CHAPTER 1

Preliminaries: OO and Design Patterns 101

Normally a book of this sort would start with a quote from Christopher Alexander, the architect (of buildings) who came up with the notion of a design pattern. I've found that though Alexander is a brilliant man who writes wonderful books, his prose can be a bit opaque at times, so I'll skip the mandatory quote. His ideas launched the entire design-pattern movement, however.

Similarly, the seminal book on design patterns in software is Gamma, Helm, Johnson, and Vlissides's *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995). (The four authors are jokingly called the *Gang of Four* by most working designers.) My book wouldn't exist if the Gang-of-Four book hadn't been written, and I (and OO programmers in general) owe an enormous debt of gratitude to the authors. Nonetheless, the Gang-of-Four book is a formal academic presentation of patterns, and most beginners find it too dense to penetrate. At the risk of losing some academic precision, I'll take a kinder and gentler approach.

Patterns vs. Idioms

Let's start exploring the notion of a *pattern* by discussing simple programming idioms. Many design patterns are used so commonly that, in many programmers' minds, they cease to be patterns at all but are idioms of the language. That is, you don't think of these patterns as anything special—they're just "how things are done." Some people distinguish between patterns and idioms based on usage (for example, a pattern is represented in a formal way, and an idiom isn't). I don't see a distinction, however. An idiom is just a pattern, the use of which has become commonplace.

Derivation is a great example of the evolution of pattern to idiom. Back in the early 1980s when C was king, derivation was a design pattern. You can find several examples of an "extends" relationship in C. For example, the standard implementation of `malloc()` uses a header (the base class) that's extended to create another struct (the derived class), which effectively inherits the `free()` method from the base class.

Abstract functions were also part of the Derivation pattern. It was commonplace in C to pass around tables of function pointers, initialized differently for different "classes." This is exactly how C++ implements both abstract methods and interface inheritance, but back in the C world, we didn't have a name for it.

Derivation wasn't built into C, and most C programmers weren't programming in an object-oriented way, so Derivation was not a programming idiom—it was a pattern. It was something you saw in many programs that had to solve similar problems, but it wouldn't occur naturally to your average C programmer.

Nowadays, of course, derivation and interfaces are just built into the language; they've become idioms.

So What *Is* a Design Pattern, Anyway?

Design patterns are, first and foremost, discovered, not invented. When Christopher Alexander looked at many successful buildings, concentrating on one aspect of that building (such as what makes a room "pleasant"), certain *patterns started to emerge.* Successful "pleasant" rooms tend to solve certain classes of problems (such as lighting) in similar ways. By the same token, when you look at several programs written by diverse programmers, and when you focus on a particular implementation problem that those programs must solve (isolating subsystems, for example), patterns start to emerge there as well. You find that several programmers independently develop similar techniques to solve similar problems. Once you're sensitive to the technique, you tend to start seeing patterns everywhere you look. It's not a pattern, though, unless you find it in several independently developed programs. It's a sure sign that authors don't know what they're talking about when they say, "We've invented a design pattern that...." They may have come up with a design, but it's not a pattern unless several people invent it independently. (It's possible, of course, for an invented "pattern" to become a real pattern if enough people adopt it.)

A design pattern, then, is a general technique used to solve a class of related problems. It isn't a specific solution to the problem. Probably every architect who came up with an observably pleasant room brought light into that room in a different way, and probably every programmer implemented their solution differently. The pattern is the general structure of the solution—a "metasolution" if you will—not the solution itself.

You can find a good analogy in music. You can think of the notion of "classical music" as a compositional pattern. You can identify music that fits the "classical music" pattern because it sounds like classical music. The individual pieces are quite different, however.

Given the broad nature of a pattern, you can't cut-and-paste a design pattern from one program to another (though you *might* be able to reuse a specific solution if the current context is similar to the original one). This particular issue is an enormous point of confusion amongst people new to patterns. Judging by the comments I've seen on the web, many programmers seem to think that if a book doesn't present the same examples as the Gang-of-Four book, the author doesn't understand patterns. This attitude simply shows that the person who wrote the comment doesn't understand patterns; they've confused the piece of code that demonstrates the pattern with the pattern itself. For that reason, I'll try to give several different examples for each of the patterns I discuss so you can see how the pattern relates to disparate concrete implementations—and I won't use the Gang-of-Four examples unless they're relevant to real programming issues (many aren't).

To make things more complicated, the actual objects and classes that participate in a pattern almost always participate in other patterns at the same time. Focus on it one way, and it looks like one thing; change your focus, and it looks like something else. To make things even more confusing, many pattern implementations share identical static structures. When you look at the UML static-structure diagrams in the Gang-of-Four book, they all look the

same: You'll see an interface, a client class, and an implementation class. The difference between patterns lies in the dynamic behavior of the system and in the intent of the programmer, not in the classes and the way they interconnect.

I'll try to illustrate these problems with an example from the architecture of buildings, focusing on two domains: ventilation and lighting.

In the ventilation domain, I don't want a room to feel "stuffy." Looking at several rooms that indeed are comfortable, a pattern, which I'll call *Cross Ventilation*, emerges. The rooms that participate in this pattern have an air source and an air exit directly across from one another at window height. Air enters at the source, flows across the room, and then leaves from the exit. Having identified (and named) the pattern, I create a capsule description—called the *intent* by the Gang of Four—that summarizes the general problem and the solution addressed by the pattern. In the case of Cross Ventilation, my intent is to "eliminate stuffiness and make a room more comfortable by permitting air to move directly across the room horizontally, at midbody height." Any architectural mechanism that satisfies this intent is a legitimate reification (I'll explain that word in a moment) of the pattern. (The Gang of Four's use of the word *intent* in this context is pretty strange. I don't use it much in this book, preferring words such as *purpose*.)

Reification is an obscure word, but I've found it pretty handy. It's not commonly used in the literature, however. Literally, *to reify* means "to make real." A reification of an idea is a concrete realization of that idea, and a given idea may have millions of possible reifications. I use *reify*, rather than some more commonplace word, to emphasize what a pattern isn't. A pattern is not "instantiated," for example. Every instantiation of a class is identical (at least in structure) to every other instantiation. This isn't so with a design pattern. Similarly, a reification is not an "implementation" of a pattern—the reification of a pattern is a design, not code, and a given design has many possible legitimate implementations.

So, what are some of the reifications of Cross Ventilation? You could have a window across from a window, a window across from a door, two doors across from each other, a window across from "negative" ventilator that sucked in air, input and output ventilators on opposite walls, or a huge bellows operated by an orangutan jumping up and down on it across from a gaping hole in the other wall. In fact, you don't even need walls: A room with no walls at all on two opposite sides would fit the pattern. A given pattern has myriad reifications.

Though there's a lot of flexibility in reifying the pattern, you can't pick and choose the attributes you like. For example, simply having air entrances and exits isn't sufficient if the height and directly-across-from requirements aren't met. Putting the entrance and exit in the ceiling, for example, isn't a legitimate reification of the pattern (as any of us who occupy stuffy big-building offices with ceiling ventilators can attest).

To summarize, the intent of Cross Ventilation is to "eliminate stuffiness and make a room more comfortable by permitting air to move directly across the room horizontally, at midbody height." The *participants* in the pattern, be they windows, doors, or orangutans, have the *roles* of air entrance and exit.

Moving to the lighting domain: After looking at many rooms I notice that the most pleasant rooms have windows on two adjacent walls. That's why corner offices are so desirable: The multidirectional natural-light source makes the room seem more pleasant. Dubbing this pattern *Corner Office*, I come up with the following intent: I intend to "make a room more pleasant by locating two sources of natural light on two adjacent walls." Again, there are a myriad reifications: windows on two walls, windows on one wall and French doors on the other, French doors on two walls. You could argue that windows on one wall and mirrors on an adjacent wall would also fit since the reflected natural light does serve as a light source. If I were Bill Gates, I could put a window on

one wall and a 600-inch plasma display that showed what you'd see if the wall wasn't there on the other, but that's not a legitimate reification because the plasma display isn't "natural light." You have, of course, millions of ways to implement the Window and French Door patterns as well.

Now let's consider a concrete design—the plans for a building. Figure 1-1 shows reifications of both Cross Ventilation and Corner Office in a single design. I've put both an architectural diagram and the equivalent UML in the figure. Patterns are identified using UML 1.5's *collaboration* symbol. The pattern name is put into an oval, with dashed lines extending to the classes that participate in the patterns. The lines are annotated with the role that that class plays within the pattern.

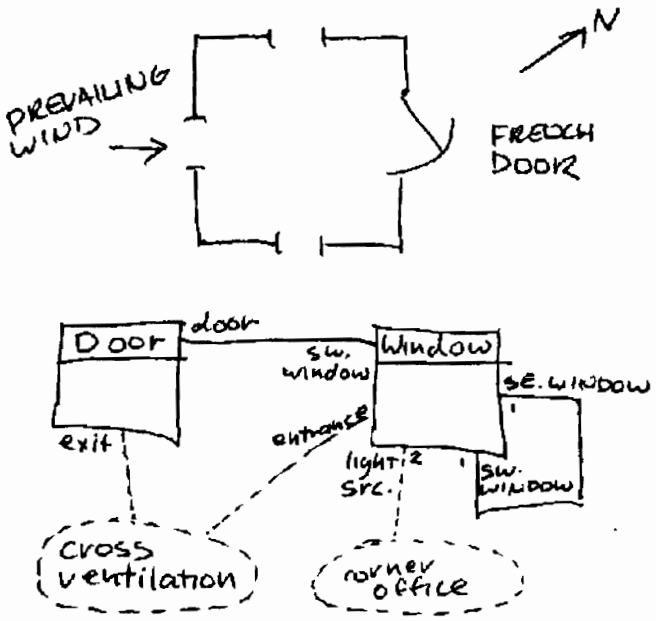

Figure 1-1. *Combined reification of Cross Ventilation and Corner Office*

The southwest window serves as an air entrance in Cross Ventilation, and the door across from it serves as an exit. The other two windows don't participate in Cross Ventilation since the prevailing wind is from the southwest. Refocusing, the southwest and southeast windows participate in Corner Office as the two light sources. Neither the door nor the northwest window is a participant since they aren't significant sources of light. That southwest window is interesting because it participates in two patterns simultaneously. It has the role of "air source" in Cross Ventilation and "light source" in Corner Office. The objects and classes that participate in various patterns often intermesh in this way.

It's critical to note that there's no way to identify the patterns simply from structure. For example, the wind may be blocked by another structure, in which case none of the windows can be an air entrance. By the same token, one of the windows may be two feet away from the blank wall on the building next door or look onto a hallway, so it wouldn't be a significant light source (though it could be an air entrance or exit). As you'll see when you start looking at the actual patterns, you need contextual information—including the intent of the architect—to

identify a design pattern in a computer program. You can't just look at a UML diagram and identify all the patterns. You have to know the intended use of the objects or classes. You'll see many examples of this phenomenon in the examples in subsequent chapters.

Reopening the cut-and-paste issue, I'm hoping you can now see how a pattern can be reified into a vast number of designs, each of which could be implemented in myriad ways. To say that you can cut-and-paste a pattern in a design tool is nonsensical. Nonetheless, many object-oriented CASE tools claim to have "pattern libraries," from which you can insert patterns into your designs. In practice, these libraries contain prebuilt UML structure for the single reification of a given pattern that's presented in the Gang-of-Four book. Though pasting one of these structures into your design can be useful at times, don't confuse this "paste" operation with actually using a pattern in a design. A good design almost always must use a custom reification that's appropriate in context. The mindless cut-and-paste approach is no more designing than paint-by-numbers is painting.

So, What's It All Good For?

So, if patterns are so amorphous, what are they good for?

When I first read the Gang-of-Four book, I was unimpressed. It seemed like nothing but a pedagogic presentation of stuff that most competent designers had already discovered, usually by beating their heads against brick walls trying to find elegant solutions to the problems that the patterns addressed. True, had I read the book a few years earlier, my head would have many fewer bumps on it, but the whole thing seemed to be much ado about nothing.

I thought that way until the first time I needed to discuss a project with another designer. He pointed at a piece of the design and said, "These interfaces comprise a bridge between these two subsystems; the bridge itself is implemented with this set of object adapters." I was struck with the economy of what just happened. In two sentences, he had eliminated probably half an hour of elaborate explanation. Maybe there was something to all this pattern stuff after all.

Then I went to a presentation at the first Java One, where all of AWT was described in terms of patterns. The description was both short and lucid—much shorter and clearer, in fact, than could possibly have been the case had the speaker not taken a patterns approach.

I went back and reread the book before starting my next design project and then consciously tried to think of my next design in terms of the patterns. That is, I started asking myself, "What am I trying to accomplish here, and are there any patterns that address this problem?" (using the purpose section of the pattern description to see what was relevant). When the answer was "yes," I used the pattern right off the bat. I found that taking this approach noticeably shortened the design time and that the resulting design was better quality as well. The better I knew the patterns, the faster things went. Moreover, my initial design needed much less refinement than usual to be acceptable.

I was hooked.

The patterns provide an organizational framework that vastly improves communication, which in the long run is what design is all about. Conversations that previously took hours could happen in a few minutes, and everyone could get more real work done in less time. I went back and read everything about patterns that I could lay my hands on and discovered that the Gang-of-Four book just scratched the surface. Hundreds of documented patterns were out there on the web and in the literature, and many of these were applicable to work I was doing. In practice, I've found that a solid familiarity with the patterns that are relevant to

my work have made that work go much faster and given me much better results. (By "solid," I mean that you know the stuff cold—you don't have to look things up in a book.)

The Role of Patterns in Design

When do patterns come up in the design process, and what role do they play in design? The answer to this question varies with the methodology you're using—I hope that you *do* use a methodology—but design patterns are of interest primarily at the implementation level, so they start coming up when you start thinking about implementation. The deeper question then is, when does analysis (which concerns itself with the problem domain) stop and design (which concerns itself with implementation) begin?

The best analogy that I know is in the design and construction of buildings. The plans of a building don't show every construction detail. They show where the walls go, but not how to build a wall. They show where the plumbing fixtures go, but not how to route pipes. When the building is constructed, design activities involving wall construction and pipe routing *do* happen, but the artifacts are rarely kept since the implementation speaks for itself. A carpenter, for example, may use a "stud-placement" pattern to build a strong wall. The design shows where the wall goes, but not how to build the wall.

Moving the analogy to software: In most projects, design activities should stop when you get the point that a *good* programmer can implement without difficulty. I would never consider putting the mechanics of creating a window with Swing into a design. That's just something that the programmer should know how to do, and if the code is written up to professional standards (well-chosen names, good formatting, comments where necessary, and so on), the implementation choices should be self-documenting.

Consequently, design patterns are often not spelled out in detail in the design documents but, rather, represent decisions that the implementer makes. Patterns applied by an implementer are rarely documented in depth, though the name of the participants (or other comment) should identify what's going on. (For example, `WidgetFactory` reifies Factory).

Of course, exceptions exist to this don't-design-patterns rule. The software equivalent of the windows used in the Corner Office pattern may well appear in the design documents (which show you where to place the windows). Similarly, very complex systems, where much more detail is required in the design (in the same way that the architectural plans for a skyscraper are more detailed than those of a small house), often document the patterns in depth.

The Tension Between Patterns and Simplicity

A related issue is the complexity that patterns tend to introduce into a system. If "foolish consistency is the hobgoblin of little minds," unnecessary complexity is the hobgoblin of bad programmers. Just like Emerson's "little statesmen and philosophers and divines" who adore consistency, many "little" programmers and architects think that patterns are good for their own sake and should be used at every possible opportunity. That mindless approach almost guarantees a fragile, unmaintainable mess of a program. Every pattern has a downside that serves as an argument for not using it.

Simple systems are easier to build, easier to maintain, smaller, and faster than complex ones. A simple system "maximizes the work done," by increasing "the amount of work not done." A program must do exactly what's required by the user. Adding unasked-for functionality dramatically increases development time and decreases stability.

Simplicity is often not an easy goal to achieve. Programmers love complexity, so they have a strong tendency to overcomplicate their work. It's often easier to quickly build an overly complex system than it is to spend the time required to make the system simple. Programmers who suspect that requirements will be added (or change) over time tend to add support for requirements that *may* exist in the future. It's a bad idea to complicate the code because you think that something *may* have to change in the future, however. (Whenever I try to predict the future, I'm wrong.) Programmers need to write the code in such a way that it's easy to add new features or modify existing ones, but not add the features now.

The flip side of this problem is oversimplification of an inherently complex problem. You really want to do "exactly" what's needed; removing required functionality is as bad as adding unnecessary functionality. One example of oversimplification is an "undo" feature. Alan Cooper—the inventor of Visual Basic and well-known UI guru—argues that you never want to ask users if they *really* want to do something. Of course they do—why else would they have asked to do it in the first place? How many times have you *not* deleted a file because that stupid confirmation dialog pops up? The best solution to the unwanted deletion or similar problem is to do what the user asks but then provide a way to undo it if the user makes a mistake. That's what your editor does, for example. (Imagine an editor that asked, "Do you *really* want to delete that character?") Undo is hard to implement, however, and a tendency exists to disguise laziness in the garb of simplicity. "A complete undo system adds too much complexity, so let's just throw up a confirmation dialog."

These three requirements—simplicity, completeness, and ease of modification—are sometimes at odds with one another. The patterns described in this book help considerably when it comes time to change or add something, but by the same token, the patterns complicate the code. Unfortunately, no hard-and-fast rule describes when using a pattern is a good idea—it's a seat-of-the-pants judgment call on the part of the programmer. A sensitive seat comes from experience that many designer/programmers simply don't have (and, as Ken Arnold—coauthor of the original book on Java programming—points out, from a sense of aesthetics that many don't cultivate.) Thus, you end up with bad programs that use design patterns heavily. Simply using patterns doesn't guarantee success.

On the other hand, the building blocks of patterns, such as the heavy use of interfaces, are always worth incorporating into the code, even when a full-blown pattern is inappropriate. Interfaces don't add much complexity, and down-the-line refactoring is a lot easier if the interfaces are already in place. The cost of doing it now is low, and the potential payoff is high.

Classifying Patterns

It's sometimes useful to classify patterns in order to make it easier to choose appropriate ones. Table 1-1, taken from the Gang-of-Four book, shows you one way to look at the Gang-of-Four patterns. But you can also create similar tables of your own that categorize the patterns in different ways, however.

The Gang of Four broke the patterns into two scopes: Class patterns require implementation inheritance (`extends`) to be reified. Object patterns should be implemented using nothing but interface inheritance (`implements`). It's not an accident that there are many more Object than Class patterns. (You'll find more on this issue in the next chapter.)

Within a scope, the patterns are further divided into three categories. The Creational patterns all concern themselves with object creation. For example, Abstract Factory provides you with a means of bringing objects into existence without knowing the object's actual class

name. (I'm simplifying here, but I'll explain this notion in depth later in the book.) The Structural patterns are all static-model patterns, concerned with the structural organization of your program. For example, Bridge describes a way to separate two subsystems from each other so that either subsystem can be modified without affecting the other. The Behavioral patterns are all dynamic-model patterns, addressing the way that various objects will interact at runtime. Chain of Responsibility, for example, describes an interobject message-passing system that allows a message to be fielded by the particular object that knows how to deal with it. You don't have to know which object that will be at compile time—it's a runtime decision.

Table 1-1. *The Gang-of-Four Design Patterns Classified*

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Class Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Facade Flyweight Object Adapter Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

I'll cover all these patterns in depth (though not in order), but bear in mind that there are many other pattern categories than the ones identified by the Gang of Four. Examples include real-time programming patterns, threading patterns, Java Enterprise JavaBean (EJB) patterns, and so forth.

One other issue is the interdependence between patterns. For example, as you'll see later in the book, Command appears in one form or another in most of the other Behavioral patterns. The Gang-of-Four book includes a diagram showing these dependency relationships, but frankly, the diagram looks like a mess of spaghetti and is of little practical use. The main thing to remember is that the various patterns are indeed related to each other, sometimes in significant and intricate ways.

If you have trouble distinguishing one pattern from another, you aren't alone. Most often the confusion is caused precisely because of the natural interdependence of patterns. My advice is to focus on the intent/purpose section of the pattern description—remember, any reification that satisfies the designer's intent is legitimate. Looking solely at the structure—natural for a programmer—often adds confusion instead of clarity. You'll find, for example, that the patterns in the Structural category have almost identical static structures, but these structures are used toward profoundly different ends. The patterns are as much about communication as about software, so don't focus solely on the software issues.

On Design, Generally

The other main preliminary topic I have to discuss before leaping into the patterns themselves is OO design generally.

First, Object-Oriented Design (OOD) and Object-Oriented Programming (OOP) are very different things. The design process starts at requirements gathering, involves an orderly progression through tasks such as use-case analysis, and arrives, eventually, at a design from which you code. The programming process starts with the design or some portion of the design, and using concepts such as derivation, encapsulation, and design patterns results in a computer program—a realization of the design. Many people confuse programming with design. Simply because you've used Java for six years, understand subclassing, and can write 1,000 lines of debugged code a day doesn't mean that you know OOD. In fact, the contrary is more likely: many spectacularly good programmers don't understand the basic principles of OOD.

A good analog is in the building trades. Buildings are *designed* by architects, and they're *built* by contractors. In the same way, OO systems are designed by OO designers and implemented by OO programmers. These two roles can be filled the same people, but often aren't. Architects have to know how to construct a building, or they couldn't come up with a workable design. Contractors, on the other hand, don't have to have much understanding at all of what architects do. (This isn't to say that there aren't architects who will happily design buildings that can't be built or lived in or that there aren't contractors who can easily identify a bad design when they see it.) The best programmers are also good architects, and the best architects are good programmers. This melding of skills is particularly important in the now-fashionable Agile methodologies, where design and coding go on in parallel. No Agile methodology supports the notion of a puppet-master architect who pulls all the strings while the programmers dance.

That being said, many programmers are experienced craftsmen and craftswomen who produce beautiful code but don't understand the design process at all—they're builders, not designers. Please don't think that I'm in any way degenerating the considerable skills of a good builder, but the ad-hoc designs that these programmers come up with are often less than ideal.

A recent Standish Group report, which looked at thousands of programming projects over multiple years, determined that roughly 72 percent of software projects were failures. The lack of up-front design, and everything that entails (requirements gathering, for example), was pegged the primary cause of this failure. That is, even skilled architects can fail when they abandon the architectural process.

This book is about OO programming and architecture, not about process. Design patterns are typically implementation details that are applied by OO programmers when they translate the initial design to code. You can't arrive at a reasonable design, however, without using a reasonable process. (The Agile processes are certainly reasonable.) You can't arrive at reasonable code without the benefit of a reasonable design (which may evolve). Simply applying design patterns to your code in an ad-hoc way will not make your programs significantly better and may make them worse. Unnecessary complexity—and many patterns are complex—never improves anything.

So, please don't confuse the topics discussed elsewhere in this book with the OOD process as a whole. Patterns are just a small part of the puzzle—and in some ways an insignificant part. This isn't a book about OOD—it's a book about moving an OO design toward a concrete implementation. To really apply design patterns effectively, you need to know how to design. You

need to know the process. I've listed several books on the subject of design on the web page mentioned in the preface, and I recommend you peruse them.

Programming FORTRAN in Java

Given that this book takes a hard-line attitude toward OO programming, it seems worthwhile to discuss the differences between OO and procedural approaches at the system (as compared to structural) level. Procedural approaches to programming can be characterized as "data-centric." A procedural program is structured around the flow of data between subroutines that manipulate or examine that data. The database is central to the design of the program; in fact, many procedural programs do little beyond exposing database tables via a nice user interface. Procedural systems tend to be heavily hierarchical, centered on the notion of "global control." A global entity (a subroutine toward the top of a hierarchy) performs work on data that it collects from elsewhere—either from subroutines beneath it in the hierarchy or by harvesting global data created earlier. The main disadvantage of procedural systems is in debugging and maintenance. The shared data creates "coupling" relationships (undesirable dependencies) between subroutines. When you change one subroutine, you affect others. In extreme cases, the effects of a seemingly trivial change could take months to become clear and to fix.

Object-oriented systems, on the other hand, are networks of intercooperating agents that communicate by means of some messaging system. The objects are peers—there's no one object that's clearly in charge, issuing directives to the other objects. I'll discuss the characteristics of a well-done object throughout the remainder of this chapter, but a few broad principles are worth introducing now. Looking at an object from the outside, you should have no idea how it's implemented. It should be possible to replace the entire implementation without affecting any of the client objects (objects that use the one you've just changed). Though objects sometimes pass other objects to each other, data doesn't flow through the system in a procedural sense. An object jealously guards its data and performs operations on that data in response to receiving some message. Objects don't give the data to other objects unless absolutely necessary, and then, the data is itself encapsulated in another object. These two concepts (*implementation hiding* and *data abstraction*) are key.

One good way to tell the difference between an object-oriented and procedural system is to note what happens when you change something. In procedural systems, changes tend to "ripple out" into the rest of the program; large changes in behavior typically require widespread modification of the code. Object-oriented systems tend to concentrate changes into one place. A single change in the code tends to make large changes in program behavior. For example, if you need to change a data format used for persistent storage, procedural systems often must be changed in several places because each procedure is responsible for parsing the data. In an OO system, you'd change the object that's stored persistently, and that's it.

Of course, OO principles such as data abstraction (hiding the way that a bunch of functions work by hiding the data structures from the users of those functions) have been around for a long time and are the foundation of any quality programming—procedural or otherwise. The C language file-I/O system and Ken Arnold's Curses library are both object oriented, for example. A procedural system can look object oriented in places. A "pure" OO system is characterized primarily by the consistent and meticulous use of concepts such as data abstraction.

OO systems have other key differences from procedural ones. For example, object-oriented systems tend to be models of real-world processes. This train of thought gets you into the entire OOD process, however, and this book is primarily about OO structure, so I won't follow this avenue further.

Unfortunately, many people who grew up in a procedural world think of an OO approach to a problem as wrong, not different. I'm always flabbergasted by the controversy that my articles on OO technique seem to engender. When I published (in the online magazine JavaWorld) early drafts of the sections of this book, I was shocked by the invective that was hurled at me for discussing far-from-earth-shattering concepts—concepts that have been tossed around in the literature for 30 years. I was called "incompetent," "clueless," "a shyster," " a dunderhead," and various other epithets that aren't polite to print. My articles are "badly thought out" and "tosh." One reader actually threatened physical violence, titling an invective-filled epistle (which the site removed) with "THIE [*sic*] AUTHOR SHOULD BE SMACKED AROUND WITH A PIPE!"

Don't confuse "familiar" with "correct." Many programmers assume that the libraries they use regularly are "right," and if that library does things in a certain way, then that library sets a standard. This disease is particularly prevalent with people who learn programming from how-to books focused on particular tasks. If the only architecture they've ever seen is EJB and Struts, they'll tend to classify everything that doesn't look like EJB and Struts as bad. Just because we've done things historically in a particular way doesn't mean that that's the best way to do things; otherwise, we'd all still be programming in assembly language.

I had an interesting discussion many years ago with the person who led Microsoft's C++ and Foundation Class (MFC) efforts. When I brought up that MFC wasn't particularly object oriented, his response was that he was well aware of that fact, but most of the people who programmed Microsoft systems didn't understand OO concepts. It wasn't Microsoft's job to teach OO, he said. Consequently, Microsoft deliberately created a procedural system in C++, because that system would be "easier to understand." That OO-is-hard-to-understand philosophy is still dominant at Microsoft. The .NET APIs are procedural in structure, for example, and C# has language features that encourage procedural thinking. So, it's not surprising to find Microsoft applications that don't follow some of the basic principles of OO systems. Many Microsoft programmers seem to take violent exception to any OO practice that doesn't jibe with the way .NET does things, however. They're confusing "familiarity" with "correct."

Please don't try to apply procedural thinking to OO systems, and don't criticize an OO technique that I'm describing simply because the approach isn't procedural. Many common OO notions simply aren't embodied in a lot of existing code that you may have seen. Saying that some coding practice isn't viable in an OO system isn't the same as saying that code that uses those practices is never viable. I'm not going to bring this point up every time I discuss an OO approach to a problem, however.

Finally, bear in mind that a "pure" OO solution isn't always required. As is the case with most design issues, there are always trade-offs and risks. For example, a simple web site that's using Servlets to put a thin front end on a database probably doesn't need to be object oriented. The risk is that, as the small program evolves, it turns into a mass of unmaintainable spaghetti code. Similarly, many programmers don't understand OO concepts, so if your system doesn't have a significant long-term maintainability requirement and if business requirements are not likely to change, assigning a programmer to it who quickly implements a procedural solution isn't necessarily a bad decision. The risk, of course, is that the lifetime of that program is longer than you expect, or that significant changes to the business rules indeed occur, and it ends up being less expensive to just toss the original code than it is to try to modify it. Nothing is inherently wrong with choosing a procedural solution; but you should make that choice knowing the risks you're taking.

Programming with Your Eyes Open

So, let's talk about my general philosophy of design.

Design is a series of informed choices, trade-offs, and risk management. If you don't understand both sides of an issue, you can't make an intelligent choice or manage risk effectively; in fact, if you don't understand all the ramifications of what you're doing, you're not designing at all. You're just stumbling in the dark. It's not an accident that every chapter in the Gang-of-Four book includes a "Consequences" section that describes when using a pattern is inappropriate and why.

Moreover, "good" and "bad" aren't absolutes. A "good" decision in one context may be "bad" in another. Every choice has a good and a bad side and is made in the context of overall criteria that are defined by necessity. Decisions aren't binary. You often have shades of goodness—consequences associated with your decisions—that can mean that none of the possibilities you're contemplating is "best." Moreover, a decision that seems good right now may not seem so good six months from now.

Saying that some language feature or common programming idiom has problems isn't the same thing as saying that you should never use that feature or idiom under any circumstances. By the same token, simply because a feature or idiom is in common use doesn't mean you should use it. Lots of programs are written by uninformed programmers, and simply being hired by Sun, Microsoft, or IBM doesn't magically improve someone's programming or design abilities. You'll find a lot of great code in the Java packages. You'll also find a lot of code that, I'm sure, the author is embarrassed to admit to writing.

To further muddy the waters, some design idioms are pushed for marketing or political reasons. Sometimes a programmer makes a bad decision, but the company wants to push what the technology can *do*, so it deemphasizes the way in which you have to do it. It's making the best of a bad situation. In this context, adopting any programming practice simply because "that's the way you're supposed to do things" is acting irresponsibly. Many failed EJB projects give proof to this principle. EJB can be a good technology when used appropriately; it can literally bring down a company when used inappropriately.

The point I'm trying to make is that you shouldn't be programming blindly. By understanding the havoc that a feature or idiom can wreak, you're in a much better position to decide whether using that feature or idiom is appropriate. Your choices should be both informed and pragmatic, made from a position of strength. That's why I'm bothering to write this book, so that you can approach your programming with your eyes open.

What Is an Object?

What does object orientation actually mean?

The patterns discussed in this book are creatures of OO systems. If a system as a whole isn't really object oriented, you don't get much benefit from using an OO pattern in some corner of the code. I've found that many programmers, even programmers who have been working with languages such as C++ or Java for years, don't have a good grasp of what exactly constitutes an OO system, however, so I have to make sure we're all clear on this point.

Balderdash!

Bjarne Stroustrup, the creator of C++, once characterized OO programming as "buzzword-oriented programming," and certainly one of the most abused (or at least misunderstood) buzzwords in the pack is *object* itself. Since the idea of an object is so central, a full discussion of what exactly an object actually is is essential to understanding OO systems and their needs.

First of all, think of an OO system as a bunch of intelligent animals inside your machine (the objects) talking to each other by sending *messages* to one another. Think "object." Classes are irrelevant—they're just a convenience provided for the compiler. The animals that comprise this system can be classified together if they have similar characteristics (if they can handle the same messages as other objects in the class, for example), but what you have at runtime is a bunch of objects, not classes. What programmers call *classes* are really classes of objects. That is, objects that have the same properties comprise a class of objects. This usage is just English, not techno-speak, and is really the correct way to think about things. We're doing object-oriented design, not class-based design.

The most important facet of OO design is *data abstraction*. This is the CIA, need-to-know school of program design. All information is hidden. A given object doesn't have any idea of what the innards of other objects look like, any more than you may know what your spouse's gallbladder looks like. (In the case of both the object and the gallbladder, you really don't want to know either.)

You may have read in a book somewhere that an object is a data structure of some sort combined with a set of functions, called *methods*, that manipulate that data structure. Balder-dash! Poppycock!

An Object Is a Bundle of Capabilities

First and foremost, an object is defined by what it can *do*, not by how it does it. In practical terms, this means an object is defined by the messages it can receive and send. The "methods" that handle these messages comprise its sole interface to the outer world. The emphasis must be on what an object can do—what capabilities it has—not on how those capabilities are implemented. The "data" is irrelevant. Most OO designers will spend considerable time in design before they even think about the data component of an object. Of course, most objects will require some data in order to implement their capabilities, but the makeup of that data is—or at least should be—irrelevant.

The *prime directive* of OO systems is as follows:

> *Never ask an object for information that you need to do something; rather, ask the object that has the information to do the work for you.*

Ken Arnold says, "Ask for help, not for information."

I'll explain the whys and wherefores in a moment, but this prime directive engenders a few rules of thumb that you can apply to see if you're really looking at an object-oriented system (I've presented them in a rather pithy way; details follow):

- Objects are defined by "contract." They don't violate their contract.
- All data is private. Period. (This rule applies to all implementation details, not just the data.)

- It must be possible to make any change to the way an object is implemented, no matter how significant that change, by modifying the single class that defines that object.
- "Get" and "set" functions are evil when used blindly (when they're just elaborate ways to make the data public). I've a lot more to say on this issue later in the "Getters and Setters Are Evil" section.

If the system doesn't follow these rules, it's not object oriented. It's that simple. That's not to say non-OO systems are bad—many perfectly good procedural systems exist in the world. Nonetheless, not exposing data is a fundamental principle of OO, and if you violate your principles, then you're nothing. The same goes for OO systems. If they violate OO principles, they're not OO by definition; they're some sort of weird hybrid that you may or may not ever get to work right. When this hybrid system goes down in flames and takes the company with it, don't blame OO. Note, however, that an OO system can be written in a procedural language (and vice versa). It's the principles that matter, not the language you're using.

Don't be fooled, by the way, by marketing hype such as "object based" and "there are lots of ways to define an object." Translate this sort of sales-speak as follows: "Our product isn't really OO—we know that, but you probably don't, and your manager (who's making the purchase decision) almost certainly doesn't, so we'll throw up a smoke screen and hope nobody notices." In the case of Microsoft, it has just redefined OO to mean something that fits with its product line. Historically, VB isn't in the least bit OO, and even now that VB has transmogrified into an OO language, most VB programs aren't object oriented because the Microsoft libraries aren't object oriented. (How many Microsoft programmers does it take to screw in a light bulb? None—let's define darkness as the new industry standard.)

Now for the "whereas" and "heretofores."

First, the notion of a *contract*: An object's contract defines the way in which the object appears to behave from the outside. The users of the objects assume that this behavior won't change over time. The interfaces that an object implements are part of the contract (so you can't lightly change method arguments or return values, for example), but other aspects of the contract can include performance guarantees, size limitations, and so forth. The object's implementation isn't part of the contract. You should be able to change it at will.

The rules in the earlier list are really just ways of enforcing the notion of a contract. Exposed implementation details would effectively make those details part of the object's contract, so the implementation couldn't change (as you discovered bugs or introduced new business requirements).

Similarly, the nuanced interpretation of the everything-is-private rule is this: If it's not private, then it's part of the contract and can't be changed. The decision to make a field public may well be correct in some (rare) situations, but the consequences of making that decision are significant.

The notion of a contract also comes into play with the third rule I mentioned earlier. Ideally, the scope of a change is limited to a single class, but interdependencies are sometimes necessary. For example, the `HashMap` class expects contained objects to implement `hashCode()`. This expectation is part of the contained object's contract.

How Do You Do It Wrong?

The main reason for following the rules in the previous section is that the code becomes easier to maintain, because all the changes that typically need to be done to fix a problem or add a feature tend to be concentrated in one place. By the way, don't confuse ease of maintenance with lack of complexity. OO systems are usually more complex than procedural systems but are easier to maintain. The idea is to organize the inevitable complexity inherent in real computer programs, not to eliminate it—a goal that an OO designer considers impossible to meet.

Consider a system that needs to get a name from some user. You may be tempted to use a `TextField` from which you extract a `String`, but that just won't work in a robust application. What if the system needs to run in China? (Unicode—Java's character set—comes nowhere near representing all the ideographs that comprise written Chinese.) What if someone wants to enter a name using a pen (or speech recognition) rather than a keyboard? What if the database you're using to store the name can't store Unicode? What if you need to change the program a year from now so that both a name and employee ID are required every place that a name is entered or displayed? In a procedural system, the solutions you may come up with as answers to these questions usually highlight the enormous maintenance problems inherent in these systems. There's just no easy way to solve even the simplest-seeming problem, and a vast effort is often required to make simple changes.

An OO solution tries to encapsulate those things that are likely to change so that a change to one part of the program won't impact the rest of the program at all. For example, one OO solution to the problems I just discussed requires a `Name` class whose objects know how to both display themselves and to initialize themselves. You'd display the name by saying, "Display yourself over there," passing in a `Graphics` object or perhaps a `Container` to which the name could drop in a `JPanel` that displayed the name. You would create a UI for a name by asking an empty `Name` object to "initialize yourself using this piece of this window." The `Name` object may choose to create a `TextField` for this purpose, but that's its business. You, as a programmer, simply don't care *how* the name goes about initializing itself, as long as it gets initialized. (The implementation may not create a UI at all—it may get the initial value by getting the required information from a database or from across a network.)

Getting back to my Visual Basic critique from a few paragraphs back, consider the way that a UI generated by VB (or VB-like systems, of which there are legions) is typically structured: You create a `Frame` class whose job is to collect messages coming in from "control" or "widget" objects in response to user actions. The `Frame` then sends messages into the object system in response to the user action. Typically, the code takes the following form:

1. "Pull" some value out of a widget using a "get" method.
2. "Push" that value into a "Business" object using a "set" method.

This architecture is known as Model/View/Controller (MVC)—the widgets comprise the "view," the `Frame` is the "controller," and the underlying system is the "model."

MVC is okay for implementing little things such as buttons, but it fails miserably as an application-level architecture because MVC requires the controller to know way too much about how the model-level objects are implemented. Too much data is flowing around in the system for the system to be maintainable.

Rather than take my word for it, let's explore a few of the maintenance problems that arise when you try to develop a significant program using the MVC architecture I just described. Taking the simple problem I mentioned earlier of needing to add an employee ID to every screen that displays an employee, in their VB-style architecture you'll have to modify every one of these screens by hand, modifying or adding widgets to accommodate the new ID field. You'll also have to add facilities to the `Employee` class to be able to set the ID, and you'll also have to examine *every* class that uses an `Employee` to make sure that the ID hasn't broken anything. (For example, comparing two `Employee` objects for equality must now use the `ID`, so you'll have to modify all this code.) If you had encapsulated the identity into a `Name` class, none of this work would be necessary. The `Name` objects would simply display themselves in the new way. Two `Name` objects would now compare themselves using the ID information, but your code that called `fred.compareTo(ginger)` or `fred.equals(ginger)` wouldn't have to change at all.

You can't even automate the update-the-code process, because all that WYSIWYG form layout touted in the advertisements hides the code-generation process. In any event, if you automatically modify machine-generated code, your modifications will be blown away the next time somebody uses the visual tool. Even if you don't use the tool again, modifying machine-generated code is always risky since most of the VB-style tools are picky about what this code looks like, and if you do something unexpected in your modifications, the tool is likely to become so confused that it'll refuse to do anything at all the next time you *do* need to use it. Moreover, this machine-generated code is often miserable stuff, created with little thought given to efficiency, compactness, readability, and other important issues.

The real abomination in MVC architecture is the "data-bound grid control," a table-like widget that effectively encapsulates the SQL needed to fill its cells from a database. What happens when the underlying data dictionary changes? All this embedded SQL breaks. You'll have to search out every screen in the system that has a data-bound control and change that screen using a visual tool. Going to a "three-tier" system, where the UI layer talks to a layer that encapsulates the SQL, which in turn talks to the database, does nothing but make the problem worse since the code you have to modify has been distributed into more places. In any event, if the middle tier is made of machine-generated code (usually the case), then it's very existence is of little use from a maintenance point of view.

All this modifying-every-screen-by-hand business is way too much work for me. Any time savings you may have made in using some tool to produce the initial code is more than lost as soon as the code hits maintenance.

The appeal of these systems often lies in familiarity. They help you program in an unfamiliar OO language using a familiar procedural mind-set. This sort of I-can-program-FORTRAN-in-any-language mindset precludes your leveraging the real maintenance benefits of OO systems, however. I personally think there's absolutely no reason to use Java unless you're indeed implementing an OO design. Java is simple only when compared against C++. You're better off just using some procedural language that really is simple if you want to write procedural systems. (I don't agree with many Java proponents who claim that the side benefits of Java such as type safety, dynamic loading, and so forth, justify writing procedural Java.)

On the other hand, if you *are* doing an OO design, a language designed to implement OO systems (such as Java) can make the implementation dramatically easier. Many C programmers try to program in Java as if they were programming in C, however, implementing procedural systems in Java rather than OO systems. This practice is really encouraged by the language, which unfortunately mimics much of C and C++'s syntax, including flaws such as the messed-up

precedence of the bitwise operators. Java mitigates the situation a bit because it's more of a "pure" OO language than C++. It's harder, though not impossible, to abuse. A determined individual can write garbage code in any language.

So How Do You Do It "Right?"

Because the OO way of looking at things is both essential and unfamiliar, let's look at a more involved example of both the wrong (and right) way to put together a system from the perspective of an OO designer. I'll use an ATM machine for this example (as do many books), not because any of us will be implementing ATMs but because an ATM is a good analog for both OO and client/server architectures. Look at the central bank computer as a server object and an ATM as a client object.

Most procedural database programmers would see the server as a repository of data and the client as a requester of the data. Such a programmer may approach the problem of an ATM transaction as follows:

1. The user walks up to a machine, inserts the card, and punches in a PIN.
2. The ATM then formulates a query of the form "give me the PIN associated with this card," sends the query to the database, and then verifies that the returned value matches the one provided by the user. The ATM sends the PIN to the server as a string—as part of the SQL query—but the returned number is stored in a 16-bit `int` to make the comparison easier.
3. The user then requests a withdrawal.
4. The ATM formulates another query; this time it's "give me the account balance." It stores the returned balance, scaled appropriately, in a 32-bit `int`.
5. If the balance is large enough, the machine dispenses the cash and then posts an "update the balance for this user" to the server.

(By the way, this isn't how real ATM machines work.)

So what's wrong with this picture? Let's start with the returned balance. What happens when Bill Gates walks into the bank wanting to open a non-interest-bearing checking account and put all his money in it? You *really* don't want to send him away, but last time you looked he was worth something like 100 gigabucks. Unfortunately, the 32-bit `int` you're using for the account balance can represent at most 20 megabucks (4 gigabucks divided by 2 for the sign bit divided by 100 for the cents). Similarly, the 16-bit `int` used for the PIN can hold at most 4 decimal digits. And what if Bill wants to use "GATES" (five digits) for his PIN? The final issue is that the ATM formulates the SQL queries. If the underlying data dictionary changes (if the name of a field changes, for example), the SQL queries won't work anymore. (Though this example is obviously nonsensical, consider the before-the-euro lira and the pain of transitioning to the euro.)

The procedural solution to all these problems is to change the ROMs in every ATM in the world (since there's no telling which one Bill will use) to use 64-bit doubles instead of 32-bit `int`s to hold account balances and to 32-bit longs to hold 5-digit PINs. That's an enormous maintenance problem, of course.

Stepping into the real world for a moment, the cost of software deployment is one of the largest line items on an IT department's budget. The client/sever equivalent of "swapping all the ROMs"—deploying new versions of the client-side applications—is a *big* deal. You can find similar maintenance problems inside most procedural programs, even those that don't use databases. Change definitions of a few central data types or global variables (the program's equivalent of the data dictionary), and virtually every subroutine in the program may have to be rewritten. It's exactly this sort of maintenance nightmare that OO solves.

To see how an OO point of view can solve these problems, let's recast the earlier ATM example in an object-oriented way, by looking at the system as a set of cooperating objects that have certain capabilities. The first step in any OO design is to formulate a "problem statement" that presents the problem we're trying to solve entirely in what's called the "problem domain." In the current situation, the problem domain is Banking. A problem statement describes a *problem*, not a computer program. I could describe the current problem as follows:

> *A customer walks into a bank, gets a withdrawal slip from the teller, and fills it out. The customer then returns to the teller, identifies himself, and hands him or her the withdrawal slip. (The teller verifies that the customer is who he says he is by consulting the bank records). The teller then obtains an authorization from a bank officer and dispenses the money to the customer.*

Armed with this simple problem statement, you can identify a few potential "key abstractions" (classes) and their associated operations, as shown in Table 1-2. I'll use Ward Cunningham's CRC-Card format (discussed in more depth shortly).

Table 1-2. *Use-Case Participants Listed in CRC-Card Format*

Class	Responsibility	Collaborates With
Bank Records	Creates withdrawal slips. Verifies that the customers are who they say they are.	Teller: Requests empty deposit slip.
Bank Officer	Authorizes withdrawals.	Teller: Requests authorization
Withdrawal Slip	Records the amount of money requested by the teller.	Bank Records: Creates it. Bank Officer: Authorizes the withdrawal. Teller: Presents it to customer.
Teller	Gets deposit slips from the Bank Records and routes the deposit slip to the Bank Officer for authorization.	Bank Records: Creates deposit slips. Bank Officer: Authorizes transactions.

The server, in this model, is really the Bank-Officer object, whose main role is to authorize transactions. The Bank, which is properly a server-side object as well, creates empty deposit slips when asked. The client side is represented by the Teller object, whose main role is to get a deposit slip from the Bank and pass it on. Interestingly, the customer (Bill) is external to the system so doesn't show up in the model. (Banks certainly have customers, but the customer isn't an attribute of the bank any more than the janitorial service is part of the bank. The customer's accounts could be attributes, certainly, but not the actual customers. You, for example, don't define yourself as a piece of your bank.) An OO ATM system just models the earlier problem statement. Here's the message flow:

1. Bill walks up to an ATM, presents his card and PIN, and requests a withdrawal.
2. The `Teller` object asks the server-side `BankRecords` object, "Is the person with this card and this PIN legitimate?"
3. The `BankRecords` object comes back with "yes" or "no."
4. The `Teller` object asks the `BankRecords` object for an empty `WithdrawalSlip`. This object will be an instance of some class that implements the `WithdrawalSlip` interface and will be passed from the `BankRecords` object to the `Teller` object *by value*, using RMI. That's important. All that the `Teller` knows about the object is the interface it implements—the implementation (the .class file) comes across the wire along with the object itself, so the `Teller` has no way of determining how the object will actually process the messages sent to it. This abstraction is a *good* thing because it lets you change the way that the `WithdrawalSlip` object works without having to change the `Teller` definition.
5. The `Teller` object tells the `WithdrawalSlip` object to display a user interface. (The object complies by rendering a UI on the ATM screen using AWT.)
6. Bill fills in the withdrawal slip.
7. The `Teller` object notices that the initialize-yourself operation is complete (perhaps by monitoring the OK key) and passes the filled-out `WithdrawalSlip` object to the server-side `BankOfficer` object (again by value, using RMI) as an argument to the message, "Am I authorized to dispense this much money?"
8. The server-side `BankOfficer` object comes back with "yes" or "no."
9. If the answer is "yes," the ATM dispenses the money. (For the sake of simplicity, I won't go into how that happens.)

Of course, this isn't the only (or even the ideal) way to do things, but the example gets the idea across—bear with me.

The main thing to notice in this second protocol is that all knowledge of how a balance or PIN is stored, how the server decides whether it's okay to dispense money, and so forth, is hidden inside the various objects. This is possible because the server is now an object that implements the "authorization" capability. Rather than requesting the data that you need to authorize a transaction, the `Teller` asks the (server-side) `BankOfficer` object (which has the data) to do the work for it. No data (account balance or PIN) is shipped to the ATM, so there's no need to change the ATM when the server code changes.

Also note that the `Teller` object isn't even aware of how the money is specified. That is, the requested withdrawal amount is encapsulated entirely within the `WithdrawalSlip` object. Consequently, a server-side change in the way that money is represented is entirely transparent to the client-side `Teller`. The bank's maintenance manager is happily sleeping it off in the back office instead of running around changing ROMs.

If only ATMs had been written this way in Europe, translation to the euro would have been a simple matter of changing the definition of the `WithdrawalSlip` (or `Money`) class on the server side. Subsequent requests for a `WithdrawalSlip` from an ATM would get a euro-enabled version in reply.

Cellular Automata

Let's expand our notions of OO to include things such as interfaces with another example that will pave the way for understanding the Game of Life program used later in the book.

A good case study of a natural OO system is a class of programs called *cellular automata*. These programs solve complex problems in a very object-oriented way: A large problem is solved by a collection of small, identical objects, each of which implements a simple set of rules, and each talks only to its immediate neighbors. The individual cells don't actually know anything about the larger problem, but they communicate with one another in such a way that the larger problem seems to solve itself.

The classic example of a cellular automaton, a solution for which is way beyond the scope of this book, is traffic modeling. The problem of predicting traffic flow is extremely difficult; it's a classic chaos-theory problem. Nonetheless, you can model traffic flow in such a way that watching the simulation in action can help you make predictions based on the model's behavior. Predicting traffic flow and simulating it are different problems, and cellular automata are great at simulating chaotic processes.

I'll spend a few pages discussing the traffic-flow problem, not only because it demonstrates automata, but also because the example illustrates several basic principles of OO design that I want you to understand before you can look at an OO system such as Game of Life.

Most programs work by implementing an algorithm—a single (though often complex) formula that has well-defined behavior when presented with a known set of inputs. Any solution that attempts to model traffic flow in an entire city using a single (complex) algorithm is just too complicated to implement. As is the case with most chaos problems, you don't even know how to write an algorithm to "solve" the traffic-flow problem.

Cellular automata deal with this problem by avoiding it. They don't use algorithms per se, but rather they model the behavior of a tractable part of the system. For example, rather than modeling traffic flow for an entire city, a cellular automaton breaks up the entire street grid into small chunks of roadway and models only this small chunk. The road chunks can talk to adjoining road chunks, but the chunks don't know anything about the entire street grid.

You can model the behavior of a small chunk of Roadway pretty easily. The chunk has a certain capacity based on number of lanes, and so on. There's a maximum speed based on the percentage of capacity and speed limits, and there's a length. That's it. Cars arrive at one end of the road and are pushed out the other end sometime later. We'll need two additional objects to round out the system: a Car, and a Map, both of which also have easy-to-model behavior. (I'll talk about these other objects in a moment.)

The various objects in this system must communicate across well-defined interfaces. (Figure 1-2 shows the entire conversation I'm about to discuss.)

The Road interface has two methods.

1. Can you take *N* cars?

```
boolean canYouAcceptCars(int n, Road fromThisRoad )
```

2. Give me *N* cars.

```
Car[] giveMeCars(int n)
```

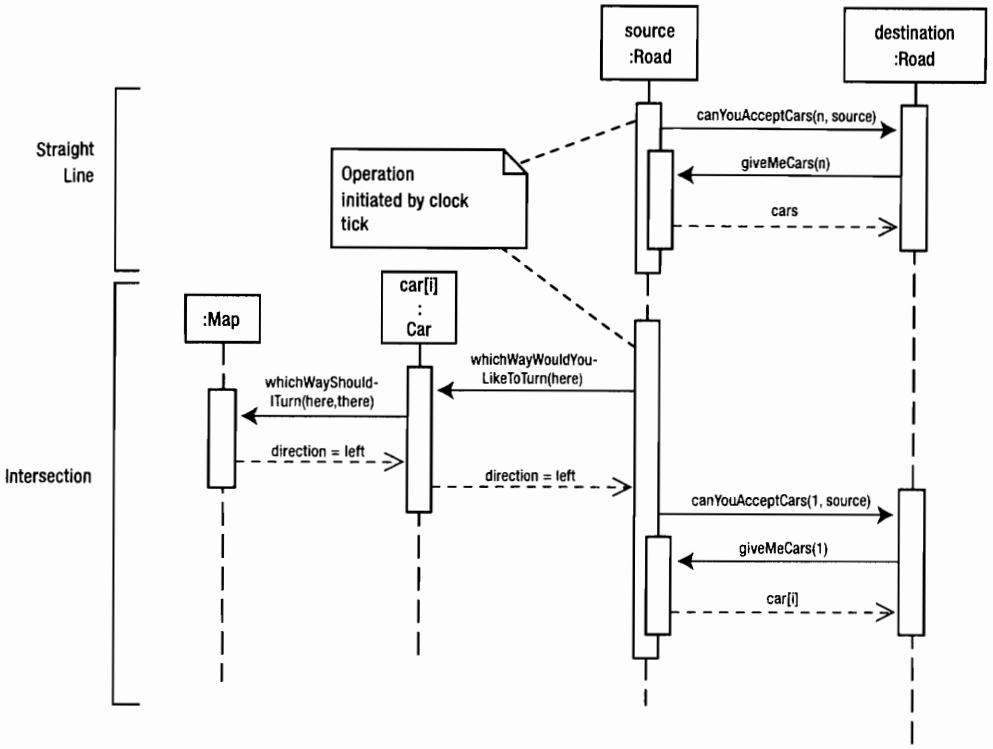

Figure 1-2. *UML sequence diagram for the traffic model*

Road segments communicate using a simple handshake. The current segment decides that it has to get rid of a couple of cars, so it asks an adjacent segment if it can take them (Message 1). The adjacent segment accepts the cars by asking for them (Message 2).

You'll see this two-part handshake again in Chapter 3. The initial request has to carry with it a Road reference that the receiving Road can use to request the cars; otherwise the receiving segment doesn't know which source segment is making the request. A segment in the middle of the block talks to two neighbors (the two adjacent Road segments), an intersection has four neighbors, and so forth. (These connections are set up when the street grid is created and would be implemented as constructor arguments.)

The Road segment has a few rules that it uses internally to decide when to evict cars. For example, the average effective speed of a Car (the difference in time between when the Car enters the Road and when it leaves) may be a function of traffic density—the number of Cars on the segment. Different road types (highway, alley, and so on) may implement these rules differently. These rules are known only by the Road, however. As is the case in any OO system, these sort of rules can be changed radically without impacting the surrounding code, because the interface to a Road segment doesn't change.

The next object you need is a Car. The Road is primarily a caretaker of Cars. Since the speed limit and Road-segment length are attributes of the Road, the Road can easily determine how long to hold onto a particular car without having to interact with the Car at all.

The only difficulty is an intersection. The Road needs to know to which neighbor to route the Car. Solve this problem with a second simple interface. (The Car implements this one, and the Road uses it.)

1. You are *here*; which way would you like to turn?

```
Direction whichWayWouldYouLikeToTurn(Location here)
```

Again, the Road couldn't care less how the Car answers the question, as long as it gets an answer. When the code is in debugging, the method that handles this message may print a query on a console and do whatever you type. The real system would need an automated solution, of course, but you can make the change from manual to automated by changing the Car class alone. None of the rest of the system is affected.

Notice that the Car doesn't know exactly where it is (just like the real world that we're modeling). The Road does know where it is (its Location), however, so the Road passes its Location into the Car. Since the Location changes, the Car doesn't bother to store it internally. The Car needs only a single attribute: a destination.

The Car needs a way to answer the which-way-do-you-want-to-turn question, so you need one more object: a Map. The Map needs another one-message interface.

2. I am *here*, and I need to go *there*; which way should I turn?

```
Direction whichWayShouldITurn(Location here, Location there)
```

Again, the Car has no idea how the map answers the question, as long as it gets an answer. (This routing problem is, by far, the hardest part of the system to write, but the problem has already been solved by every GPS navigator on the market. You may be able to buy the solution.) Note how the Car is passed its location, which it relays to the Map. This process, called *delegation*, is also commonplace in OO systems. A given object solves a problem by delegating to a contained object, passing that contained object any external information it needs. As the message propagates from delegator to delegate, it tends to pick up additional arguments.

The last piece of the puzzle is figuring out how cars get onto the Road to begin with. From the perspective of traffic modeling, a house is really a kind of dead-end Road called a *driveway*. Similarly, an office building is a kind of Road called a *parking lot*. The house and office-building objects implement the Road interface, know the Road segments to which they're connected, and inject cars (or accept them) into the system at certain times of day using the Road interface—all easy code to implement.

Now let's add a user interface. It's a classic requirement of OO systems that an object not expose implementation details. Our goal is maintainability. If all the implementation information is a closely guarded secret of the object, then you can change the implementation of that object without impacting the code that uses the object. That is, the change doesn't "ripple out" into the rest of the system. Since all changes are typically concentrated in a single class definition, OO systems are easy to maintain, but only if they follow this encapsulation rule. (You may have a good reason to violate the encapsulation occasionally, but do so knowing that your system will be harder to maintain as a consequence.)

The encapsulation requirement implies that a well-designed object will have at least some responsibility for creating its own UI. That is, a well-done class won't have getter or setter methods because these methods expose implementation details, introducing down-the-line maintenance problems as a consequence. If the implementation of the object

changes in such a way that the type or range of values returned by a getter method needs to change, for example, you'll have to modify not only the object that defines the getter but also all the code that calls the "getter." I'll talk more about this issue and about how to design systems without getter and setter methods in a moment.

In the current system, you can build a UI by adding a single method to the Road interface.

3. Draw a representation of yourself along this *line*:

```
drawYourself(Graphics g, Point begin, Point end);
```

The Road UI could indicate the average speed of the traffic (which will vary with traffic density) by changing the line color. The result would be a map of the city where traffic speed is shown in color. The Map, of course, needs to know about Roads, so the Map builds a rendition of itself, delegating drawing requests to Road objects when necessary. Since the Road objects render themselves, there's no need for a bunch of getter methods that ask for the information that some external UI builder needs to do the rendering: methods such as `getAverageSpeed()` are unnecessary.

Now that the groundwork is done, you'll set the wheels in motion, so to speak. You hook up Roads, driveways, and parking lots to each other at compile time. Put some cars in the system (also at compile time), and set things going. Every time a clock "ticks," each Road segment is notified, decides how many cars it needs to get rid of, and passes them along. Each Road segment automatically updates its piece of the UI as the average speed changes. Voilà! Traffic flow.

Once you've designed the messaging system, you're in a position to capture what you've learned in a static-model diagram. Associations exist only between classes whose objects communicate with one another, and only those messages that you need are defined. Figure 1-3 shows the UML. Note that it would have been a waste of time to start with the static model. You need to understand the message flow before you can understand the relationships between classes.

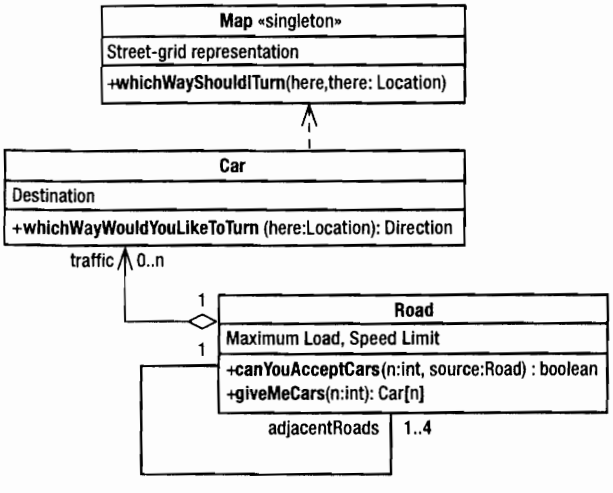

Figure 1-3. *UML static-model diagram for the traffic model*

If you want hands-on experience playing with a traffic simulator of this sort, look at Maxis Software's SimCity. Having not seen the source code, I don't actually know if SimCity *is* implemented as an automaton, but I'd be shocked if it wasn't one. It certainly acts like it on at the user-interface level, so it will do for our purposes. Maxis has a free online version of SimCity Classic on its web site (`http://www.maxis.com`).

Getters and Setters Are Evil

As I mentioned earlier, it's a fundamental precept of OO systems that an object not expose any of its implementation details. This way, you can change the implementation without needing to change the code that *uses* the object. It follows that you should avoid getter and setter functions, which typically do nothing but provide access to implementation details (fields), in OO systems. Note that neither the ATM nor traffic-flow example used getter or setter methods to do their work.

This isn't to say that your functions shouldn't return values or that "get" or "set" functionality is never appropriate. Objects must sometimes move through the system to get work done. Nonetheless, get/set functions are often used inappropriately as a means of accessing otherwise private fields, and it's that usage that will give you the most trouble. I'll discuss what I consider to be appropriate uses of get/set methods at the end of this section. Getter and setter methods (often called *accessors* and *mutators*, though the word *accessor* is commonly used for both) usually indicate a lack of clear, up-front thinking about the problem you're solving. Programmers often put them into class definitions because they don't want to think about how objects of that class will actually communicate at runtime. The presence of a getter lets you defer that thinking until you're actually coding. This behavior is plain laziness; it isn't "programming for flexibility."

Consider this trivial example of why "getters" should be avoided: There may be 1,000 calls to a `getX()` method in your program, and every one of those calls assumes that the return value is a particular type. The return value of `getX()` may be stored in a local variable, for example, and the variable type must match the return-value type. If you need to change the way that the object is implemented in such a way that the type of *X* changes, you're in deep trouble. If *X* used to be an `int`, but now has to be a `long`, you'll now get 1,000 compile errors. If you fix the problem incorrectly by casting the return value to `int`, the code will compile cleanly but won't work. (The return value may be truncated.) You have to modify the code surrounding every one of those 1,000 calls to compensate for the change. I, at least, don't want to do that much work.

Now consider the case of a `Money` class. Originally written to handle only U.S. dollars, it has a `getValue()` method that returns a double and a `setValue()` that sets a new value. The first problem is that you can do nonsensical things with money, illustrated in the following code:

```
Money a, b, c;
//...
a.setValue( b.getValue() * c.getValue() );
```

What does it mean to multiply $2 by $5?

The second problem is more significant: You need to internationalize the application to handle multiple currencies. You go into the class and add a field called `currency` that's set (internally) to values such as `US_DOLLAR`, `YEN`, `LEU`, and `HRYVNA`. Small change; big problems.

In any event, what's `getValue()` going to return? It can't just return a `double` because that value no longer tells you anything useful. You need to know the currency, too. It can't normalize the return value on U.S. dollars because the exchange rate changes by the minute. What are you going to do with the value in any case? You can't just print it, because you need the currency again. You could augment `getValue()` with `getCurrency()`, but now all the code that uses the value must also get the currency and normalize on some standard currency locally. That's a lot of work that may need to be duplicated in 1,000 places in the code. You also have to find every screen in the system where the value of money is displayed and change the display logic to include currency. This "simple" change is rapidly becoming an incredible mess.

Another example: Think of all the problems that were caused by `System.in`, `System.out`, and `System.err` when the `Reader` and `Writer` classes were introduced to Java. These three fields were `public`, which is itself anathema. Simply wrapping them (with a `System.getOut()` that returned `System.out`, for example) doesn't improve the actual problem: `System.out` and `System.err` need to be a (Unicode-based) `Writer` objects, not (byte-based) `PrintStream` objects. Ditto for `System.in` and `Reader`. Changing the declared types of the objects that hold `System.out` isn't, in and of itself, enough. Writers are used differently than Output streams. They have different semantics and different methods. You have to change (or at least examine) all the code surrounding the use of `System.out` access as a consequence. If your program had been writing Unicode using the old `System.out`, for example, it needed two `write()` calls to write a single glyph. It also needed some logic to extract the high and low bytes of the glyph to write them separately. All that code has to be removed with the `Writer` version.

The problem is compounded by force of habit. When procedural programmers come to Java, they tend to start out by building code that looks familiar. Procedural languages don't have classes, but they do have things such as the C `struct` (think: a class without methods; everything's `public`). It seems natural, then, to mimic a `struct` by building class definitions with virtually no methods and nothing but `public` fields. These procedural programmers read somewhere that fields should be `private`, however, so they make the fields `private` and supply `public` get/set methods. They haven't achieved much other than complicating the public access, though. They certainly haven't made the system object oriented.

Procedural programmers will argue that a public accessor that wraps a private field is somehow "better" than a public field because it lets you control access. An OO programmer will respond that all access—controlled or otherwise—leads to potential maintenance problems. Controlled access may be better than unfettered access, but that doesn't make the practice good. The accessor-is-better-than-direct-access argument misses the real point entirely: The vast majority of your classes don't need the accessor (or mutator) methods at all. That is, if the messaging system is designed carefully (I'll talk about how in a moment), then you can probably dispense with the get/set methods entirely and make your classes more maintainable as a consequence.

This isn't to say that return values are bad or that you can eliminate all "get" methods from your program—you can't. But minimizing the getter/setter functions will make the code more maintainable.

From a purely practical perspective, heavy use of get/set methods make the code more complicated and less agile. Consider a typical procedural "god" class, which collects the information that it needs to do some piece of work from other objects. The god-class implementation is littered with "get" calls. What if the object that already has the data does the work, though? That is, what if you moved the code that does real work from the god class to the place where the data is stored? The accessor calls disappear, and the code is simplified.

The get/set methods also make the program inflexible (it can't accommodate new business requirements easily) and hard to maintain. Perhaps the most important principle of OO systems is *data abstraction*: The way in which an object goes about implementing a message handler should be completely hidden from other objects. That's one of the reasons that all of your instance variables (the nonconstant fields of a class) should be `private`. If you make an instance variable `public`, then you can't change the field as the class evolves over time, because you'd break the external code that used the field. You really don't want to search out 1,000 uses of some class simply because you make a change to that class.

Naive getter and setter methods are dangerous for the same reason that `public` fields are dangerous: They provide external access to implementation details. What if you need to change the type of the accessed field? You also have to change the return type of the accessor. This return value is used lots of places, though, so you'll have to change all of that code as well. I want the effects of a change to be limited to a single class definition, however. I don't want them to ripple out into the entire program.

This principle of implementation hiding leads to a good acid test of the quality of an OO system: Can you make massive changes to a class definition—even throw out the whole thing and replace it with a completely different implementation—without impacting any of the code that uses objects of that class? This sort of modularization makes maintenance much easier and is central to the notion of object orientation. Without implementation hiding, there's little point in using other OO features.

Since accessors violate the principle of encapsulation, you can argue quite reasonably that a system that makes heavy or inappropriate use of accessors simply isn't object oriented. More to the point, if you go through a design process, as compared to just coding, you'll find that there will be hardly any accessors in your program. The process is important.

You'll notice that there are no getter/setter methods in the traffic-modeling example. There's no `getSpeed()` method on a Car or `getAverageSpeed()` method on a Road segment. You don't need `getLocation()` or `setLocation()` methods on the Car because you're storing location information in the Road, where it belongs. You don't need a `setAverageSpeed()` on the Road because it figures its own speed. You don't need a `getAverageSpeed()` on the Road because no other object in the system needs that information. The lack of getter/setter methods doesn't mean that some data doesn't flow through the system; the Road passes its location to the Car, for example. Nonetheless, it's best to minimize data movement as much as possible. You can go a long way toward getting it "right" by observing the following rule: **Don't ask for the information that you need to do some work; ask the object that has the information to do the work for you.**

For example, you don't say the following:

```
Money a, b, c;
//...
a.setValue( a.getValue() + b.getValue() );
```

Rather, you ask the `Money` object to do the work, as follows:

```
Money a, b, c;
//...
a.increaseBy( b );
```

You don't say, "Give me this attribute so I can print it." You say, "Give me a printable rendering of this attribute" or "print yourself."

Another way to cast this rule is to think about coarse-grained vs. fine-grained operations. A coarse-grained operation asks an object to do a lot of work. A fine-grained operation asks the object to do only a small amount of work. Generally, I prefer coarse-grained methods because they simplify the code and eliminate the need for most getter and setter methods.

Accessor and mutator methods end up in the model because, without a well-thought-out dynamic model to work with, you're only guessing how the objects of a class will be used. Consequently, you need to provide as much access as possible, because you can't predict whether you'll need it. This sort of design-by-guessing strategy is inefficient at best because you end up wasting time writing methods that aren't used (or adding capabilities to the classes that aren't needed). When you follow the static-model-first approach, the best you can hope for is a lot of unnecessary work developing these unused or too-flexible methods. At worst, the incorrect static model creates so much extra work that the project either fails outright, or if you manage to get it built, the maintenance cost is so high that a complete rewrite is less expensive. Remembering back to the traffic-flow example, I used the static model to capture relationships that I discovered while modeling the messaging system. I didn't design the static model first and then try to make the dynamic model work within the confines of that static model.

By designing carefully, focusing on what you need to do rather than how you'll do it, you'll eliminate the vast majority of getter/setter methods in your program.

Render Thyself

Probably the most shocking thing I've done in the traffic-model example is put a `drawYourself(...)` method on the Road segment. I've (gasp!) put UI code into the business logic! Consider what happens when the requirements of the UI change, though. For example, I may want to represent the Road as a bifurcated line with each direction having its own color. I may want to actually draw dots on the lines representing the cars, and so on. If the Road draws itself, then these changes are all localized to the `Road` class. Moreover, different types of Roads (parking lots, for example) can draw themselves differently. The downside, of course, is that I've added a small amount of clutter to the `Road` class, but that UI clutter is easily concentrated in an inner class to clean up things.

Also, bear in mind that I haven't actually put any UI code into the business logic. I've written the UI layer in terms of AWT or Swing, both of which are abstraction layers. The actual UI code is in the AWT/Swing implementation. That's the whole point of an abstraction layer—to isolate your "business logic" from the mechanics of a subsystem. I can easily port to another graphical environment without changing the code, so the only problem is a little bit of clutter. This clutter is easily eliminated by concentrating it into an inner class (or by using the Facade pattern, which I'll discuss soon).

Note that only the most simple classes can get away with a simplistic `drawYourself()` method. Usually, you need finer control. Objects sometimes need to draw themselves in various ways (HTML, a Swing `JLabel`, and so on), or you may need to render only a few of the object's attributes.

Moreover, an object doesn't need to physically draw itself on the screen to isolate its implementation from the rest of the program. All you need is some sort of universal (with respect to the program) representation. An object could pass an XML rendering of itself to a display subsystem, for example. A helper class along the lines of `java.text.NumberFormat` could transform this representation for specific locals. The `Money` class that I discussed earlier could return a Unicode `String` rendering that concatenates the currency symbol and value, represented in a localized fashion. You could even return a .gif image or a `JLabel`.

My main point is that if these attribute representations are handled properly, then you can still change the internal workings of a class without impacting the code that uses the representation. (A representation of some object or attribute that's presented in such a way that it can be displayed, but not manipulated, is a variant on the Memento pattern, discussed later in the current chapter.) Also, you can use several design patterns [notably, Builder] to allow an object to render itself but nonetheless isolate the UI-creation code from the actual object. I'll discuss this pattern further in Chapter 4.

JavaBeans and Struts

"But," you may object, "what about JavaBeans, Struts, and other libraries that use accessors and mutators?" What about them? You have a perfectly good way to build a JavaBean without getters and setters; the `BeanCustomizer`, `BeanInfo`, and `BeanDescriptor` classes all exist for exactly this purpose. The designers of the JavaBean specification threw the getter/setter idiom into the picture because they thought it'd be an easy way to for a junior programmer to create a bean, something you could do while you were learning how to do it "right." Unfortunately, nobody did that.

People often let the "tail wag the dog" when they talk about JavaBeans (or whatever library they use that has procedural elements). People seem to forget that these libraries started out as some programmer's personal attempt at solving a problem. Sometimes the programmers had a procedural bias; sometimes they had an OO bias. Sometimes the designers deliberately "dumbed down" an interface because they knew a lot of people just wouldn't "get it" otherwise.

The JavaBeans get/set idiom is an example of this last problem. The accessors were meant solely as a way to tag certain properties so that they could be identified by a UI-builder program or equivalent. You weren't supposed to call these methods yourself. They were there so that an automated tool (such as a UI builder) could use the introspection APIs in the `Class` class to infer the existence of certain "properties" by looking at method names. This approach hasn't worked out well in practice. It has introduced a lot of unnecessary methods to the classes, and it has made the code vastly too complicated and too procedural. Programmers who don't understand data abstraction actually call the tagging methods, and the code is less maintainable as a consequence. For this reason, a "metadata" feature will be incorporated into the 1.5 release of Java. Instead of using the following get/set idiom to mark an attribute, like so:

```
private int  property;
public  int  getProperty (         ){ return property;  }
public  void setProperty (int value){ property = value; }
```

you'll be able to say something like this:

```
private @property int property;
```

The UI-construction tool or equivalent will be able to use the introspection APIs to find the properties, rather than having to examine method names and infer the existence of a property from a name. More to the point, no runtime accessor is damaging your code.

Returning to Struts, this library isn't a model of OO architecture and was never intended to be. The MVC architecture embodied in Struts pretty much forces you to use get/set methods. You can reasonably argue, that given the generic nature of Struts, it *can't* be fully OO, but other

UI architectures manage to hide encapsulation better than MVC. (Perhaps the real solution is to avoid an MVC-based UI framework altogether. MVC was developed almost 30 years ago, and we've learned a lot since then.) There's one compelling reason for using Struts: The library contains a lot of code that you don't have to write, and it's "good enough" for many purposes. If "good enough" is good enough, go for it.

To sum up, people have told me that fundamental concepts of object orientation, such as implementation hiding, are "hogwash," simply because the libraries that these people use (JavaBeans, Struts, .NET, and so on) don't embody them. That argument is, I think, hogwash.

Refactoring

The other argument I've heard to justify the use of accessors and mutators is that an integrated development environment such as Eclipse or its cousins make it so easy to refactor a method definition to return a different argument type that there's no point in worrying about this stuff. I still worry, though.

Firstly, Eclipse just refactors within the scope of the existing project. If your class is being reused in many projects, then you have to refactor all of them. A company that properly reuses class will have many groups of programmers all working on separate projects in parallel, and these other programmers won't take kindly to your telling them that they have to refactor all their code because of some specious change you want to make to a shared class.

Secondly, automated refactoring works great for simple things, but not for major changes. The ramifications of the change are typically too far-reaching for an automated tool to handle. You may have to change SQL scripts, for example, and the effects of the change may ripple indirectly into the methods that are called from the place where the refactoring is made.

Finally, think about the changes to `Money` and `System.out` discussed earlier. Simply changing a few return-value types isn't sufficient to handle the changes I discussed. You have to change the code that surrounds the getter invocation as well. Though it's hard to argue that refactoring the code isn't a good thing, you can't do this sort of refactoring with an automated tool.

People who use the automated-refactoring argument also tend not to understand the most important issue: Overuse of accessors and mutators at the key-abstraction level is an indication of a poorly designed messaging system. In other words, the code is probably structured so poorly that maintenance is unnecessarily difficult, whether or not you can refactor easily. A redesign is required, not a refactor.

Using the earlier `System.out` example as a characteristic, imagine that you redesigned Java to print a `String` on the console as follows:

```
String s = "hello world";

s.print( String.TO_CONSOLE );
```

and loaded a `String` like this:

```
s.load( String.FROM_CONSOLE );
```

All the byte-vs.-Unicode problems would disappear into the `String` class implementation. Any changes from byte-based to glyph-based I/O would disappear. Since the whole point of the `Reader` and `Writer` interfaces is to load and store strings, you could dispense with them entirely. Overloads of `print(...)` and `load(...)` could handle file I/O.

You can argue with me about whether things *should* be done in this way. You can also quibble about whether `TO_CONSOLE` should be a member of the `String` or `File` class. Nonetheless, the redesign eliminated the need for `System.out` and its accessors. Of course, you can think of a billion things to do with a string and can reasonably argue that *all* of those things shouldn't be part of the `String` class, but design patterns (Visitor, Strategy, and so on) can address this problem.

Life Without Get/Set

So, how do you end up with a design without getters and setters in it? That is, how do you design a messaging system that minimizes the need for accessors and mutators? The solution is in design, not in coding. There's no simplistic just-replace-this-code-with-that-code solution because the problem has to do with the way you think about the interaction of objects. You can't just refactor the get/set methods out of the code—you have to rebuild the code from scratch with a fundamentally different structure.

The OO-design process is centered on *use cases*: stand-alone tasks performed by an end user that have some useful outcome. "Logging On" isn't a use case because there's no outcome that's useful in the problem domain. "Drawing a Paycheck" is a use case. In the earlier ATM example, I was flushing out the "Depositing Funds" use case.

An OO system, then, implements the activities needed to play out the various "scenarios" that comprise a use case. The runtime objects that have roles in the use case act out their roles by sending messages to one another. Not all messages are equal, however. You haven't accomplished much if you've just built a procedural program that uses objects and classes.

Back in 1989, Kent Beck and Ward Cunningham were teaching classes on OO design, and they were having problems getting programmers to abandon the get/set mentality. They characterized the problem as follows:

> *The most difficult problem in teaching object-oriented programming is getting the learner to give up the global knowledge of control that is possible with procedural programs, and rely on the local knowledge of objects to accomplish their tasks. Novice designs are littered with regressions to global thinking: gratuitous global variables, unnecessary pointers, and inappropriate reliance on the implementation of other objects.*

When they talk about "global knowledge of control," they're describing the "god" class I discussed earlier—a class whose objects collect information from elsewhere and then process that information (rather than allowing the object that has the data to do the processing). That "inappropriate reliance on the implementation of other objects" is an accessor or mutator call.

Cunningham came up with a teaching methodology that nicely demonstrates the design process: the CRC card. The basic idea is to make a set of 4×6 index cards that are laid out in the following three sections:

Class: The name of a class of objects.

Responsibilities: What those objects can do. These responsibilities should be focused on a single area of expertise.

Collaborators: Other classes of objects to which the current class of objects can talk. This set should be as small as possible.

The initial pass at the CRC card is just guesswork—things will change.

In class, Beck and Cunningham picked a use case and made a best guess at determining which objects would be required to "act out" the use case. They typically started with two objects and added others as required as the scenario played out. People from the class were selected to be those objects and were handed a copy of the associated CRC card. If several objects of a given class were needed, several people represented those objects. The students literally acted out the use case. Here are the rules I use when acting out a use case with CRC cards:

- Perform the activities that comprise the use case by talking to one another.
- You can talk only to your collaborators. If you need to talk to someone else, talk to a collaborator who can talk to the other person. If that turns out not to be possible, add a collaborator to your CRC card.
- You may not ask for the information you need to do something. Rather, you must ask the collaborator who has the information to do the work. It's okay to give your collaborators some bit of information that they need to do the work, but keep this sort of passing to a minimum.
- If something needs to be done and nobody can do it, create a new class (and CRC card) or add a responsibility to an existing class (and CRC card).
- If a CRC card gets too full, you must create another class (CRC card) to handle some of the responsibilities. Complexity is limited by what you can fit on a 4×6 index card.
- Stick to the "domain" of the problem (accounting, purchasing, and so on) in both your vocabulary and your processes. That is, model what would happen if real people who were domain experts were solving the problem. Pretend computers don't exist. It's not very often that two people say "getX" to each other in the course of doing some task, so in practice, the get/set methods won't even come up.

Once you've worked out a conversation that solves the problem, turn on a tape recorder or transcribe it. That transcription is the program's "dynamic model." The finished set of CRC cards is the program's "static model." With lots of fits and starts, it's possible to solve just about any problem in this way.

The process I just described *is* the OO-design process, albeit simplified for a classroom environment. Some people design real programs this way, using CRC cards, but the technique tends not to scale to nontrivial programs. More often than not, designers develop the dynamic and static models in UML, using some formal process (for example, RUP, Crystal, and even some flavors of Extreme Programming). The point is that an OO system is a conversation between objects. If you think about it for a moment, get/set methods just don't come up when you're having a conversation. By the same token, get/set methods won't appear in your code if you design in this way before you start coding.

The modeling must stay in the "problem domain" as long as possible, as I mentioned in the last rule. What gets most people in trouble is that they think they're doing domain

modeling but are actually modeling at the implementation level. If your messaging system isn't using the vocabulary of the problem domain—if it doesn't make sense to an average end user of your program—then you're doing implementation-level modeling. Things such as computers (or worse, the databases or UI-construction kits) have no place at this level of modeling.

In CRC modeling, for example, you need to keep the conversation in the problem domain by using the vocabulary and processes that real users would use. This way the messaging system reflects the domain directly. The database is just an internal thing that some of the classes use as a persistence mechanism and won't appear in the initial model at all.

If you keep the message structure in the problem domain, then you'll eliminate the vast majority of get/set methods, simply because "get" and "set" isn't something your domain experts do when solving most problems.

When Are Accessors and Mutators Okay?

If you must pass information between objects, encapsulate that information into other objects. A "get" function that returns an object of class `Money` is vastly preferable to one that returns a `double`.

It's best if a method returns an object in terms of an interface that the object implements because the interface isolates you from changes to the implementing class. This sort of method (that returns an interface reference) isn't really a getter in the sense of a method that just provides access to a field. If you change the internal implementation of the provider, you must change the definition of the returned object to accommodate the changes, of course. You can even return an object of different class than you used to return as long as the new object implements the expected interface. The external code that uses the object through its interface is protected.

In general, though, I try to restrict even this relatively harmless form of accessor to return only instances of classes that are *key abstractions* in the system. (If the class or interface name appears regularly in the English, domain-level description of the problem, then it's a key abstraction.)

Generally, messages should carry as little data as possible with them as arguments, but it's better to "push" data into an object than to "pull" it out. Put another way, it's better to delegate to another object, passing it some bit of information that it doesn't have, than it is for that object to call one of your methods to get the information. This isn't to say that return values are bad, but insofar as it's possible, you should return either objects that encapsulate their implementation or booleans, which give away nothing about implementation. In an ATM machine, it's better to ask "am I authorized to give Bill $20?" (a Boolean result) than it is to say "give me Bill's account balance" and make the decision locally.

One big exception exists to the no-getter-or-setter rule. I think of all OO systems as having a "procedural boundary layer." The vast majority of OO programs run on procedural operating systems and talk to procedural databases, for example. The interfaces to these external procedural subsystems are by their nature generic. The designer of JDBC hasn't a clue about what you'll be doing with the database, so the class design has to be unfocused and highly flexible. UI-builder classes such as Java's Swing library are another example of a "boundary-layer" library. The designers of Swing can't have any idea about how their classes will be used; they're too generic. Normally, it would be a bad thing to build lots of flexibility that you didn't use because it increases development time. Nonetheless, the extra flexibility is unavoidable in these boundary APIs, so the boundary-layer classes are loaded with accessor and mutator methods. The designers really have no choice.

In fact, this not-knowing-how-it-will-be-used problem infuses all of the Java packages. It's difficult to eliminate all the accessors and mutators if you can't know how objects of the class are actually used. Given this constraint, the designers of Java did a pretty good job of hiding as much implementation as they could. This isn't to say that the design decisions that went into JDBC and its ilk apply to your code. They don't. You *do* know how the classes are going to be used, so you don't have to waste time building unnecessary flexibility.

I should also mention constant values, which are often accessed directly as public members. Here's my advice:

- Don't do it if you don't have to do it. It's better to have a list scale to fit its contents than have a `MAX_SIZE`, for example.
- Use the new (JDK 1.5) `enum` facility whenever possible rather than expressly declared and initialized `static final int` values. Alternatively, use the typesafe-enum pattern described by Joshua Bloch in his book *Effective Java Programming Language Guide* (Addison-Wesley, 2001).

  The basic notion is to define an enum like this:

```
private static class Format{ private Format(); }
public static final Format SINGLE_LINE  = null;
public static final Format POPUP_DIALOG = new Format();
public static final Format PANEL        = new Format();

public displayYourselfAs( Format how )
{   // display the current value of calendar in
    // the format specified.
}
```

  Since the argument to `displayYourselfAs(...)` is a `Format` object, and since only two instances of (and three references to) format can possibly exist, you can't pass a bad value to `displayYourselfAs(...)`. Had you used the following more common `int`-enum idiom:

```
public static final int SINGLE_LINE  = 0;
public static final int POPUP_DIALOG = 1;
public static final int PANEL        = 2;

public displayYourselfAs( int how )
{   //...
}
```

  you could pass an arbitrary nonsense value (say, -1) to the method. Bloch devotes ten pages to this idiom, and I refer you to his book for more information.

- If you do have to expose a constant, make sure that it's really a constant. Java's `final` keyword guarantees that a reference can't be changed to reference something else, but it doesn't protect the referenced object. If an object is used as a constant, you have to write the class in such a way that the object can't be modified. (Java calls this kind of class *immutable*, but other than declaring all the fields of the class as `final`, there's no language mechanism to guarantee immutability. You just program the class that way.)

Consider Java's `Color` class. Once the object is created, you can't change the color, simply because the color class doesn't expose any methods that change the color. Consider this code:

```
public static final Color background = Color.RED;
//...
c.darken();
```

The call to `darken()` doesn't modify the object referenced by `background`; rather, it returns a new `Color` object that's a shade darker than the original. The foregoing code doesn't do anything, since the returned `Color` object isn't stored anywhere, and you can't say this:

```
background = c.darken();
```

because background is final.

Finally, it's sometimes the case that an object is a "caretaker" for other objects. For example, a Java `Collection` holds a bunch of objects that were passed into it from outside. Though the words "get" and "set" are often used in the names of the methods that give an object to a caretaker and fetch the object back from the caretaker, these methods don't expose any information about how the caretaker works, so they're also okay. In general, if you pass something into an object, it's reasonable to expect to be able to get that something back out again.

Databases are extreme examples of caretakers of data, though their interfaces are pushed even further in the direction of get/set methods because a database is a fundamentally procedural thing—a big bag of data; a database is part of the "boundary layer" I discussed earlier. Consequently, it's impossible to access a procedural database in an OO way. The get/set methods are unavoidable. Nonetheless, you can (and should) encapsulate the procedural calls to the database layer into higher-level domain objects and then write your code in terms of the interfaces to these encapsulating objects. Inside the encapsulating objects, you'll be doing what amounts to get/set calls on the database. Most of the program won't see this work, however, because they'll be interacting with the higher-level encapsulating object, not the database.

Summing Up the Getter/Setter Issues

So let's sum up: I'm *not* saying that return values are bad, that information can't flow through the system, or that you can eliminate all accessors and mutators from your program. Information has to flow, or the program won't do anything. That information should be properly encapsulated into objects that hide their implementation, however.

The basic issues are as follows:

- The maintainability of a program is inversely proportional to the amount of data that flows between objects.
- Exposing implementation harms maintainability. Make sure that the accessor or mutator really is required before you add it.
- Classes that directly model the system at the domain level, sometimes called *business objects*, hardly ever need accessors or mutators. You can think of the program as partitioned broadly into generic libraries that have to relax the no-getter/no-setter rule and domain-specific classes that should fully encapsulate their implementation. Getters and setters at this level are an indication that you didn't do enough up-front design work. In particular, you probably didn't do enough dynamic modeling.

- By keeping the design process in the problem ("business") domain as long as possible, you tend to design messaging systems that don't use getters and setters because statements such as "Get this" or "Set that" don't come up in the problem domain.
- The closer you get to the procedural boundary of an OO system (the database interface, the UI-construction classes, and so on), the harder it is to hide implementation. The judicious use of accessors and mutators has a place in this boundary layer.
- Completely generic libraries and classes also can't hide implementation completely so will always have accessors and mutators.
- Sometimes it's not worth the trouble to fully encapsulate the implementation. Think of trivial classes such as `Point` and `Dimension`. Similarly, private implementation classes (a `Node` class defined as a `private` inner class of `Tree`, for example) can often used a relaxed encapsulation model. On the other hand, think of all the problems that were caused by `System.in`, `System.out`, and `System.err` when the `Reader` and `Writer` classes were introduced, and what if I want to add units (feet, inches) to a Dimension?

At a JavaOne conference (I think in 1991) James Gosling was asked to give some pithy piece of programming advice to the multitude. He chose to answer (I'm paraphrasing) that maintainability was inversely proportional to the amount of data that moves between objects. The implication is that you can't get rid of all data movement, particularly in an environment where several objects have to collaborate to accomplish some task, but you should try to minimize date flow as much as possible.

When I have to pass information around, I use the following two rules of thumb:

- Pass around objects (ideally in terms of the interfaces they implement) rather than raw data.
- Use a "push" model, not a "pull" model. For example, an object may delegate to a collaborator, passing the collaborator some piece of information that the collaborator needs to do its work. The alternative—the collaborator "pulling" the information from the delegator using a getter method—is less desirable. The Flyweight pattern relies on this "push" model.

Converting a pull to a push is often just a matter of routing the message differently.

Maintainability is a continuum, not a binary. Personally, I like to err in the direction of easy maintenance, because maintenance really begins two seconds after you write the code. Code that's build with maintenance in mind tends to come together faster and have fewer bugs.

Nonetheless, you must decide where on that ease-of-maintenance continuum you want to place your program. The Java libraries are, for the most part, examples of how you need to compromise maintainability to get generic functionality. The authors of the Java packages hid as much implementation as they could, given the fact that the libraries were both completely generic and also on the procedural boundary. The price they paid is that it's difficult to make structural changes to libraries such as Swing because too many existing programs depend on implementation specifics.

Not all of the Java libraries expose implementation. Think of the Crypto APIs (in *javax.crypto*) and the `URL`/`URLConnection` classes, which expose hardly any information and are extraordinarily flexible as a consequence. The Servlet classes are a good example of an encapsulated implementation that nonetheless supports information movement, though they could go even further by providing an abstraction layer that you could use to build HTML.

So, when you see methods starting with the "get" or "set," that's a clue. Ask yourself whether the data movement is really necessary. Can you change the messaging system to use coarser-grained messages that will make the data movement unnecessary? Can you pass the information as a message argument instead of using a separate message? Would an alternative architecture work better at hiding implementation? If you have no alternative, though, go ahead and use it.