Levels of Design

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

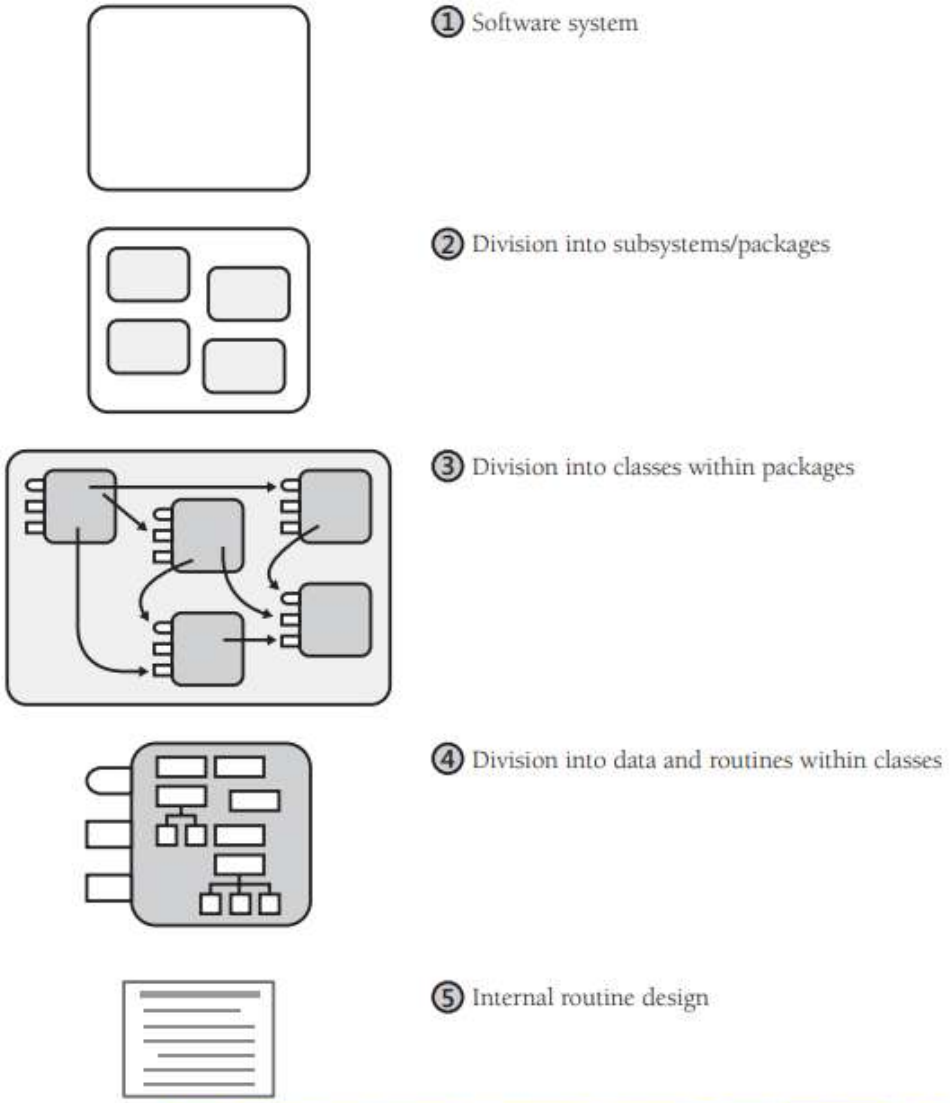

Figure 5-2 The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5).

Level 1: Software System

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

Level 2: Division into Subsystems or Packages

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

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

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

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

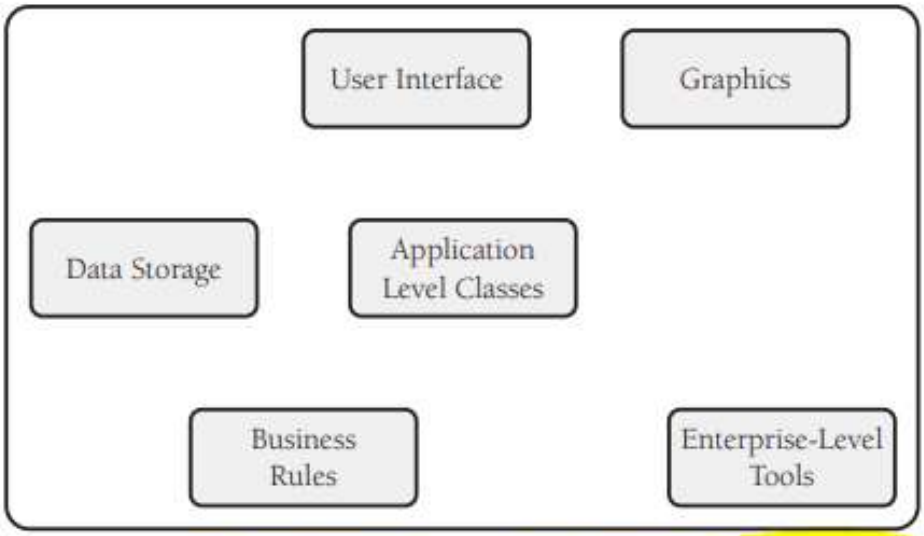

Figure 5-3 An example of a system with six subsystems.

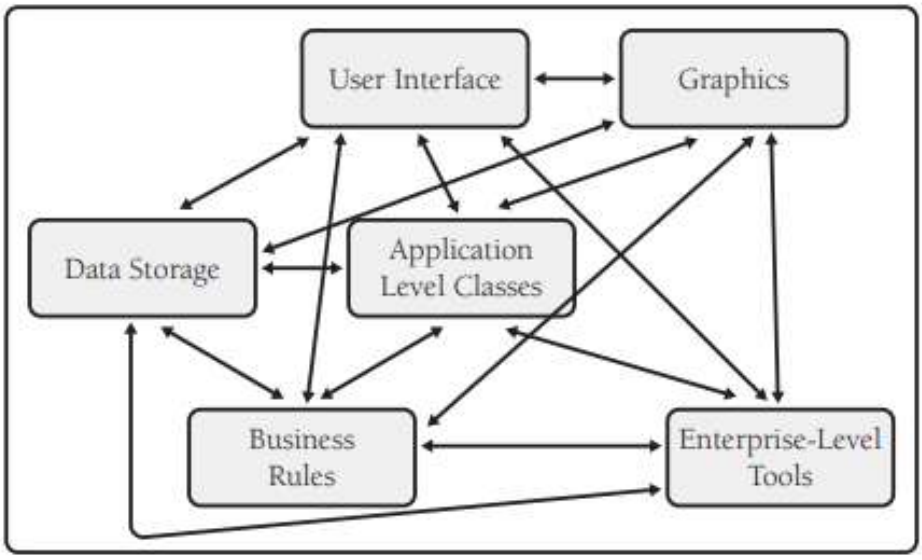

Figure 5-4 An example of what happens with no restrictions on intersubsystem communications.

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

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

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

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

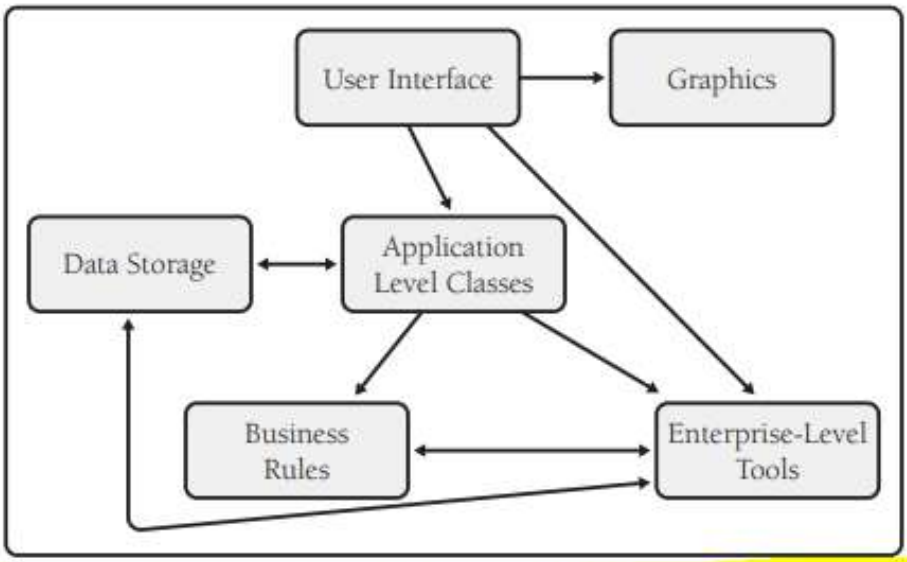

Figure 5-5 With a few communication rules, you can simplify subsystem interactions significantly.

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

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

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

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

Cross-Reference For more on simplifying business logic by expressing it in tables, see Chapter 18, "Table-Driven Methods."

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

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

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

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

Level 3: Division into Classes

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

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

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

Level 4: Division into Routines

Design at this level includes dividing each class into routines. The class interface defined at Level 3 will define some of the routines. Design at Level 4 will detail the class's private routines. When you examine the details of the routines inside a class, you can see that many routines are simple boxes but a few are composed of hierarchically organized routines, which require still more design.

The act of fully defining the class's routines often results in a better understanding of the class's interface, and that causes corresponding changes to the interface—that is, changes back at Level 3.

This level of decomposition and design is often left up to the individual programmer, and it's needed on any project that takes more than a few hours. It doesn't need to be done formally, but it at least needs to be done mentally.

Level 5: Internal Routine Design

Design at the routine level consists of laying out the detailed functionality of the individual routines. Internal routine design is typically left to the individual programmer working on an individual routine. The design consists of activities such as writing pseudocode, looking up algorithms in reference books, deciding how to organize the paragraphs of code in a routine, and writing programming-language code. This level of design is always done, though sometimes it's done unconsciously and poorly rather than consciously and well. In Figure 5-2, design at this level is marked with a 5.