Finally, there's a technique for decoupling modules even further by providing a meeting place where modules can exchange data anonymously and asynchronously. This is the topic of *Blackboards*.

Armed with these techniques, you can write code that will "roll with the punches."

Decoupling and the Law of Demeter

Good fences make good neighbors.
▶ **Robert Frost, "Mending Wall"**

In *Orthogonality*, page 34, and *Design by Contract*, page 109, we suggested that writing "shy" code is beneficial. But "shy" works two ways: don't reveal yourself to others, and don't interact with too many people.

Spies, dissidents, revolutionaries, and such are often organized into small groups of people called *cells*. Although individuals in each cell may know each other, they have no knowledge of those in other cells. If one cell is discovered, no amount of truth serum will reveal the names of others outside the cell. Eliminating interactions between cells protects everyone.

We feel that this is a good principle to apply to coding as well. Organize your code into cells (modules) and limit the interaction between them. If one module then gets compromised and has to be replaced, the other modules should be able to carry on.

Minimize Coupling

What's wrong with having modules that know about each other? Nothing in principle—we don't need to be as paranoid as spies or dissidents. However, you do need to be careful about *how many* other modules you interact with and, more importantly, *how* you came to interact with them.

Suppose you are remodeling your house, or building a house from scratch. A typical arrangement involves a "general contractor." You hire the contractor to get the work done, but the contractor may or may not do the construction personally; the work may be offered to various subcontractors. But as the client, you are not involved in dealing with the subcontractors directly—the general contractor assumes that set of headaches on your behalf.

We'd like to follow this same model in software. When we ask an object for a particular service, we'd like the service to be performed on our behalf. We *do not* want the object to give us a third-party object that we have to deal with to get the required service.

For example, suppose you are writing a class that generates a graph of scientific recorder data. You have data recorders spread around the world; each recorder object contains a location object giving its position and time zone. You want to let your users select a recorder and plot its data, labeled with the correct time zone. You might write

```
public void plotDate(Date aDate, Selection aSelection) {
  TimeZone tz =
     aSelection.getRecorder().getLocation().getTimeZone();
  ...
}
```

But now the plotting routine is unnecessarily coupled to *three* classes—`Selection`, `Recorder`, and `Location`. This style of coding dramatically increases the number of classes on which our class depends. Why is this a bad thing? It increases the risk that an unrelated change somewhere else in the system will affect *your* code. For instance, if Fred makes a change to `Location` such that it no longer directly contains a `TimeZone`, you have to change your code as well.

Rather than digging though a hierarchy yourself, just ask for what you need directly:

```
public void plotDate(Date aDate, TimeZone aTz) {
  ...
}
plotDate(someDate, someSelection.getTimeZone());
```

We added a method to `Selection` to get the time zone on our behalf: the plotting routine doesn't care whether the time zone comes from the `Recorder` directly, from some contained object within `Recorder`, or whether `Selection` makes up a different time zone entirely. The selection routine, in turn, should probably just ask the recorder for its time zone, leaving it up to the recorder to get it from its contained `Location` object.

From The Pragmatic Programmer
by Andrew Hunt and David Thomas
2000

Traversing relationships between objects directly can quickly lead to a combinatorial explosion[1] of dependency relationships. You can see symptoms of this phenomenon in a number of ways:

1. Large C or C++ projects where the command to link a unit test is longer than the test program itself
2. "Simple" changes to one module that propagate through unrelated modules in the system
3. Developers who are afraid to change code because they aren't sure what might be affected

Systems with many unnecessary dependencies are very hard (and expensive) to maintain, and tend to be highly unstable. In order to keep the dependencies to a minimum, we'll use the *Law of Demeter* to design our methods and functions.

The Law of Demeter for Functions

The Law of Demeter for functions [LH89] attempts to minimize coupling between modules in any given program. It tries to prevent you from reaching into an object to gain access to a third object's methods. The law is summarized in Figure 5.1 on the next page.

By writing "shy" code that honors the Law of Demeter as much as possible, we can achieve our objective:

> **Tip 36**
>
> Minimize Coupling Between Modules

Does It Really Make a Difference?

While it sounds good in theory, does following the Law of Demeter really help to create more maintainable code?

Studies have shown [BBM96] that classes in C++ with larger *response sets* are more prone to error than classes with smaller response sets (a

1. If n objects all know about each other, then a change to just one object can result in the other $n-1$ objects needing changes.

Figure 5.1. Law of Demeter for functions

```
class Demeter {
private:
   A *a;
   int func();
public:
   //...
   void example(B& b);
}

void Demeter::example(B& b) {
   C  c;
   int f = func();          // itself

   b.invert();              // any parameters that were passed in to the method

   a = new A();
   a->setActive();          // any objects it created

   c.print();               // any directly held component objects
}
```

The Law of Demeter for functions states that any method of an object should call only methods belonging to:

response set is defined to be the number of functions directly invoked by methods of the class).

Because following the Law of Demeter reduces the size of the response set in the calling class, it follows that classes designed in this way will also tend to have fewer errors (see [URL 56] for more papers and information on the Demeter project).

Using The Law of Demeter will make your code more adaptable and robust, but at a cost: as a "general contractor," your module must delegate and manage any and all subcontractors directly, without involving clients of your module. In practice, this means that you will be writing a large number of wrapper methods that simply forward the request on to a delegate. These wrapper methods will impose both a runtime cost and a space overhead, which may be significant—even prohibitive—in some applications.

As with any technique, you must balance the pros and cons for *your* particular application. In database schema design it is common practice to "denormalize" the schema for a performance improvement: to

Physical Decoupling

In this section we're concerned largely with designing to keep things logically decoupled within systems. However, there is another kind of interdependence that becomes highly significant as systems grow larger. In his book *Large-Scale C++ Software Design* [Lak96], John Lakos addresses the issues surrounding the relationships among the files, directories, and libraries that make up a system. Large projects that ignore these *physical design* problems wind up with build cycles that are measured in days and unit tests that may drag in the entire system as support code, among other problems. Mr. Lakos argues convincingly that logical and physical design must proceed in tandem—that undoing the damage done to a large body of code by cyclic dependencies is extremely difficult. We recommend this book if you are involved in large-scale developments, even if C++ isn't your implementation language.

violate the rules of normalization in exchange for speed. A similar trade-off can be made here as well. In fact, by reversing the Law of Demeter and *tightly* coupling several modules, you may realize an important performance gain. As long as it is well known and acceptable for those modules to be coupled, your design is fine.

Otherwise, you may find yourself on the road to a brittle, inflexible future. Or no future at all.

Related sections include:

- *Orthogonality*, page 34
- *Reversibility*, page 44
- *Design by Contract*, page 109
- *How to Balance Resources*, page 129
- *It's Just a View*, page 157
- *Pragmatic Teams*, page 224
- *Ruthless Testing*, page 237

Challenges

- We've discussed how using delegation makes it easier to obey the Law of Demeter and hence reduce coupling. However, writing all of the methods needed to forward calls to delegated classes is boring and error prone. What are the advantages and disadvantages of writing a preprocessor that generates these calls automatically? Should this preprocessor be run only once, or should it be used as part of the build?

Exercises

24. We discussed the concept of physical decoupling in the box on on the facing page. Which of the following C++ header files is more tightly coupled to the rest of the system? *Answer on p. 293*

person1.h:

```
#include "date.h"
class  Person1 {
private:
  Date myBirthdate;
public:
  Person1(Date &birthDate);
  // ...
```

person2.h:

```
class Date;
class  Person2 {
private:
  Date *myBirthdate;
public:
  Person2(Date &birthDate);
  // ...
```

25. For the example below and for those in Exercises 26 and 27, determine if the method calls shown are allowed according to the Law of Demeter. This first one is in Java. *Answer on p. 293*

```
public void showBalance(BankAccount acct) {
  Money amt = acct.getBalance();
  printToScreen(amt.printFormat());
}
```

26. This example is also in Java. *Answer on p. 294*

```
public class Colada {
  private Blender myBlender;
  private Vector myStuff;

  public Colada() {
    myBlender = new Blender();
    myStuff = new Vector();
  }
  private void doSomething() {
    myBlender.addIngredients(myStuff.elements());
  }
}
```

27. This example is in C++. *Answer on p. 294*

```
void processTransaction(BankAccount acct, int) {
  Person *who;
  Money amt;

  amt.setValue(123.45);
  acct.setBalance(amt);
  who = acct.getOwner();
  markWorkflow(who->name(), SET_BALANCE);
}
```