermodynamic, and macroeconomic systems, composed of billions of analog components, somehow use hierarchy, layering, abstraction, and modularity to operate despite noise, but they are so complex that we do not understand them well enough to adopt the same techniques.)

Noise thus provides a limit on the number of analog components that a designer can usefully compose or on the number of stages that a designer can usefully cascade. This argument applies to any engineered analog system: a bridge across a river, a stereo, or an airliner. It is the reason a photocopy of a photocopy is harder to read than the original. There may also be other limits on size (arising from the strength of materials, for example), but noise is always a limit on the complexity of analog systems.

In contrast, digital systems are noise-free; complexity can therefore grow without any constraint of a bound arising from noise. The designers of digital logic use a version of the *robustness principle* known as the *static discipline*. This discipline is the primary source of the magic that seems to surround digital systems. The static discipline requires that the range of analog values that a device accepts as meaning the digital value ONE (or ZERO) be wider than the range of analog values that the device puts out when it means digital ONE (or ZERO). This discipline is an example of being tolerant of inputs and strict on outputs.

Digital systems are, at some lower level, constructed of analog components. The analog components chosen for this purpose are non-linear, and they have gain between input and output. When used appropriately, non-linearity allows inputs to have a wide tolerance, and gain ensures that outputs stay within narrow specifications, as shown in Figure 1.3. Together they produce the property of digital circuits called level restoration or regeneration. Regenerated signal levels appear at the output of every digital component, whatever their level of granularity: a gate, a flip-flop, a memory chip, a processor, or a complete computer system. Regenerated levels create clean interfaces that allow one subsystem to be connected to the next with

The probability of failure of a system tends to be proportional to the confidence that its designer has in its reliability.

**— systems folklore, source lost**

confidence. Unlike the civil engineer's pile of rocks, a logic gate performs exactly as its designer intends.

The static discipline and level restoration do *not* guarantee that devices with digital inputs and outputs never make mistakes. Any component can fail. Or an input signal that is intended to be a ONE may be so far out of tolerance that the receiving component accepts it as a ZERO. When that happens, the output of the component that accepted that value incorrectly is likely to be wrong, too. The important consequence is that digital components make big mistakes, not little ones, and as we shall see when we reach the chapter on fault tolerance, big mistakes are relatively easy to detect and handle.

If a signal does not accumulate noise as it goes through a string of devices, then noise does not limit the number of devices one can string together. In other words, noise does not constrain the maximum depth of composition for digital systems. Unlike analog systems, digital systems can grow in complexity until they exceed the ability of their designers to understand them. As of 2009, processor chips contain over two billion transistors, far more than any analog chip. No airliner has nearly that many components—except in its on-board computers.

The second reason composition has no nearby bounds is that computer systems are controlled by software. Bad as the contribution to complexity from the static discipline may be, the contribution from software turns out to be worse. Hardware is at least subject to *some* physical limits—the speed of light, the rate of settling of signals in real semiconductor materials, unwanted electrical coupling between adjacent components, the rate at which heat can be removed, and the space that it occupies. Software appears to have no physical limits whatever beyond the availability of memory to store it and processors to execute it. As a result, composition of software can go on as fast as people can create it. Thus one routinely hears of operating systems, database systems, and even word processors consisting of more than 10 million program statements.

In principle, abstraction can help control software composition by hiding implementation beneath module interfaces. The problem is that most abstractions are, in reality, slightly "leaky" in that they don't perfectly conceal the underlying implementation. A simple example of leakiness is addition of integers: in most implementations, the addition operation perfectly matches the mathematical specification as long as the result fits in the available word size, but if the result is larger than that, the resulting overflow becomes a complication for the programmer. Leakiness, like noise in analog systems, accumulates as the number of software modules grows. Unlike noise, it accumulates in the form of complexity, so the lack of physical constraints on

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.

**— Douglas Adams, *Mostly Harmless (Hitchhiker's Guide to the Galaxy V)* (1993)**

software composition remains a fundamental problem. It is, therefore, mechanically easy to create a system with complexity that is far beyond the ability of its designers to understand. And since it is easy, it happens often, and sometimes with disastrous results.*

Between the absence of a noise-imposed limit on composition of digital hardware and very distant physical limits on composition of software, it is too easy for an unwary designer to misuse the tools of modularity, abstraction, layering, and hierarchy to include still more complexity. This phenomenon is quite unknown in the design of bridges and airliners. *In contrast with other systems, computer systems allow composition to a depth whose first limit is the designer's ability to understand.* Unfortunately, this lack of nearby natural, physical bounds on depth of composition tempts designers to build more complex systems. If nature does not impose a nearby limit on composition, the designer must self-impose a limit. Since it can be hard to say no to a reasonable-sounding feature, features keep getting added. Therein lies the fate of too many computer system designs.

### 1.4.2 d(technology)/dt is Unprecedented

For reasons partly explained by Sidebar 1.6, during the last 35 years the cost of the digital hardware used for computation and communication has dropped an average of about 30% each year. This rate of change means that just two years' passage of time has been enough to allow technology to cut prices in half, and in seven or eight years it has led to a drop in prices by a factor of 10. Some components have experienced even greater rates of improvement. Figure 1.4 shows the cost of magnetic disk storage over a 25-year span. During that time, disk prices have actually dropped by a factor of 10 roughly every five years, so disk prices have dropped nearly 60% each year. Disk experts project a similar rate of improvement for at least another few years. Their projection seems relatively safe, since no major roadblocks have been reported by development laboratories that are already working on the next rounds of magnetic recording technology. Similar charts apply to random access memory, processor cost, and the speed of optical fiber transmission.

This rapid change of technology has created a substantial difference between computer systems and other engineering systems. Since complex systems can take several years to build, by the time a computer system is ready for delivery, the

*The terminology "leaky" is apparently due to software developer Joel Spolsky.

Structural engineering is the art of modeling materials we do not wholly understand, into shapes we cannot precisely analyse so as to withstand forces we cannot properly assess, in such a way that the public has no reason to suspect the extent of our ignorance.

**— A. R. Dykes, Scottish Branch, Institution of Structural Engineers (1946)**

**Sidebar 1.6 Why Computer Technology has Improved Exponentially with Time** Popular media frequently use the term "exponential" to describe the explosive rate of improvement of computer technology. Stephen Ward has pointed out that there is a good reason this adjective is appropriate: computer technology appears to be the rare engineering discipline in which the technology being improved is routinely employed to improve the technology. People building airplanes, bridges, skyscrapers, and chemical plants rarely, if ever, have this opportunity.

For example, the performance of a microprocessor is determined at least in part by the cleverness of its layout, which in turn is limited by the time available to use computer-assisted layout tools that can take advantage of lithography advances. If Intel, through improved layout, makes a version of the Pentium that is twice as fast, as soon as that new Pentium is available, it will be used as the processor to make the layout tools for the next Pentium run twice as fast; the next design can benefit from twice as much computation in its layout. This effect is probably one of the drivers of Moore's law, which predicts an exponential increase in component count on chips with a doubling time of 18 months [Suggestions for Further Reading 1.6.1].

If indeed the rate at which we can improve our technology is proportional to the quality of the technology itself, we can express this idea as

$$\frac{d(technology)}{dt} = K \times technology$$

which has an exponential solution,

$$technology = e^{K \cdot t}$$

The actual situation is, of course, far more complicated than that equation suggests, but all equations that even remotely resemble that form, in which technology's rate of growth is some positive function of its current state, have growing exponentials in their solution.

In the real world, exponentials must eventually hit some limit. In hardware there are fairly clear fundamental physical limits to exponential growth, such as the uncertainty principle, the minimum energy required to switch a gate, and the rate at which heat can be removed from a device. The interesting part is that it isn't obvious which one is going to become the roadblock, or when. Thus far, engineering ingenuity in exploiting trade-offs has postponed the day of reckoning. For software, similar limits on exponential growth must exist, but their nature is not at all clear.

More to the immediate point, virtually every improvement in computer and communications technology—whether faster chips, better Internet routing algorithms, more effective prototyping languages, better browser interfaces, faster compilers, bigger disks, or larger RAM—is immediately put to work by everyone who is working on faster chips, better Internet routing algorithms, more effective prototyping languages, better browser interfaces, faster compilers, bigger disks, or larger RAM. Computer system designers live inside a giant feedback system that, at least for the moment, is enjoying exponential solutions.

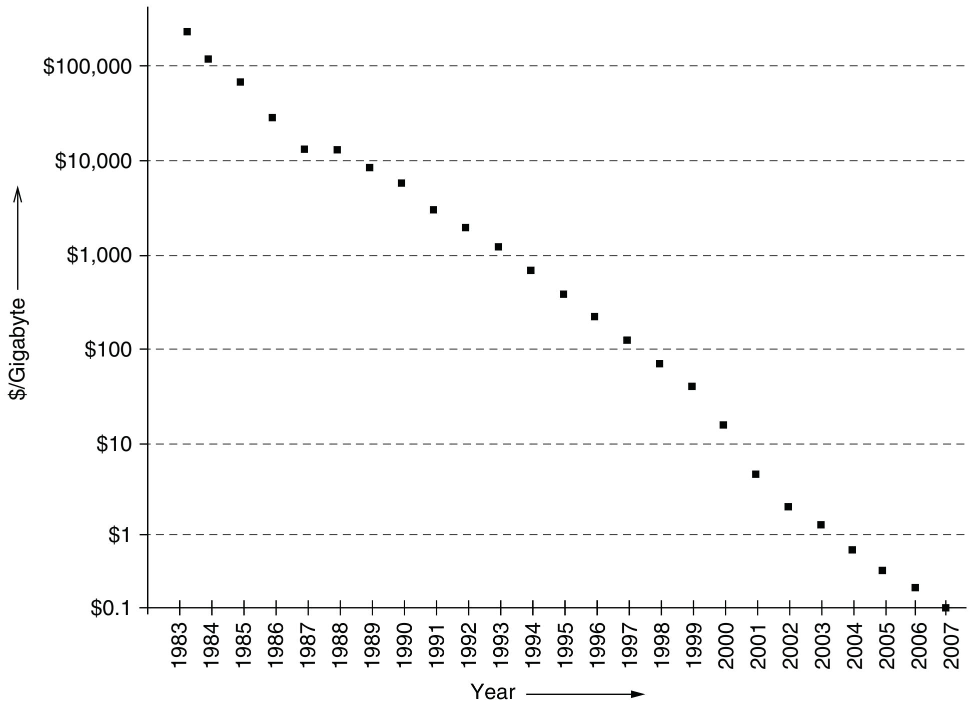

**FIGURE 1.4**

Magnetic disk price history and projection, 1983–2007.

ground rules under which it was originally designed have shifted. Incommensurate scaling typically means that the designer must adjust for strains when any system parameter changes by a factor of 2, because not all of the components scale up (or down) by the same proportion. More to the point, a whole new design is usually needed when any system parameter changes by a (decimal) order of magnitude. This rule of thumb about strains caused by parameter changes gives us our next design principle:

**The incommensurate scaling rule**

*Changing any system parameter by a factor of 10 usually requires a new design.*

If you design it so that it can be assembled wrong, someone will assemble it wrong.

**— Edward A. Murphy, Jr. (paraphrase of the original Murphy's law, 1949; see sidebar 2.5)**

This rule, when combined with the observed rate of change of technology, means that by the time a newly designed computer system is ready for delivery it may have already needed two rounds of adjustment and be ready for a complete redesign. Even if the designer has tried to predict the impact of technology change, crystal balls are at best cloudy. Worse, during the development of the system, things may run an order of magnitude slower than they will when the system is finished, the code and data don't fit in the available address space, or perhaps the data has to be partitioned across several hard disks instead of nicely fitting on one. One can compensate for each of these problems, but each such compensation absorbs intellectual resources and contributes complexity to the development process.

Even without those adjustments or redesign, the original plan was probably already a new design. A bridge (or airplane) may have a modest number of things that are different from the previous one, but a civil (or aeronautical) engineer almost always ends up designing something that is only a little different from some previous bridge (or airplane). In the case of computer systems, ideas that were completely unrealistic a year or two ago can become mainstream in no time, so the computer system designer almost always ends up designing something that is significantly different from the previous computer system. This difference makes deep analysis of previous designs more rewarding for civil and aeronautical engineers than for computer system designers, and also usually means that in computer systems there hasn't been time to discover and iron out most of the mistakes of the previous design before going on to the next major revision. Those mistakes can contribute strongly to complexity.

Because technology has improved so rapidly, the field of computer system design tends to place much less emphasis on detailed performance analysis and fine-tuning than do most other engineering endeavors. Where an electric power generation system may benefit dramatically from a new steam turbine that improves energy transfer by 1%, a needed 20% improvement in performance of a computer system can usually be obtained just by waiting four months for the next round of hardware product announcements. If a proposal to rewrite an application to obtain that same improvement would require a year of work, it is probably more cost-effective to just wait for technology change to solve the problem. Put another way, rapidly improving technology means that brute-force solutions (buy more memory, wait for a faster processor, use a simpler algorithm) are often the right approach in computer systems, whereas in other systems they may be unthinkable. The owner of the railroad through the canyon probably would not view as economically reasonable a proposal to blast the canyon wider and install a second track. Even if the resources were available, the environmental impact would be a deterrent.

This "telephone" has too many shortcomings to be seriously considered as a means of communication. The device is inherently of no value to us.

**— frequently attributed to an 1876 Western Union internal memo, but there is no evidence of this memo and it is probably a myth.**

A second major consequence of the rapid rate of change of technology in computer systems is that usability, and related qualities that go under the label "human engineering", of computer systems is always ragged. It takes years of trial and error to make systems usable, friendly, and forgiving, but by the time one level of computer technology has been tamed, a new level of computer technology opens the possibilities of many new features at the same cost, or of providing the previous features more cheaply to a vast new audience of unprepared users.

Similarly, legal and judicial processes take decades to come to grips with new issues, as people debate the wisdom of various policies, discover abuses, and explore alternative remedies. In the face of rapidly changing computer system technology, these processes fall far behind, delaying resolution of such concerns as how to reward innovative software ideas, or what rules should protect information stored in computers, and adding uncertainty of requirements to the burden of the computer system designer.*

Finally, modern high-speed communications with global reach have greatly accelerated the rate at which people discover that a new technology is useful and adopt it. Where it took several decades for electricity and the telephone to move from curiosities to widespread use, recent innovations such as digital cameras and DVDs have swept their markets in less than a decade, and a single mention of a previously obscure World Wide Web site on CNN or in *Newsweek* magazine can cause that site to be suddenly overwhelmed with millions of hits per day. More generally, newly viable applications, such as peer-to-peer file sharing, can change the shape of the workload on existing systems practically overnight.

Thus, the study of computer systems involves telescoping of the usual processes of planning, examining requirements, tailoring details, and integrating with users and society. This telescoping leads to the delivery of systems that have rough edges and without the benefit of the cleverest thought. People who build airplanes and bridges do not have to face these problems. Such problems can be viewed either as a frustrating difficulty or as an exciting challenge, depending on one's perspective.

## 1.5 COPING WITH COMPLEXITY II

Modest physical limits in hardware and very distant physical limits in software together give us the opportunity to create systems of unimaginable—and unmanageable—complexity, and the rapid pace of technology change tempts designers to deliver

*Lawrence Lessig provides a good analysis of the interactions of law, society, and computer technology in *Code: and Other Laws of Cyberspace* [Suggestions for Further Reading 1.1.4].

Books will soon be obsolete in the public schools. . . . It is possible to teach every branch of human knowledge with the motion picture. Our school system will be completely changed inside of ten years.

**— Thomas A. Edison, as quoted in the *New York Dramatic Mirror* (July 9, 1913)**

systems using new and untested ground rules. These two effects amplify the complexity of computer systems when compared with systems from other engineering areas. Thus, computer system designers need some additional tools to cope with complexity.

### 1.5.1 Why Modularity, Abstraction, Layering, and Hierarchy aren't Enough

Modularity, abstraction, layering, and hierarchy are a major help, but by themselves they aren't enough to keep the resulting complexity under control. The reason is that all four of those techniques assume that the designer understands the system being designed. In the real, fast-changing world of computer systems, it is hard to choose

- the *right* modularity from a sea of plausible alternative modularities.
- the *right* abstraction from a sea of plausible alternative abstractions.
- the *right* layering from a sea of plausible alternative layerings.
- the *right* hierarchy from a sea of plausible alternative hierarchies.

Although some design principles are available, they are far too few, and the only real guidance comes from experience with previous systems.

As might be expected, designers of computer systems have developed and refined at least one additional technique to cope with complexity. Designers of other kinds of systems use this technique as well, but they usually do not consider it to be so fundamental to success as it is for computer systems, probably because the technique is particularly feasible with software. It is a development process called *iteration*.

### 1.5.2 Iteration

The essence of iteration is to start by building a simple, working system that meets only a modest subset of the requirements and then evolve that system in small steps to gradually encompass more and more of the full set of requirements. The idea is that small steps can help reduce the risk that complexity will overwhelm a system design. Having a working system available at all times helps provide assurance that something can be built and provides on-going experience with the current technology ground rules as well as an opportunity to discover and fix bugs. Finally, adjustments for technology changes that arrive during the system development are easier to incorporate as part of one or more of the iterations. When you see a piece of software identified as "release 5.4", that is usually an indication that the vendor is using iteration.

Successful iteration requires considerable foresight. That foresight involves several elements, two of which we identify as design principles:

I think there is a world market for maybe five computers.

**— Frequently claimed to be said by Thomas J. Watson, Sr., chairman of IBM, in a 1943 talk, but there is little evidence that it is anything but a legend.**

- First of all,

**Design for iteration**

*You won't get it right the first time, so make it easy to change.*

Document the assumptions behind the design so that when the time comes to change the design you can more easily figure out what else has to change. Expect not only to modify and replace modules, but also to remodularize as the system and its requirements become better understood.

- *Take small steps. T*he purpose is to allow discovery of both design mistakes and bad ideas quickly, so that they can be changed or removed with small effort and before other parts of the system in later iterations start to depend on them and they effectively become unchangeable. Systems under active development may be subjected to a complete system rebuild every day because the rebuilding process invokes a large number of checks and tests that can reveal implementation mistakes, while the changes that caused the mistakes are fresh in the minds of the implementers.
- *Don't rush*. Even though individual steps may be small, they must still be well planned. In most projects, the temptation is to rush to implementation. With iterative design, that temptation can be stronger, and the designer must make sure that the design is ready for the next step.
- *Plan for feedback.* Include as part of the design both feedback paths and positive incentives to provide feedback. Testers, installers, maintainers, and users of the system can provide much of the information needed to refine it. Alpha testing ("we're not at all sure this even works") and beta testing ("seems to work, use at your own risk") are common examples, and many vendors encourage users to report details of problems and transcripts of failures by e-mail. A well-designed system will provide many such feedback schemes at all levels.
- *Study failures.* An important goal is to learn from failures rather than assign blame for them. Incentives must be carefully designed to ensure that feedback about failures is not ignored or even suppressed by people fearful of being blamed. Then, having found the apparent cause of a failure,

**Keep digging**

*Complex systems fail for complex reasons.*

Computers in the future may weigh no more than 1.5 tons.

— ***Popular Mechanics*** **(March 1949)**

> Continue looking for other contributing or more basic causes. Working systems often work for reasons that aren't well understood. It is common to find that a new release of a system reveals a bug that has actually been in the system for a long time but has never mattered until now. Much can be learned by figuring out why it never mattered. It can also be useful to explore the mindset of the designers to understand what allowed them to design a system that could fail in this way.* Similarly, don't ignore unexplained behavior. If the feedback reports something that now seems not to be a problem or to have gone away, it is probably a sign that something is wrong rather than that the system magically fixed itself.

Iteration sounds like a straightforward technique, but several obstacles tend to interfere with it. The main obstacle is that as a design evolves through a series of iterations, a risk of losing conceptual integrity arises. That risk suggests that the overall plan for the initial, simplest version of the system must accommodate all of the iterations needed to reach the final version (thus the need for foresight). Someone must constantly be on guard to make sure that the overall design rationale remains clear despite changes made during iteration.

In most organizations, good news (e.g., a major piece of the system is working ahead of schedule) flows rapidly throughout the organization, but bad news (e.g., an important module isn't working yet) often gets confined to the part of the organization that discovers it, at least until it can fix the problem and report good news. This phenomenon, the *bad-news diode*, can prevent realization that changing a different part of the system is more appropriate.

A related problem is that when someone finally realizes that the modularity is wrong, it can be hard to change, for two reasons. First, the *unyielding foundations rule* (see page 20) comes into play. Changing modularity by definition involves changing more than one module, and sometimes several. Second, designers who have invested time and effort in developing a module that, from their point of view, is doing what was intended can be reluctant to see this time and effort lost in a rework. Simply put, to change modularity one must deal with both committed components and committed designers.

*The idea of learning from failure and the observation that complex systems fail for complex reasons are the themes of a fascinating book by Henry Petroski, *Design Paradigms: Case Histories of Error and Judgment in Engineering* [Suggestions for Further Reading 1.2.3].

> Based on extensive financial and market analysis, it's projected that no more than five thousand of the new Haloid machines will sell. . . . Model 914, has no future in the office copying market.
>
> **— Consulting firm Arthur D. Little's report to IBM on the prospects for xerographic copying machines (1959)**

A longer-term risk of iteration sometimes shows up when the initial design is both simple and successful. Success can lead designers to be overconfident and to be too ambitious on a later iteration. Technology has improved in the time since deployment of the initial version of the system and feedback has suggested lots of new features. Each suggested feature looks straightforward by itself, and it is difficult to judge how they might interact. The result is often a disastrous overreaching and consequent failure that is so common that it has a name: the *second-system effect*.

Iteration can be thought of as applying modularity to the management of the system design and implementation process. It thus takes us into the realm of management techniques, which are not directly addressed in this book.*

### 1.5.3 Keep it Simple

Remarkably, one of the most effective techniques in coping with complexity is also one that is most difficult to apply: *simplicity*. As Section 1.4.1 explained, computer systems lack natural physical limits to curb their complexity, so the designer must impose limits; otherwise the designer risks being overwhelmed.

The problem with the apparently obvious advice to keep it simple is that

- previous systems give a taste of how great things could be if more features were added.
- the technology has improved so much that cost and performance are not constraints.
- each of the suggested new features has been successfully demonstrated somewhere.
- none of the exceptions or other complications seems by itself to be especially hard to deal with.
- there is fear that a competitor will market a system that has even more features.
- among system designers, arrogance, pride, and overconfidence are more common than clear awareness of the dangers of complexity.

These considerations make it hard to say "no" to any one requirement, feature, exception, or complication. It is their cumulative impact that produces the complexity explosion illustrated in Figure 1.1. The system designer must keep this cumulative

*An excellent book on the subject of system development, by a veteran designer, is Frederick P. Brooks Jr., *The Mythical Man-Month* [Suggestions for Further Reading 1.1.3]. Another highly recommended reading is the Alan Turing Award lecture by Fernando J. Corbató, "On building systems that will fail" [Suggestions for Further Reading 1.5.3].

There is no reason anyone would want a computer in their home.

**— Kenneth Olsen, president of Digital Equipment Corporation (1977)**

impact in mind at all times. The bottom line is that a computer system designer's most potent weapon against complexity is the ability to say, "No. This will make it too complicated."

As we proceed to study specific computer system engineering topics, we shall make much use of a particular kind of simplicity, to the extent that it is yet another design principle:

**Adopt sweeping simplifications**

*So you can see what you are doing.*

Each topic area will explicitly introduce one or more sweeping simplifications. The reason is that they allow the designer to make compelling arguments for correctness, they make detail irrelevant, and they make clear to all participants exactly what is going on. They will turn out to be one of our best hopes for keeping control of complexity.

## WHAT THE REST OF THIS BOOK IS ABOUT

This chapter has introduced some basic ideas that underlie the study of computer systems. In the course of building on these basic ideas, the ensuing chapters explore a series of system engineering topics in the light of three recurring themes:

- The pervasive importance of modularity
- Principle-based system design
- Making systems robust and resilient

Modularity appears in each engineering topic either as one of the goals of that topic or as one of its design cornerstones. Words from chapter titles suggest this theme. *Abstractions and layering* are particular ways to build on modularity. *Naming* is a fundamental mechanism for interconnecting and replacing modules. *Clients and services* and *virtualization* are two ways of enforcing modularity. *Networks* are built on a foundation of modularity. In *fault tolerance*, the module is the unit that limits the extent of failure. A*tomicity* is an exceptionally robust form of modularity that the designer can exploit to obtain *consistency*. Finally, *protection of information* involves further strengthening of modular walls.

The second theme, principle-based system design, has already emerged, both in explicit mention of several principles and in the list of *design principles* on the inside front cover. These principles capture, in brief phrases, widely applicable nuggets of wisdom that have been developed by generations of computer system designers. Later chapters apply these general principles and also introduce additional design principles that are more specific to particular engineering areas. Even with these principles in mind, it is often difficult to offer a precise recipe for design. Therefore throughout the text the reader will find a second form of captured wisdom in the form of several design *hints* that encode rationales for making trade-offs.* Together, the principles and hints suggest that computer system design, though for the most part not based on mathematical theories, is also not completely *ad hoc*: it is actually based on sound principles derived from experience and analysis of both successful and failed systems. The reader who understands and absorbs these principles and hints will have learned much of what this book has to say.

The third theme, making systems robust and resilient, has also already emerged, both in the statement of the *robustness principle* and with the idea that modularity, by limiting interconnections, can help control propagation of effects. The terms *robustness* and *resilience* are informal and overlapping descriptions of a general goal of design: that a system should not be sensitive to modest, long-term shifts in its environment (usually called robustness) and that it should continue operating correctly in the face of transient adversity (usually called resilience). Each succeeding chapter introduces at least one progressively stronger way to make a system more robust and resilient. Thus, the chapter on naming shows how indirection of names can make systems less fragile. Then, the chapters on clients and services and on virtualization demonstrate how to enforce modularity to limit the effects of mistakes and accidents. The chapter on networks introduces techniques that provide reliable communications despite communication failures. The chapter on fault tolerance then generalizes those techniques to make entire systems resilient, even though they contain faulty components. The chapters on atomicity and consistency apply fault tolerance techniques to the particular problem of maintaining the integrity of stored data, despite concurrent activity and in the face of software and hardware failures. Finally, the chapter on protecting information introduces techniques to limit the impact of malicious adversaries who would deliberately steal, modify, or deny access to information.

## EXERCISES

**1.1** True or false? Explain: modularity reduces complexity because

**A.** It reduces the effect of incommensurate scaling.
**B.** It helps control propagation of effects.

*1994–1–3d and 1995-1-1e*

**1.2** True or false? Explain: hierarchy reduces complexity because

**A.** It reduces the size of individual modules.
**B.** It cuts down on the number of interconnections between elements.
**C.** It assembles a number of smaller elements into a single larger element.
**D.** It enforces a structure on the interconnections between elements.
**E.** All of the above.

*1994-1-3c and 1999–1–02*

*Many, if not all, of the hints were originally described by Butler Lampson in his paper "Hints for computer system design" [Suggestions for Further Reading 1.5.4].

**1.3** If one created a graph of personal friendships, one would have a hierarchy. True or false?

*1995–1–1b*

**1.4** Which of the following is usually observed in a complex computer system?

**A.** The underlying technology has a high rate of change.

**B.** It is easy to write a succinct description of the behavior of the system.

**C.** It has a large number of interacting features.

**D.** It exhibits emergent properties that make the system perform better than envisioned by the system's designers.

*2005-1-1*

**1.5** Ben Bitdiddle has written a program with 16 major modules of code. Each module contains several procedures. In the first implementation of his program, he finds that each module contains at least one call to every other module. Each module contains 100 lines of code.

**1.5a** How long is Ben's program in lines of code?

**1.5b** How many module interconnections are there in his implementation? (Each call from one module to another is an interconnection.)

Ben decides to change the implementation. Now there are four main modules, each containing four submodules in a one-level hierarchy. The four main modules each have calls to all the other main modules, and within each main module, the four submodules each have calls to one another. There are still 100 lines of code per submodule, but each main module needs 100 lines of management code.

**1.5c** How long is Ben's program now?

**1.5d** How many interconnections are there now? Include module-to-module and submodule-to-submodule interconnections.

**1.5e** Was using hierarchy a good decision? Why or why not?

*1996–1–2a…e*

**Additional exercises relating to Chapter 1 can be found in the problem sets beginning on page 425.**

# Interface Stability

Mark Kampe

## Interface Specifications

American History books like to credit Eli Whitney with the invention of manufacturing with interchangeable parts. Actually,

- Whitney's parts weren't all that interchangeable,
- he got the idea from a French gunsmith (Honore Blanc),
- the Swedish clock maker Christopher Polhem did a much better job with clock gears 50 years earlier.

But, be that as it may, interchangeable parts revolutionized, and in fact enabled modern manufacturing. The underlying concept is that there be specifications for every part. If each part is manufactured and measured to be within its specifications, then **any** collection of these parts can be assembled into a working whole. This greatly reduces the cost and difficulty of building a working system out of component parts.

The same principle applies to software. If you want to open a file on a Posix system, you use the **open** system call. There may be a hundred different implementations of open but this doesn't matter because they all conform to the same interface specification. The rewards for this standardization are:

a. Your work as a programmer is simplified, because you don't have to write your own file I/O routines.
b. Your program is likely to be easily portable to any Posix compliant system, because they all provide the same file I/O services.
c. Training time for new programmers is reduced, because everybody already knows how to use the Posix file I/O functions.

## The Criticality of Interfaces in Architecture

A system architecture defines the major components of the system, the functionality of each, and the interfaces between them. Clearly the component interface specifications are a critical element of any architecture. To the extent that these interfaces have been well considered and well defined, it should be possible to (at least semi) independently design and implement those components. In principle, any implementations that satisfy those functional and interface specifications should combine to yield a working system.

The converse is also true. To the extent that interfaces between components were poorly considered or specified, it becomes unlikely that independently developed components will work when combined together, and more likely that subsequent changes to one component will break others.

## The Importance of Interface Stability

We write contracts is so that everybody knows what will be expected of them, and what they can expect from the other parties to the agreement. Everybody will depend on you to uphold your obligations under the contract. A software interface specification is a form of contract:

- the contract describes an interface and the associated functionality.
- implementation providers agree that their systems will conform to the specification.
- developers agree to limit their use of the functionality to what is described in the interface specification.

And in return for this they get all the benefits described above.

Suppose that some architectural genius realized that Posix file semantics are too weak to describe the behavior of files in a distributed system, and that this could be solved by adding a new (required) parameter (for a call-back routine) to the Posix open call. What would happen?

- software written to the new semantics would have access to richer behavior, and would function better in cases of node and network failures.
- software written to the new semantics would not work on other Posix compliant systems.
- software written to the old semantics would not work on the new systems.
- customers (knowing nothing about this techno-feud) would be faced with inexplicable failures, and would start complaining to their software and system providers (none of whom were responsible for the change).
- programmers would be confused about which version of open they were using, and would probably get around the problem by writing their own file I/O packages ... a bunch of extra work, but at least it would protect them from future such acts of mischief.

And as a result of this, we would lose all of the benefits described above. When you promise somebody that you will do something (e.g. conform to a specified interface), and they depend on you (e.g. by writing code to that interface), and you don't follow through (e.g. make an incompatible change) ... problems are likely to ensue (e.g. failures, bug reports, product gets a bad reputation, going out of business, etc.).

## Interfaces vs. Implementations

Suppose that someone writes a handy library to efficiently provide reliable communication over an unreliable network, and I decide I want to use it. I download their development kit and start using it, and find it to be great. A few weeks into the project I realize that I need a feature that is not included in their documentation, but after reading their code I discover that the needed feature is actually available. I use it and my product is a great success.

Six months later, they release a new version of their reliable communications library, and my product immediately breaks. After a little debugging I discover that they have changed the undocumented code that I was depending on. It turns out that I had not designed my product to work with their *interfaces*. My product only worked with a particular *implementation* ... and implementations change.

People often put much more work into designing and maintaining their code than to clarifying and maintaining their documentation. This makes it tempting to reverse engineer the interface specifications from a provided implementation. But an interface should exist (and be defined) independently from any particular implementation. Confusing an implementation with an interface usually ends badly!

## Program vs. User Interfaces

Human beings are amazingly robust. We could change the text in dialog boxes, rearrange input forms, add new required input items, and rework all of the menus in a program, and many users would figure out the changes in a few seconds. This inclines many developers to be cavalier in making changes to user interfaces.

Code is nowhere nearly so robust. If I expect my second parameter to be a file name, and you pass me the address of a call-back routine instead, the best we can hope for is a good error message and a quick coredump with no data corruption.

If all users were developers, and only ran their own code, this might be just an irritation. We would get the core dump, track it down, figure out that some idiot had made this change, recode accordingly, and an hour later we'd be good as new. Unfortunately most people run code that they do not understand, and have no way to debug or fix. If a program breaks, all I can do is call support and complain. I am helpless. Users don't care which programmer goofed up. "I bought your product. It doesn't work. I'm taking my business elsewhere!".

If you are going to be delivering binary software to non-developers (that describes 99.999% of all software) you have to trust that what ever platform they run it on will correctly implement all of the interfaces on which your program depends.

The same argument applies, tho not quite as strongly, to independent components in a single system. If components exchange services, and I make an incompatible change to the interfaces of one component, this has the potential to break other components in the same system. I can fix the other components in our system to work with the new changes but:

- this complicates the making of the change.
- we have to ensure that mis-matched component sets are impossible.

## Is every interface graven in stone?

Absolutely not.

You are free to add features that do not change the interface specification (a faster or more robust implementation). In many cases you can add upwards-compatible extensions (all old programs will still work the same way, but new interfaces enable new software to access new functionality).

If the only code that uses my routines is code that I deliver, and I deliver it all in a single package, then I can change my interface at any time.

- customers who have an old package will have the old version of my routine, and all of the code in that package will work with the old routine.
- customers who install the new package will simultaneously get a new version of my routine **and** new versions of all the code that calls it.

But the problem with this is that nobody else is allowed to use my routine.

Another approach would be to send out (to all developers who want to use my routine) an implementation, that they could incorporate into their own products. Later, if I came up with an improved version, I could send that out to all of my developers.

- products written to the old specs would still have an old version of the routine, and simply wouldn't take advantage of my new version.
- newer products, that wanted to take advantage of my new version, could include the new version along with their programs.

It is OK to change interfaces, as long as you can ensure that all clients of the interface will be changed at the same time. It is only when the described module can be delivered independently from the software that uses it that we get into trouble. You simply have to make a choice:

- if you want to be able to change an interface at will, you also have to have control over all of the software that uses the interface.
- if you want to provide an interface to unbundled software (software that can be changed independently) then you must maintain interface stability.

This is a very painful problem, because it would seem to suggest that you have to choose between customer disruption and stifling innovation. Fortunately it is not binary. There is a continuum of stabilities:

- guaranteed for ever
- guaranteed for the life of this product
- guaranteed x months of notice before any incompatible change
- access by special arrangement, which will enable us to manage potentially disruptive changes

- caveat emptor

How stable an interface needs to be depends on how you intend to use it. What is important is that:

1. we be honest about our intentions and set realistic expectations.
2. we have a plan for how we will manage change.

## Interface Stability and Design

So, if we want to expose an interface to unbundled software with high quality requirements, we are not allowed to change it in non-upwards-compatible ways. That sounds like a bother, but if that is the rules, so be it. What has this got to do with architecture and design?

When we design a system, and the interfaces between the independent components, we need to consider all of the different types of change that are likely to happen over the life of this system. When we specify our external component interfaces we need to have high confidence that will be able to accomodate all of the envisioned evolution while preserving those interfaces.

- We might rearrange the distribution of functionality between components to create a simpler (and hence more preservable) interface between them.
- We might design features that we may never implement, just to make sure that the specified interfaces will accomodate them if we ever do decide to build them.
- We might introduce (seemingly) unnatural degrees of abstraction in our services to ensure that they leave us enough slack for future changes.

We must consider which of our interfaces will need to be how stable, and design our system with those stabilities in mind.