Minimize Dependencies

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

```java
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 through a hierarchy yourself, just ask for what you need directly:

```java
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.

Traversing relationships between objects directly can quickly lead to a combinatorial explosion[1] of dependency relationships.
The Law of Demeter

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.

Figure 5.1. Law of Demeter for functions

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

class Demeter {
  private:
    C  c;
    int func();
  public:
    // ...
    void example(B& b);
};

void Demeter::example(B& b) {

  // itself
  int f = func();

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

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

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

The Pragmatic Programmer(c) From Journeyman to Master
Authors: Hunt A., Thomas D.