CHAPTER 1

OO Principles and Patterns

Developing more resilient systems should be our first course of action. Reuse will follow.

When designing object-oriented systems, the challenges are numerous, and the solutions are various. How do we identify an approach that will help ensure we are creating an extensible, robust, and easily maintainable system? One way is by using design patterns. Design patterns are proven design solutions that can be tailored to fit the context of a particular design challenge. In essence, they are reusable design templates. While the notion of patterns has hit mainstream development since the seminal work published in 1995 by the Gang of Four [GOF95], the number of patterns available has become almost unmanageable. So many patterns are available today that attempting to find a pattern that can solve difficult design challenges conceivably could take longer than discovering a new solution, which if designed efficiently, is probably documented as a pattern somewhere anyway. When we can't find a pattern that solves our challenges, we can take an approach during design that will ensure we are solving our challenges correctly, given the absence of a readily available pattern. Such approaches are based on some fundamental principles of object orientation.

While these fundamental principles can provide helpful guidance when developing object-oriented software, our understanding of object orientation must come first. It is virtually impossible to apply a principle when we don't fully understand the value of that principle. Therefore, we must understand not only the principles, but also the true benefits of object orientation, as well as the goals that these benefits enable us to effectively and gracefully achieve.

1.0 Principles, Patterns, and the OO Paradigm

By this time, we've all been saturated with the benefits of objects. Reuse is the Holy Grail of object orientation. Unfortunately, a lot of the works discussing object orientation exist at such a theoretical level that they can be difficult to interpret and apply pragmatically, or these works exist at such a detailed level that it can be difficult to derive a concise vision of the paradigm in its entirety. Understanding concepts such as abstraction, inheritance, encapsulation, and polymorphism is wonderful, but they are just concepts and don't provide much guidance in creating more reusable and highly maintainable systems. In fact, our discussion in this book assumes a basic understanding of these terms.

We can achieve reuse, create more flexible designs, and understand the object-oriented paradigm more thoroughly by studying and applying patterns. But even patterns don't serve as a guiding set of principles that are universally applicable, and with the proliferation of patterns over the past couple of years, simply finding the most appropriate pattern can be a daunting task. This begs some interesting questions. What are the fundamental principles of the object-oriented paradigm? Is there a set of guiding principles that we can consistently and faithfully apply to help us create more robust systems? In fact there is, and we discuss the most useful principles in Section 1.1, later in this chapter.

Before we explore these principles, however, it's important to revisit the true benefit of object orientation. We've been told that reuse is the nirvana of programming, and object orientation provides it. The reason reuse has been so heavily touted is because it impacts the bottom line. When we use easily pluggable objects, which are highly reusable, we reduce the time required to develop applications. When we develop faster, we develop more cheaply as well. Certainly, one of the benefits of object orientation can be reuse; however, it may not be the most important benefit. In the December 2000 issue of *The Rational Edge*, Walker Royce cited two interesting statistics:

- For every \$1 you spend on development, you will spend \$2 on maintenance.
- Only about 15% of software development effort is devoted to programming. [WR00]

These statistics are astounding. The cost of maintaining a system is twice that of developing it. This being the case, we need a paradigm that facilitates system maintenance as much as, if not more than, reuse. Granted, effectively reusing objects can help in reducing system maintenance, but it doesn't necessarily guarantee it. In fact, consider the following:

> *Given a class R that is being reused by both classes A and B, if A requires new or modified behaviors of R, it would make sense that any changes to R would be reflected in B as well. While this is true, what happens if B does not desire this new behavior? What if this new behavior defined for R actually broke B? In this case, we have reuse, but we don't have a high degree of maintenance.*

You might already be thinking of ways in which this scenario can be resolved. You might be saying that you wouldn't have done it this way in the first place, and there are certainly many ways to resolve the preceding problem. The granularity of the method contributes greatly to the likelihood of its reusability. The fact remains that each design is centered around flexibility, which brings us to Royce's second statistic cited earlier. If we are spending roughly 15 percent of our time programming, what are we spending the remaining 85 percent of our time doing? The answer is design, or at least variations of what many of us associate with a traditional design phase in the software development lifecycle. Of course, we also spend time managing requirements, planning projects, and testing. Focusing strictly on the design activity, had we designed the previously described example in a more effective manner, it is likely that our maintenance cost would have been reduced. But it's this design aspect that is so difficult. Therefore, following a set of guiding principles serves us well in creating more flexible designs.

Inheritance and Reuse

Those readers new to object orientation typically assume a close relation exists between inheritance and reuse. We want to debunk this myth immediately. Though reuse is touted as a benefit of object orientation, it is in fact a goal. Reuse cannot be taken for granted, nor is it guaranteed. In reality, achieving reuse requires a lot of effort and discipline, and we'll spend a lot of time in this book talking about this aspect of object orientation.

In addition, because inheritance is new to most developers exposed to objects for the first time, a false correlation typically is made between inheritance and reuse. While reuse can be achieved through inheritance,

(continues)

(continued)

it's not the primary benefit that inheritance provides. Inheritance can be used to achieve multiple goals and can be categorized two different ways. First, *interface inheritance* is the use of inheritance to achieve polymorphic behavior. Many of the principles that we discuss later in this chapter (see Section 1.1) take advantage of interface inheritance. Second, *implementation inheritance* is utilizing inheritance for reuse. While implementation inheritance can be beneficial, it should not be heavily relied upon as the mechanism of reuse. The ramifications of doing so can be detrimental.

Java is one of the first languages to make explicit the difference between interface and implementation inheritance. In Java, the `extends` keyword exemplifies implementation inheritance (with a small amount of interface inheritance through abstract methods), whereas the `implements` keyword illustrates interface inheritance. Therefore, stating that Java doesn't support multiple inheritance is not entirely true because Java does support multiple inheritance of interfaces.

Ultimately, the design chosen for our software system will impact the maintainability of our system. We call a design that impacts the maintainability of our system the software's *architecture*, and designing a system with a resilient architecture is of utmost importance. Because we know that requirements change, the resiliency of our architecture will impact our system's survival. However, the ability of our system to change, or grow to meet new requirements, and still survive are conflicting goals, known as the *architecture paradox* [SUB99].

What Is Design?

We associate design with some activity or phase within a traditional software development lifecycle. In this book, however, when we refer to design, we refer to the set of best practices and principles of object orientation that are continuously applied throughout all phases of the software development lifecycle. We even imply that lifecycle phases such as requirements, construction, and testing contain small slices of time where an emphasis is placed upon the practices and principles of design.

Suppose we have a system that fulfills its full set of requirements. As the requirements begin to change, the software begins to die, and its survival is challenged. In order to restore its survivability, we need to change the software. With each change, the software's architecture is compromised. As more changes are made, the software becomes harder to maintain. Because changes become so difficult to make, the costs associated with maintaining the system eventually reach a point where future maintenance efforts cannot be justified or introduce new errors. At this point, our system has lost its ability to grow, and it dies. Therefore, as depicted in Figure 1.1, as changes increase, survivability decreases.

This experience is a frustrating one, and it's common to blame others for these changing requirements. However, businesses typically drive these changes, and we shouldn't try to place the blame elsewhere. In fact, the problem is not with the changing requirements. We already know from experience that requirements change. A commonly quoted adage cites three certainties in life: death, taxes, and changing requirements. The fact that requirements change and compromise our systems' internal structures is not the fault of changing requirements, but the fault of our design. Requirements always change, and it is our job to deal with it!

Fortunately, one of the benefits of the object-oriented paradigm is that it enables us to easily add new data structures to our system without having to modify the existing system's code base. We achieve this through the power of inheritance and polymorphism, illustrated in Section 1.1.1, later in this chapter.

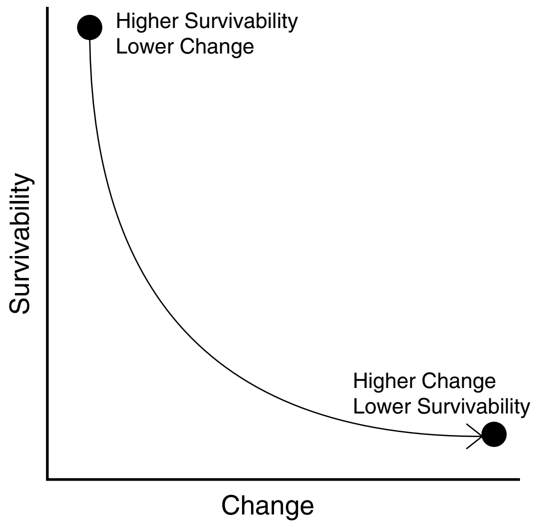

Figure 1.1 Architecture Paradox

These data structures in the object-oriented paradigm are classes. A class encapsulates behavior and data, and because we can add new classes to our system without modifying the existing code base, we can add new data and behaviors as well. Once we understand how we can realize this power when developing our applications, the only remaining trick is to apply this flexible concept to the areas within the system that are most likely to change. In this chapter, we learn how to apply this power. Throughout the remainder of this book, we examine how to identify these areas of an application requiring this flexibility.

So how do we go about designing a system that exhibits the power to make enhancements without having to actually modify the existing code base? The answer is to apply fundamental principles and patterns in a consistent, disciplined fashion. In fact, many experienced developers have an existing repertoire of proven techniques that they pull out of their bag of tricks to guide them during development. Until recently, there was not an effective way for developers to share these proven techniques with others.

Today, the software development industry abounds with patterns, of which many categories exist. Most of us have probably heard of patterns, and we will not devote our discussion here to duplicating work that has already been successfully documented. Instead, we provide an executive summary on patterns, including a few examples later in this chapter (see Section 1.3).

Patterns come in many forms. Architectural patterns focus on driving the high-level mechanisms that characterize an entire application. Analysis patterns help in solving domain-dependent obstacles. Design patterns help us solve a broad range of technical design challenges. We'll find that using patterns in conjunction with other patterns typically contributes to the achievement of the most flexible, robust, and resilient designs. Again, we'll see this firsthand as we progress throughout the book.

First, let's explore a more formal definition of a pattern:

> *A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context. [GOF95]*

Examining this definition further illustrates the potential of patterns. All patterns have a name, which enables a developer to easily refer to, and communicate with, other developers the intent of a particular pattern. Patterns help solve design challenges that continually surface. Each situation, however, is invariably different in some regards. A well-documented pattern describes the consequences of using it, as well as providing hints and examples of how to effectively utilize it. Consequently, when we use a pattern, it is unlikely that we'll implement it in the exact same manner each time.

Patterns can be thought of as algorithms for design. Certain algorithms have slight differences based on the implementation language, just as patterns vary based on the context in which they're applied. Most developers who have written sorting routines can understand the basic algorithm associated with the term *bubble sort*. Similarly, those familiar with patterns understand the structure and intent of a Strategy pattern. This naming convention is a benefit of using patterns because they enable us to communicate complex designs more effectively. Many more benefits are associated with the use of patterns, such as taking advantage of proven designs, creating more consistent designs, and providing a more concrete place to start when designing.

Patterns typically are discovered by some of the most talented object-oriented developers in the world. These patterns usually go through an intensive review and editing cycle, and thus they are proven design solutions. The review and editing cycle enables less-experienced developers to gain insights that will make their own designs as flexible as those of an experienced developer. In fact, the review and editing cycle may be the single greatest benefit associated with using patterns, because they are essentially the collective work of the most experienced designers in the object-oriented community.

Although the value of patterns is real, realizing this value also implies knowing which pattern is appropriate to use in a specific context, and how it can be applied. Because of the proliferation of patterns, it can be difficult to efficiently find a pattern that best fits a need. Principles, in comparison to patterns, exist at a higher level. The majority of patterns adhere to an underlying set of principles. In this regard, we can think of patterns as being instances of our principles. Principles are at the heart of object-oriented design. The more patterns we understand, the more design alternatives we can consider when architecting our systems. It's highly unlikely, however, that we'll ever completely understand, or even have knowledge of, all of the patterns that have been documented. By adhering to a more fundamental set of principles, it's likely that we'll encounter patterns that are new to us—patterns that may have been documented but that we aren't aware of. Or we may even discover new patterns. The point is that while patterns provide a proven starting point when designing, principles lie at the heart of what we need to accomplish when designing resilient, robust, and maintainable systems. Understanding these principles not only enhances our understanding of the object-oriented paradigm, but also helps us understand more about patterns, when to apply them, and the foundation upon which patterns are built.

1.1 Class Principles

As mentioned previously, principles lie at the heart of the object-oriented paradigm. The principles discussed in subsequent sections can help guide us during design when it might be difficult to find the most applicable pattern. We typically first look to patterns in solving our challenges. However, if we are unable to find an appropriate pattern, or are unsure if we should use a particular pattern, we should always take into consideration the principles discussed in the following sections. In fact, patterns typically are tailored slightly to fit a particular need, and these principles should be carefully considered when customizing a pattern. Many of the principles presented here first appeared in Robert Martin's *Design Principles and Design Patterns* [MARTIN00], which serves as an excellent complement to this discussion.

When applying these principles to Java, they can be broken into two categories. The first category focuses on relationships that exist between classes. These principles typically form the foundation of many design patterns. The second category of principles focuses on relationships between packages. These principles form the foundation of many architectural patterns. Keep in mind that at this point, we are primarily concerned with understanding the core concepts present within these principles. Application of these principles typically is dependent on a set of guiding heuristics, which we will continually elaborate on, and refine, as we progress throughout the book.

1.1.1 Open Closed Principle (OCP)

Classes should be open for extension but closed for modification.

The Open Closed Principle (OCP) is undoubtedly the most important of all the class category principles. In fact, each of the remaining class principles are derived from OCP. It originated from the work of Bertrand Meyer, who is recognized as an authority on the object-oriented paradigm [OOSC97]. OCP states that we should be able to add new features to our system without having to modify our set of preexisting classes. As stated previously, one of the benefits of the object-oriented paradigm is to enable us to add new data structures to our system without having to modify the existing system's code base.

Let's look at an example to see how this can be done. Consider a financial institution where we have to accommodate different types of accounts to which individuals can make deposits. Figure 1.2 shows a class diagram with accompanying descriptions of some of the elements and how we might structure a portion of our system. We discuss in detail the elements that make up various

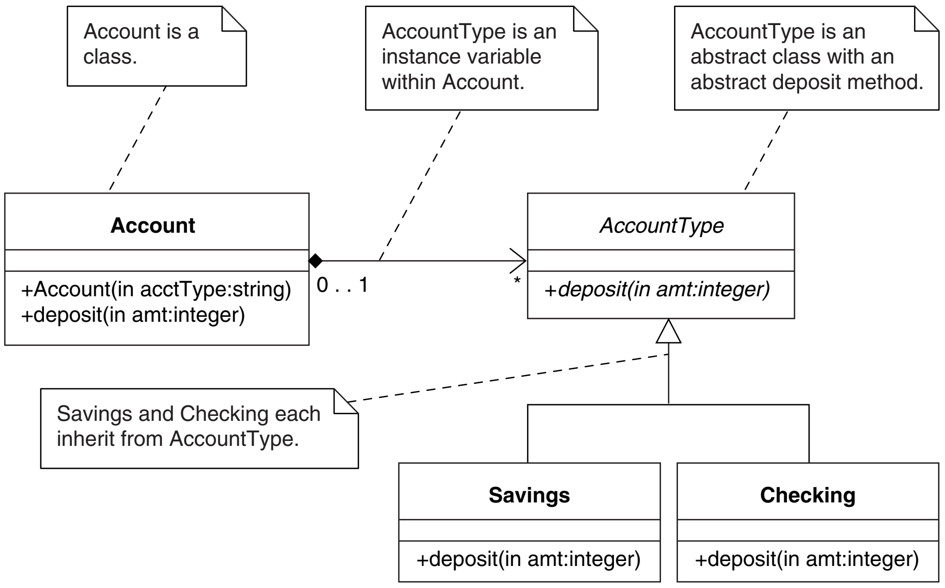
Account is a class.
AccountType is an instance variable within Account.
AccountType is an abstract class with an abstract deposit method.
Account
+Account(in acctType:string)
+deposit(in amt:integer)
0 . . 1
*
AccountType
+deposit(in amt:integer)
Savings and Checking each inherit from AccountType.
Savings
+deposit(in amt:integer)
Checking
+deposit(in amt:integer)

Figure 1.2 Open Closed Principle (OCP)

diagrams and the Unified Modeling Language (UML) in general in Chapter 3. For the purposes of our discussion in this chapter, we focus on how the OCP can be used to extend the system.

Our `Account` class has a relationship to our `AccountType` abstract class. In other words, our `Account` class is coupled at the abstract level to the `Account-Type` inheritance hierarchy. Because both our `Savings` and `Checking` classes inherit from the `AccountType` class, we know that through dynamic binding, we can substitute instances of either of these classes wherever the `AccountType` class is referenced. Thus, `Savings` and `Checking` can be freely substituted for `AccountType` within the `Account` class. This is the intent of an abstract class and enables us to effectively adhere to OCP by creating a contract between the `Account` class and the `AccountType` descendents. Because our `Account` isn't directly coupled to either of the concrete `Savings` or `Checking` classes, we can extend the `AccountType` class, creating a new class such as `MoneyMarket`, without having to modify our `Account` class. We have achieved OCP and now can extend our system without modify its existing code base.

Therefore, one of the tenets of OCP is to reduce the coupling between classes to the abstract level. Instead of creating relationships between two concrete classes, we create relationships between a concrete class and an abstract class, or in Java, between a concrete class and an interface. When we create an extension of our base class, assuming we adhere to the public methods and their respective signatures defined on the abstract class, we essentially have achieved OCP. Let's take a look at a simplified version of the Java code for Figure 1.2, focusing on how we achieve OCP, instead of on the actual method implementations.

```
public class Account {
    private AccountType _act;

    public Account(String act) {
        try {
            Class c = Class.forName(act);
            this._act = (AccountType) c.newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void deposit(int amt) {
        this._act.deposit(amt);
    }
}
```

Here, our `Account` class accepts as an argument to its constructor a `String` representing the class we wish to instantiate. It then uses the `Class` class to dynamically create an instance of the appropriate `AccountType` subclass. Note that we don't explicitly refer to either the `Savings` or `Checking` class directly.

```
public abstract class AccountType  {
    public abstract void deposit(int amt);
}
```

This is the abstract `AccountType` class that serves as the contract between our `Account` class and `AccountType` descendents. The `deposit` method is the contract.

```
public class CheckingAccount extends AccountType {
    public void deposit(int amt) {
        System.out.println();
        System.out.println();
        System.out.println("Amount deposited in checking account: "
                           + amt);
        System.out.println();
        System.out.println();
    }
}
public class SavingsAccount extends AccountType {
    public void deposit(int amt)  {
        System.out.println();
        System.out.println();
        System.out.println("Amount deposited in savings account: "
                            + amt);
        System.out.println();
        System.out.println();
    }
}
```

Each of our `AccountType` descendents satisfies the contract by providing an implementation for the `deposit` method. In the real world, the behaviors of the individual `deposit` methods would be more interesting and, given the preceding design, would be algorithmically different.

1.1.2 Liskov Substitution Principle (LSP)

Subclasses should be substitutable for their base classes.

We mentioned in our previous discussion that OCP is the most important of the class category principles. We can think of the Liskov Substitution Principle (LSP) as an extension to OCP. In order to take advantage of LSP, we must adhere to OCP because violations of LSP also are violations of OCP, but not vice versa. LSP is the work of Barbara Liskov and is derived from Bertrand Meyer's Design by Contract.[1] In its simplest form, LSP is difficult to differentiate from OCP, but a subtle difference does exist. OCP is centered around abstract coupling. LSP, while also heavily dependent on abstract coupling, is in addition heavily dependent on preconditions and postconditions, which is LSP's relation to Design by Contract, where the concept of preconditions and postconditions was formalized.

A *precondition* is a contract that must be satisfied before a method can be invoked. A *postcondition*, on the other hand, must be true upon method completion. If the precondition is not met, the method shouldn't be invoked, and if the postcondition is not met, the method shouldn't return. The relation of preconditions and postconditions has meaning embedded within an inheritance relationship that isn't supported within Java, outside of some manual assertions or nonexecutable comments. Because of this, violations of LSP can be difficult to find.

To illustrate LSP and the interrelationship of preconditions and postconditions, we need only consider how Java's exception-handling mechanism works. Consider a method on an abstract class that has the following signature:

[1]A concept that Bertrand Meyer built into the Eiffel programming language and discusses in *Object-Oriented Software Construction*. See [OOSC97].

```
public abstract deposit(int amt) throws InvalidAmountException
```

Assume in this situation that our `InvalidAmountException` is an exception defined by our application, is inherited from Java's base `Exception` class, and can be thrown if the amount we try to deposit is less than zero. By rule, when overriding this method in a subclass, we cannot throw an exception that exists at a higher level of abstraction than `InvalidAmountException`. Therefore, a method declaration such as the following isn't allowed:

```
public void deposit(int amt) throws Exception
```

This method declaration isn't allowed because the `Exception` class thrown in this method is the ancestor of the `InvalidAmountException` thrown previously. Again, we can't throw an exception in a method on a subclass that exists at a higher level of abstraction than the exception thrown by the base class method we are overriding. On the other hand, reversing these two method signatures would have been perfectly acceptable to the Java compiler. We can throw an exception in an overridden subclass method that is at a lower level of abstraction than the exception thrown in the ancestor. While this does not correspond directly to the concept of preconditions and postconditions, it does capture the essence. Therefore, we can state that any precondition stipulated by a subclass method can't be stronger than the base class method. Also, any postcondition stipulated by a subclass method can't be weaker than the base class method.

To adhere to LSP in Java, we must make sure that developers define preconditions and postconditions for each of the methods on an abstract class. When defining our subclasses, we must adhere to these preconditions and postconditions. If we do not define preconditions and postconditions for our methods, it becomes virtually impossible to find violations of LSP. Suffice it to say, in the majority of cases, OCP will be our guiding principle.

1.1.3 Dependency Inversion Principle (DIP)

Depend upon abstractions. Do not depend upon concretions.

The Dependency Inversion Principle (DIP) formalizes the concept of abstract coupling and clearly states that we should couple at the abstract level, not at the concrete level. In our own designs, attempting to couple at the abstract level can seem like overkill at times. Pragmatically, we should apply this principle in any situation where we're unsure whether the implementation of a class may change in the future. But in reality, we encounter situations during development where we know exactly what needs to be done. Requirements state this very clearly, and the probability of change or extension is quite low. In these situations, adherence to DIP may be more work than the benefit realized.

At this point, there exists a striking similarity between DIP and OCP. In fact, these two principles are closely related. Fundamentally, DIP tells us how we can adhere to OCP. Or, stated differently, if OCP is the desired end, DIP is the means through which we achieve that end. While this statement may seem obvious, we commonly violate DIP in a certain situation and don't even realize it.

When we create an instance of a class in Java, we typically must explicitly reference that object. Only after the instance has been created can we flexibly reference that object via its ancestors or implemented interfaces. Therefore, the moment we reference a class to create it, we have violated DIP and, subsequently, OCP. Recall that in order to adhere to OCP, we must first take advantage of DIP. There are a couple of different ways to resolve this impasse.

The first way to resolve this impasse is to dynamically load the object using the `Class` class and its `newInstance` method. However, this solution can be problematic and somewhat inflexible. Because DIP doesn't enable us to refer to the concrete class explicitly, we must use a `String` representation of the concrete class. For instance, consider the following:

```
Class c = Class.forName("SomeDescendent");
SomeAncestor sa = (SomeAncestor) c.newInstance();
```

In this example, we wish to create an instance of the class `SomeDescendent` in the first line but reference it as type `SomeAncestor` in the second line. This also was illustrated in the code samples in Section 1.1.1, earlier in this chapter. This is perfectly acceptable, as long as the `SomeDescendent` class is inherited, either directly or indirectly, from the `SomeAncestor` class. If it isn't, our application will throw an exception at runtime. Another more obvious problem occurs

Abstract Coupling

Abstract coupling is the notion that a class is not coupled to another concrete class or class that can be instantiated. Instead, the class is coupled to other base, or abstract, classes. In Java, this abstract class can be either a class with the abstract modifier or a Java interface data type. Regardless, this concept actually is the means through which LSP achieves its flexibility, the mechanism required for DIP, and the heart of OCP.

when we misspell the class of which we want an instance. Yet another, less apparent, obstacle eventually is encountered when taking this approach. Because we reference the class name as a string, there isn't any way to pass parameters into the constructor of this class. Java does provide a solution to this problem, but it quickly becomes complex, unwieldy, and error prone.

Another approach to resolving the object creation challenge is to use an object factory. Here, we create a separate class whose only responsibility is to create instances. This way, our original class, where the instance previously would have been created, stays clear of any references to concrete classes, which have been removed and placed in this factory. The only references contained within this class are to abstract, or base, classes. The factory does, however, reference the concrete classes, which is, in fact, a blatant violation of DIP. However, it's an isolated and carefully thought through violation and is therefore acceptable.

Keep in mind that we may not always need to use an object factory. Along with the flexibility of a factory comes the complexity of a more dynamic collaboration of objects. Concrete references aren't always a bad thing. If the class to which we are referring is a stable class, not likely to undergo many changes, using a factory adds unwarranted complexity to our system. If a factory is deemed necessary, the design of the factory itself should be given careful consideration. This factory pattern has many design variants, some of which are discussed later in this book (see Chapter 9).

Blatant Violation: A Good Thing?

At this point, you might be wondering how a blatant violation can be a good thing. Keep in mind that our goal should be to create a more highly maintainable system. The tools that enable us to create these types of systems are the principles discussed in this chapter. Therefore, it is important that each principle be given careful consideration and that violations of these principles are conscious design decisions. While an object factory may violate DIP, it does so at the expense of allowing another module within the application to adhere to OCP. Therefore, any changes are localized to the factory and should not impact its clients.

1.1.4 Interface Segregation Principle (ISP)

Many specific interfaces are better than a single, general interface.

Put simply, any interface we define should be highly cohesive. In Java, we know that an interface is a reference data type that can have method declarations, but no implementation. In essence, an interface is an abstract class with all abstract methods. As we define our interfaces, it becomes important that we clearly understand the role the interface plays within the context of our application. In fact, interfaces provide flexibility: They allow objects to assume the data type of the interface. Consequently, an interface is simply a role that an object plays at some point throughout its lifetime. It follows, rather logically, that when defining the operation on an interface, we should do so in a manner that doesn't accommodate multiple roles. Therefore, an interface should be responsible for allowing an object to assume a single role, assuming the class of which that object is an instance implements that interface.

While working on a project recently, an ongoing discussion took place as to how we would implement our data access mechanism. Quite a bit of time was spent designing a flexible framework that would allow uniform access to a variety of different data sources. These back-end data sources might come in the form of a relational database, a flat file, or possibly even another proprietary database. Therefore, our goal was not only to provide a common data access mechanism, but also to present data to any class acting as a data client in a consistent manner. Doing so clearly would decouple our data clients from the back-end data source, making it much easier to port our back-end data sources to different platforms without impacting our data clients. Therefore, we decided that all data clients would depend on a single Java interface, depicted in Figure 1.3, with the associated methods.

At first glance, the design depicted in Figure 1.3 seemed plausible. After further investigation, however, questions were raised as to the cohesion of the `RowSetManager` interface. What if classes implementing this interface were read-only and didn't need insert and update functionality? Also, what if the data client weren't interested in retrieving the data, but only in iterating its already retrieved internal data set? Exploring these questions a bit further, and carefully considering the Interface Segregation Principle (ISP), we found that it was meaningful to have a data structure that wasn't even dependent on a retrieve action at all. For instance, we may wish to use a data set that was cached in memory and wasn't dependent on an underlying physical data source. This led us to the design in Figure 1.4.

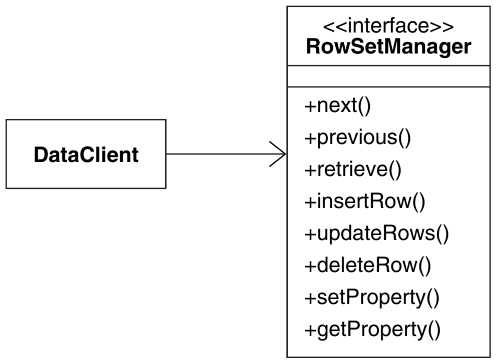

Figure 1.3 Violation of Interface Segregation Principle (ISP)

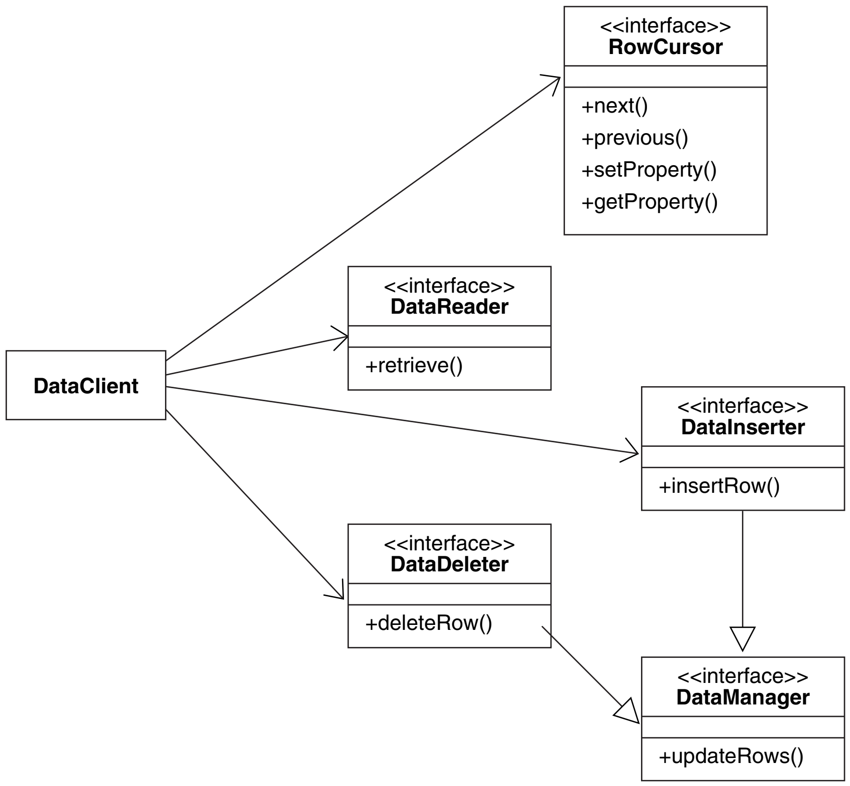

Figure 1.4 Compliance to Interface Segregation Principle (ISP)

In Figure 1.4, we see that we have segregated the responsibilities of our `RowSetManager` into multiple interfaces. Each interface is responsible for allowing a class to adhere to a cohesive set of responsibilities. Now our application can implement the interfaces necessary to provide the desired set of functionality. For example, we're no longer forced to provide data update behavior if our class is read-only.

1.1.5 Composite Reuse Principle (CRP)

Favor polymorphic composition of objects over inheritance.

The Composite Reuse Principle (CRP) prevents us from making one of the most catastrophic mistakes that contribute to the demise of an object-oriented system: using inheritance as the primary reuse mechanism. The first reference to this principle was in [GOF95]. For example, let's turn back to a section of our diagram in Figure 1.2. In Figure 1.5, we see the `AccountType` hierarchy with a few additional attributes and methods. In this example, we have added a method to the ancestor `AccountType` class that calculates the interest for each of our accounts. This approach seems to be a good one because both our `Savings` and `MoneyMarket` classes are interest-bearing accounts. Our `Checking` class is representative of an account that isn't interest bearing. Regardless, we justify this by convincing ourselves that it's better to define some default behavior on an ancestor and override it on descendents instead of duplicating the behavior across descendents. We know that we can simply define a null operation on our `Checking` class that doesn't actually calculate interest, and our problem is solved. While we do want to reuse our code, and we can prevent the `Checking` class from calculating interest, our implementation contains a tragic flaw. First, let's discuss the flaw and when it will surface. Then we'll discuss why this problem has occurred.

Let's consider a couple of new requirements. We need to support the addition of a new account type, called `Stock`. A `Stock` does calculate interest, but the algorithm for doing so is different than the default defined in our ancestor `AccountType`. That's easy to solve. All we have to do is override the `calculateInterest` in our new `Stock` class, just as we did in the `Checking` class, but instead of implementing a null operation, we can implement the appropriate algorithm. This works fine until our business realizes that the `Stock` class is doing extremely well, primarily because of its generous interest calculation mechanism. It's been decided that `MoneyMarket` should calculate interest using the same algorithm as `Stock`, but `Savings` remains the same. We have three choices in solving this problem. First, redefine the `calculateInterest` method on our `AccountType` to implement this new algorithm and define a new method on `Savings` that implements the older method. This option isn't ideal because it involves modifying at least two of our existing system classes, which is a blatant violation of OCP. Second, we could simply override `calculateInterest` on our `MoneyMarket` class, copy the code from our `Stock` class, and paste it in our `MoneyMarket calculateInterest` method. Obviously, this option isn't a very flexible solution. Our goal in reuse is not copy and paste. Third, we can define a new class called `InterestCalculator`, define a `calculateInterest` method on this class that implements our new algorithm, and then delegate the calculation of interest from our `Stock` and `MoneyMarket` classes to this new class. So, which option is best?

The third solution is the one we should have used up front. Because we realized that the calculation of interest wasn't common to all classes, we shouldn't have defined any default behavior in our ancestor class. Doing so in any situation inevitably results in the previously described outcome. Let's now resolve this problem using CRP.

In Figure 1.6, we see a depiction of our class structure utilizing CRP. In this example, we have no default behavior defined for `calculateInterest` in our

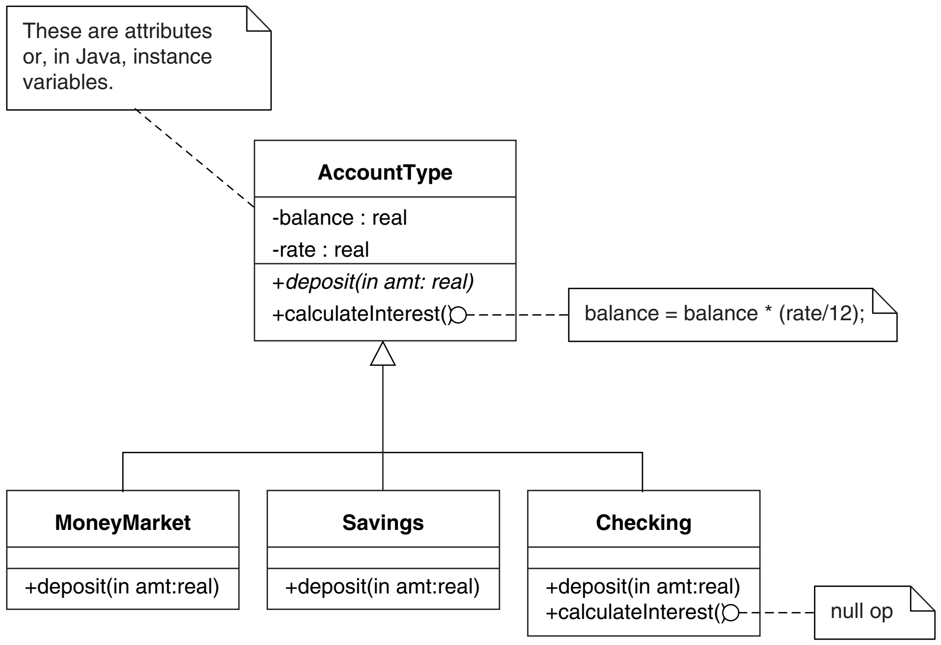

Figure 1.5 Account Structure with New Methods

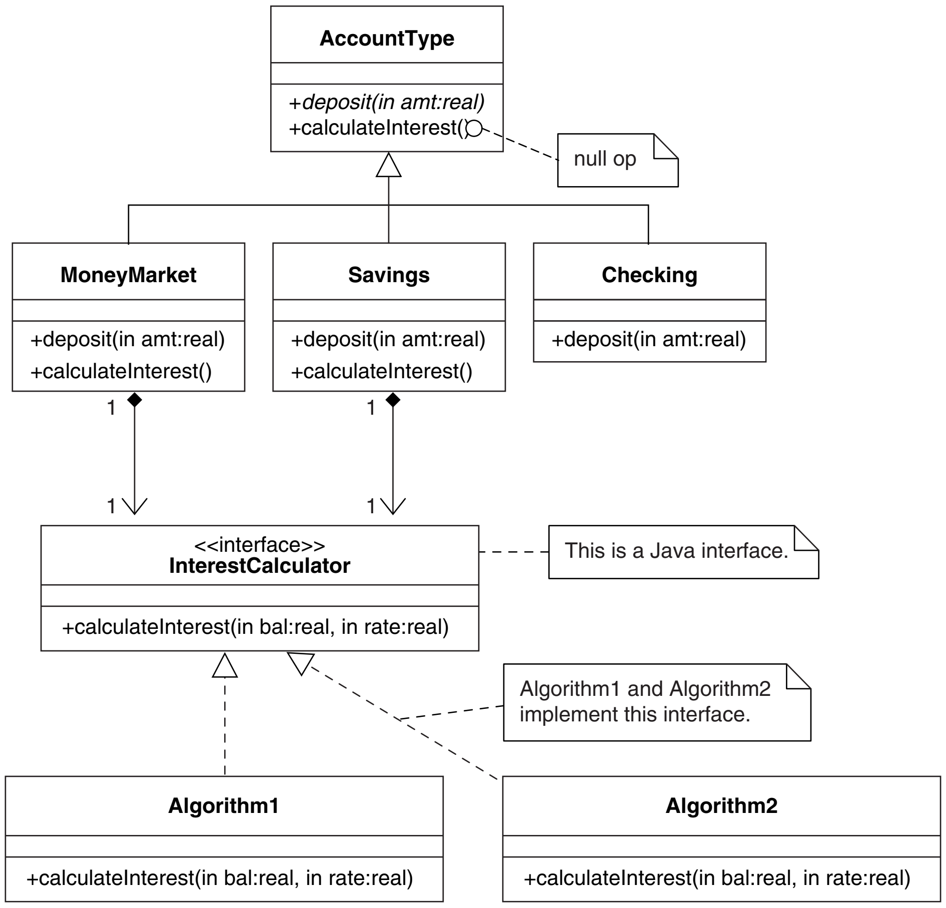

Figure 1.6 Compliance to Composite Reuse Principle (CRP)

`AccountType` hierarchy. Instead, in our `calculateInterest` methods on both our `MoneyMarket` and `Savings` classes, we defer the calculation of interest to a class that implements the `InterestCalculator` interface. When we add our `Stock` class, we now simply choose the `InterestCalculator` that is applicable for this new class or define a new one if it's needed. If any of our other classes need to redefine their algorithms, we can do so because we are abstractly coupled to our interface and can substitute any of the classes that implement the interface anywhere the interface is referenced. Therefore, this solution is ultimately flexible in how it enables us to calculate interest. This is an example of CRP. Each of our `MoneyMarket` and `Savings` classes are composed of our

InterestCalculator, which is the composite. Because we are abstractly coupled, we easily see we can receive polymorphic behavior. Hence, we have used polymorphic composition instead of inheritance to achieve reuse.

At this point, you might say, however, that we still have to duplicate some code across the Stock and MoneyMarket classes. While this is true, the solution still solves our initial problem, which is how to easily accommodate new interest calculation algorithms. Yet an even more flexible solution is available, and one that will enable us to be even more dynamic in how we configure our objects with an instance of InterestCalculator.

In Figure 1.7, we have moved the relationship to InterestCalculator up the inheritance hierarchy into our AccountType class. In fact, in this scenario, we are back to using inheritance for reuse, though a bit differently. Our AccountType knows that it needs to calculate interest, but it doesn't know how actually to do it. Therefore, we see a relationship from AccountType to our InterestCalculator. Because of this relationship, all accounts calculate interest. However, if one of our algorithms is a null object [PLOP98] (that is, it's an instance of a class that implements the interface and defines the methods, but the methods have no implementation), and we use the null object with the Savings class, we now can state that all of our accounts need to calculate interest. This substantiates our use of implementation inheritance. Because each account calculates it differently, we configure each account with the appropriate InterestCalculator.

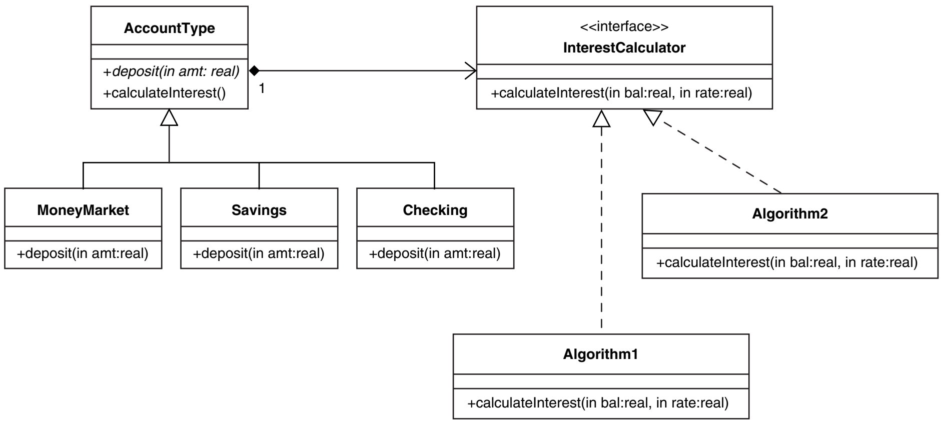

Figure 1.7 Refining CRP Compliance with Ancestral Relationship

So how did we fall into the original trap depicted in Figure 1.5? The problem lies within the inheritance relationship. Inheritance can be thought of as a generalization over a specialization relationship—that is, a class higher in the inheritance hierarchy is a more general version of those inherited from it. In other words, any ancestor class is a partial descriptor that should define some default characteristics that are applicable to any class inherited from it. Violating this convention almost always results in the situation described previously. In fact, any time we have to override default behavior defined in an ancestor class, we are saying that the ancestor class is not a more general version of all of its descendents but actually contains descriptor characteristics that make it too specialized to serve as the ancestor of the class in question. Therefore, if we choose to define default behavior on an ancestor, it should be general enough to apply to all of its descendents.

In practice, it's not uncommon to define a default behavior in an ancestor class. However, we should still accommodate CRP in our relationships. This is easy to see in Figure 1.6. We could have easily defined default behavior in our `calcuateInterest` method on the `AccountType` class. We still have the flexibility, using CRP, to alter the behaviors of any of our `AccountType` classes because of the relationship to `InterestCalculator`. In this situation, we may even choose to create a null op `InterestCalculator` class that our `Checking` class uses. This way, we even accommodate the likelihood that `Savings` accounts can someday calculate interest. We have ultimate flexibility.

1.1.6 Principle of Least Knowledge (PLK)

For an operation O on a class C, only operations on the following objects should be called: itself, its parameters, objects it creates, or its contained instance objects.

The Principle of Least Knowledge (PLK) is also known as the Law of Demeter. The basic idea is to avoid calling any methods on an object where the reference to that object is obtained by calling a method on another object. Instead, this principle recommends we call methods on the containing object, not to obtain a reference to some other object, but instead to allow the containing object to forward the request to the object we would have formerly obtained a reference to. The primary benefit is that the calling method doesn't need to understand the structural makeup of the object it's invoking methods upon. The following examples show a violation of PLK and an implementation that does not violate PLK:

```
//violation of PLK
public class Sample {
    public void lawTest(AnObject o) {
        AnotherObject ao = o.get();
        ao.doSomething();
    }
}
```

```
//adherence to PLK. Note that AnObject
//would forward the doSomething request
//on to AnotherObject, which it con-
tains.
public class Sample {
    public void lawTest(AnObject o) {
        o.doSomething();
    }
}
```

The obvious disadvantage associated with PLK is that we must create many methods that only forward method calls to the containing classes internal components. This can contribute to a large and cumbersome public interface. An alternative to PLK, or a variation on its implementation, is to obtain a reference to an object via a method call, with the restriction that any time this is done, the type of the reference obtained is always an interface data type. This is more flexible because we aren't binding ourselves directly to the concrete implementation of a complex object, but instead are dependent only on the abstractions of which the complex object is composed. In fact, this is how many classes in Java typically resolve this situation.

Consider the `java.sql.ResultSet` interface. After an SQL statement has been executed, Java stores the SQL results in a `ResultSet` object. One of our options at this point is to query this `ResultSet` object and obtain metainformation pertaining to this set of data. The class that contains this metainformation is the `ResultSetMetaData` class, and it's contained within the `ResultSet` class. If PLK were adhered to in this situation, we wouldn't directly obtain a reference to this `ResultSetMetaData` class, but instead would call methods on the `ResultSet`, which subsequently would forward these requests to the `ResultSetMetaData` class. However, this would result in an explosion of methods on the `ResultSet` class. Therefore, a `getResultSetMetaData` method on `ResultSet` does return a reference to `ResultSetMetaData`. At first, this would seem to be a blatant violation of PLK. However, `ResultSetMetaData` is an interface data type and, therefore, we aren't bound to any concrete implementation contained within `ResultSet`. Instead, we're coupled only to the abstractions of which `ResultSet` is composed.

This solution is a perfectly acceptable alternative to a direct implementation of PLK. The caveat is that careful consideration should be given to DIP. As long as this is done, we shouldn't have increased maintenance problems. The most important aspect is that we're bound, or coupled, to the internal structure of a class at an abstract level. Therefore, the class that is obtaining the reference to the object via the method call is taking advantage of DIP and, subsequently, OCP.

1.2 Package Principles

Throughout the course of development, it's common for development teams to spend a chunk of time designing the system. Much of this time, however, is spent creating a flexible class structure, with little time actually being devoted to the system's package structure. The relationships between packages typically aren't considered, and the allocation of classes to packages isn't carefully thought through. This carelessness is unfortunate because the relationships between packages are just as important as the relationships between the classes. The relationships between the packages of an application are referred to as the *package dependencies*, and we next examine principles that help to create a more robust dependency structure between our packages.

1.2.1 Package Dependency

It isn't uncommon to find that many developers haven't realized that relationships do exist among the packages within a Java application. The dependencies between packages often go unnoticed. Logically, however, if a class contains relationships to other classes, then packages containing those classes also must contain relationships to other packages. These package relationships can tell us a great deal about the resiliency of our system, and the principles discussed in Sections 1.2.2 through 1.2.7 enable us to more objectively measure the robustness of our package relationships.

First, let's examine what is meant by a *package dependency*. In Figure 1.8, we see a class diagram depicting two packages, `A` and `B`. Within each of these packages exist two classes. Class `Client` exists in package `A` and class `Service` in `B`. Simply stated, if class `Client` references in any way class `Service`, then it must hold true that `Client` has a structural relationship to class `Service`, which implies that any changes to the `Service` class may impact `Client`. Figure 1.8

A Subtle Relation

If class `Client` has a relation to class `Service`, then it's obvious that the packages containing these two classes also have a relationship, formally known as a *package dependency*. It's not so obvious that these class and package relationships can be considered two separate views of our system. One is a higher-level package view, the other a lower-level class view. In addition, these two views serve to validate each other. You'll find information on this subject in Chapter 10.

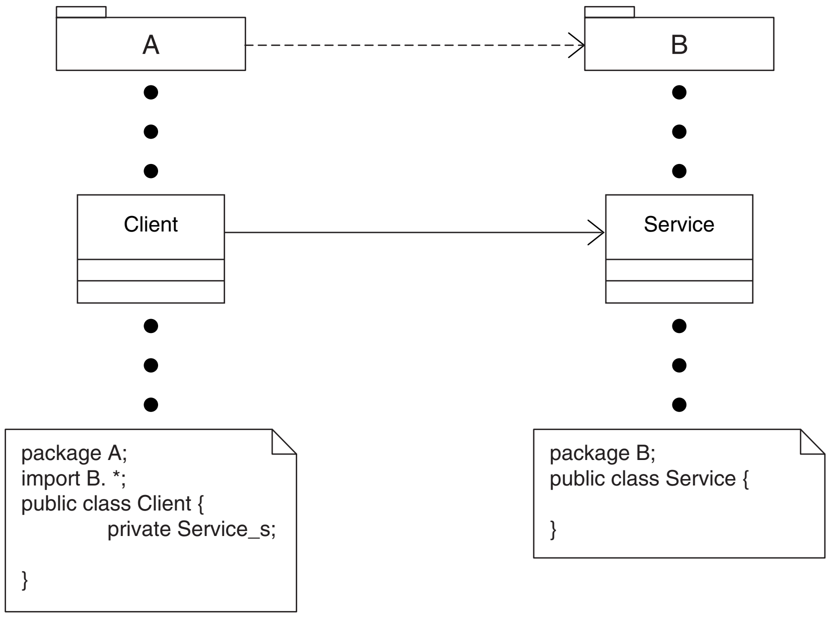

Figure 1.8 Package and Corresponding Class Relationships

illustrates how this relationship exists between packages, classes, and source code.

Let's examine this relationship from a different viewpoint. If the contents of package A are dependent on the contents of package B, then A has a dependency on B; and if the contents of B change, this impact may be noticeable in A. Therefore, the relationships between packages become more apparent, and we can conclude the following:

> *If changing the contents of a package P1 may impact the contents of another package P2, we can say that P1 has a package dependency on P2.*

Packages may contain not only classes, however, but also packages. In Java, importing the contents of a package implies we have access only to the classes within that package and don't have access to classes in any nested packages. The Unified Modeling Language (UML), however, doesn't take any formal position on nested packages. The question of how to deal with nested packages is left to the development team. We use the terms *opaque* and *transparent* to define the two options. *Opaque visibility* implies that a dependency on a package with nested packages doesn't imply access to these nested packages. *Transparent visibility*, on the other hand, does carry with it implicit dependencies.

Because the UML takes no formal position, development teams must define how they wish to deal with package dependencies. Several options are available. First, teams may take their own position and state that all package dependencies are either opaque or transparent. Any variation from this norm must be modeled explicitly. In situations such as these, we recommend selecting opaque. Adopting transparent visibility doesn't enable us to restrict access to nested packages. On the other hand, if opaque is adopted as a standard, we can always explicitly model relations to nested packages on separate diagrams. For purposes of discussion throughout this book, we assume all package dependency relationships are opaque.

An alternative approach is to create stereotypes that can be attached to the dependency relation. Consequently, visibility is determined by the stereotype attached to the dependency. Some obvious pitfalls include those relationships with no stereotype attached. Unless a default is assumed, we cannot know what the transparency is, and making any assumptions can be dangerous. In addition, because only a single stereotype can be attached to any modeling element, we may be forced to make a decision if other stereotypes are being considered for the same dependency relationship. Let's now turn our attention to the discussion of the package principles.

1.2.2 Release Reuse Equivalency Principle (REP)

The granule of reuse is the granule of release.

Whenever a client class wishes to use the services of another class, we must reference the class offering the desired services. This should be apparent from our previous discussions and is the basis upon which package relationships exist. If the class offering the service is in the same package as the client, we can reference that class using the simple name. If, however, the service class is in a different package, then any references to that class must be done using the class' fully qualified name, which includes the name of the package.

We also know that any Java class may reside in only a single package. Therefore, if a client wishes to utilize the services of a class, not only must we reference the class, but we must also explicitly make reference to the containing package. Failure to do so results in compile-time errors. Therefore, to deploy any class, we must be sure the containing package is deployed. Because the package is deployed, we can utilize the services offered by any public class within the package. Therefore, while we may presently need the services of only a single class in the containing package, the services of all classes are available to us. Consequently, our unit of release is our unit of reuse, resulting in the Release Reuse Equivalency Principle (REP). This leads us to the basis for this principle, and it should now be apparent that the packages into which classes are placed have a tremendous impact on reuse. Careful consideration must be given to the allocation of classes to packages.

1.2.3 Common Closure Principle (CCP)

Classes that change together, belong together.

The basis for the Common Closure Principle (CCP) is rather simple. Adhering to fundamental programming best practices should take place throughout the entire system. Functional cohesion emphasizes well-written methods that are more easily maintained. Class cohesion stresses the importance of creating classes that are functionally sound and don't cross responsibility boundaries. And package cohesion focuses on the classes within each package, emphasizing the overall services offered by entire packages.

During development, when a change to one class may dictate changes to another class, it's preferred that these two classes be placed in the same package. Conceptually, CCP may be easy to understand; however, applying it can be difficult because the only way that we can group classes together in this manner is when we can predictably determine the changes that might occur and the effect that those changes might have on any dependent classes. Predictions often are incorrect or aren't ever realized. Regardless, placement of classes into respective packages should be a conscious decision that is driven not only by the relationships between classes, but also by the cohesive nature of a set of classes working together.

1.2.4 Common Reuse Principle (CReP)

Classes that aren't reused together should not be grouped together.

If we need the services offered by a class, we must import the package containing the necessary classes. As we stated previously in our discussion of REP (see Section 1.2.2), when we import a package, we also may utilize the services offered by any public class within the package. In addition, changing the behavior of any class within the service package has the potential to break the client. Even if the client doesn't directly reference the modified class in the service package, other classes in the service package being used by clients may reference the modified class. This creates indirect dependencies between the client and the modified class that can be the cause of mysterious behavior. In fact, we can state the following:

> *If a class is dependent on another class in a different package, then it is, in fact, dependent on all classes in that package, albeit indirectly.*

This principle has a negative connotation. It doesn't hold true that classes that are reused together should reside together, depending on CCP. Even though classes may always be reused together, they may not always change together. In striving to adhere to CCP, separating a set of classes based on their likelihood to change together should be given careful consideration. Of course, this impacts REP because now multiple packages must be deployed to use this functionality. Experience tells us that adhering to one of these principles may impact the ability to adhere to another. Whereas REP and Common Reuse Principle (CReP) emphasize reuse, CCP emphasizes maintenance.

1.2.5 Acyclic Dependencies Principle (ADP)

The dependencies between packages must form no cycles.

Cycles among dependencies of the packages composing an application should almost always be avoided. In other words, packages should form a directed acyclic graph (DAG). In Figure 1.9, we see two separate diagrams illustrating the relationships among Java packages. First, let's explore what these relationships imply; then we will explain why we want to avoid cyclic dependencies. In the diagram at the left of Figure 1.9, package `A` has a dependency on package `B`, and package `B` has a dependency on package `A`. In Java, this implies that some class in package `A` imports package `B` and uses a class in package `B`. Also, some class in package `B` imports package `A` and uses some class. The following code illustrates this scenario:

```
package A;
import B.*;
public class SomeAClass {
    private ClassInB b;
}
```

```
package B;
import A.*;
public class SomeBClass {
    private ClassInA a;
}
```

c01.qxd p001-038 11/20/01 9:47 AM Page 28

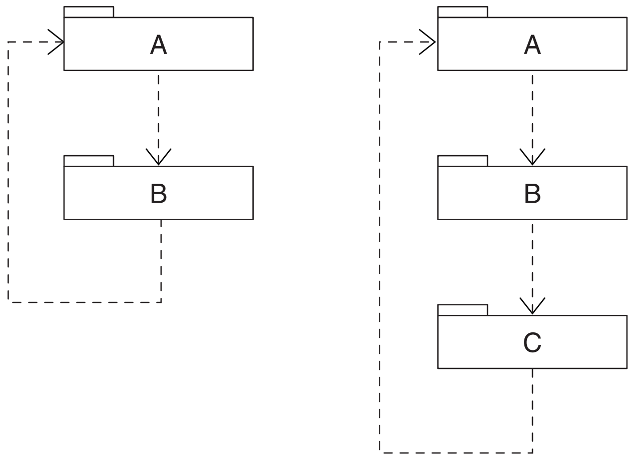

Figure 1.9 Violation of Acyclic Dependencies Principles (ADP)

The problem with this code is that, because the classes in these packages are coupled, the two packages become tightly coupled, which has a tremendous impact on REP. If some class `C` in a different package, call it `X`, uses `SomeBClass` in package `B`, it definitely implies that when package `B` is deployed, package `A` also must be deployed because `SomeBClass` is coupled to `SomeAClass` in package `A`. Neglecting to deploy package `A` with `B` results in runtime errors. In fact, were an application to have cyclic dependencies among the packages that compose it, REP would be so negatively impacted that all of these classes may as well exist in the same package. Obviously, we wouldn't desire this degree of coupling because CCP also would be severely compromised. Regardless, when developing Java applications, we should rarely find ourselves in a situation where we have violated the Acyclic Dependencies Principle (ADP). The consequences of doing so are dire, and we should avoid it at all costs.

If we do identify cyclic dependencies, the easiest way to resolve them is to factor out the classes that cause the dependency structure. This is exactly what we have done in Figure 1.10. Factoring out the classes that caused the cyclic dependencies has a positive impact on reuse. Now, should we decide to reuse package `B` in our previous example, we still need to deploy package `` A` ``, but we don't need to deploy package `A`. The impact of this situation is not fully realized until we take into consideration more subtle cycle dependencies, such as the indirect cyclic dependency illustrated at the right in Figure 1.9, and its subsequent resolution in the diagram at right in Figure 1.10.

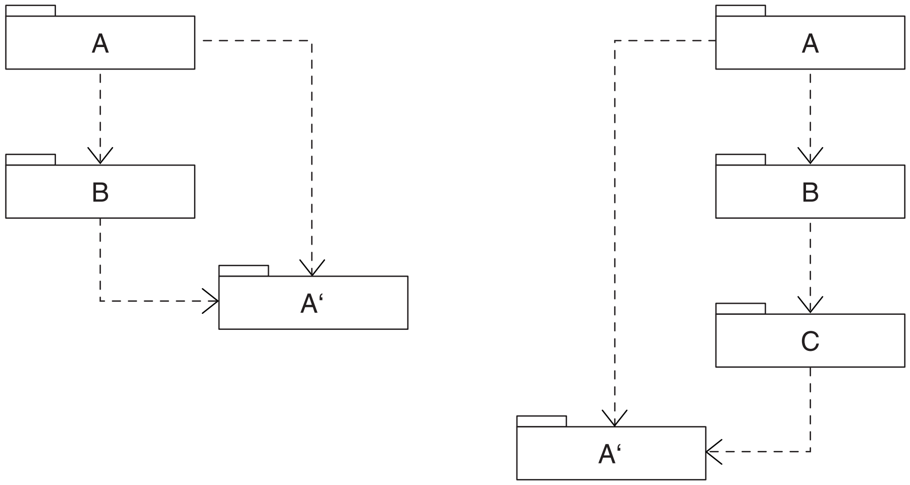

Figure 1.10 Acyclic Dependencies Principles (ADP) Compliance

1.2.6 Stable Dependencies Principle (SDP)

Depend in the direction of stability.

At first glance, the Stable Dependencies Principle (SDP) seems to be stating the obvious. However, exploring more deeply, we find the SDP contains an interesting underlying message. In the context of software development, stability often is used to describe a system that is robust, bug free, and rich in structure. In a more general sense, stability implies that an item is fixed, permanent, and unvarying. Attempting to change an item that is stable is more difficult than inflicting change on an item in a less stable state. Applying this richer meaning of stability to software implies that stable software is difficult to change. Before we revolt, however, let's point out that simply because software is stable doesn't mean that it's riddled with bugs. Stable software can certainly be robust, bug free, and rich in structure. Subsequently, the stability of our software system isn't necessarily related to its quality. Less stable software can be of high quality, yet it also can easily experience change. Stability is a characteristic indicating the ease with which a system can undergo change, and with Java, we are most concerned with the resiliency of our packages.

At this point, it's useful to ask what makes a package difficult to change. Aside from poorly written code, the degree of coupling to other packages has a dramatic impact on the ease of change. Those packages with many incoming dependencies have many other components in our application dependent on them. These more stable packages are difficult to change because of the far-reaching consequences the change may have throughout all other dependent packages. On the other hand, packages with few incoming dependencies are easier to change. Those packages with few incoming dependencies most likely will have more outgoing dependencies. A package with no incoming or outgoing dependencies is useless and isn't part of an application because it has no relationships. Therefore, packages with fewer incoming, and more outgoing dependencies, are less stable. Referring again to Figure 1.10, we can say that package `A`` is a more stable package, whereas package `A` is a less stable package, taking into consideration only the ease with which either of these packages can undergo change.

In previous sections, we've discussed that our software should be resilient and easily maintainable. Because of this, our assumptions lead us to believe that all software should be less stable, but this belief isn't always correct. Stability doesn't provide any implication as to the frequency with which the contents of a package change. Those packages having a tendency to change more often should be the packages that are less stable in nature. On the other hand, packages unlikely to experience change may be more stable, and it's in this direction that we should find the dependency relations flowing. Combining the concepts of stability, frequency of change, and dependency management, we're able to conclude the following:

- Packages likely to experience frequent change should be less stable, implying fewer incoming dependencies and more outgoing dependencies.
- Packages likely to experience infrequent change may be more stable, implying more incoming dependencies and fewer outgoing dependencies.

It should now be obvious that we naturally depend in the direction of stability because the direction of our dependency makes the packages more or less stable. Any dependency introduced, however, should be a conscious decision, and one that we know may have a dramatic impact on the stability of our application. Ideally, dependencies should be introduced only to packages that are more stable. The conscious nature of this decision is captured by our next principle, which describes the technique we employ to create more stable or less stable packages.

Up to this point, we've been carefully referring to the stability of packages as either more stable or less stable. Packages are typically not characterized as stable or unstable. Instead, stability is a metric that can be measured and is a numerical value between 0 and 1. The stability of a package can be measured using some fairly straightforward calculations. Consider the following formula:

where $$I = \frac{Ce}{Ca + Ce}$$

I represents the degree of instability associated with the package.

Ca represents the number of external classes dependent on classes internal to this package.

Ce represents the number of internal classes dependent on classes not internal.

A package becomes more stable as I approaches 0 because this implies no outgoing dependencies. As I approaches 1, a package is less stable. Less stable packages have fewer incoming dependencies, whereas more stable packages have more incoming dependencies.

1.2.7 Stable Abstractions Principle (SAP)

Stable packages should be abstract packages.

Assuming we do wish to depend in the direction of stability, we're left with no choice but to structure packages so that the less stable packages exist atop a package hierarchy, and more stable packages exist at the bottom of our package hierarchy. The diagram in Figure 1.10 is indicative of this relationship. At this point, it's extremely important that the packages that are lower in our package hierarchy must be the most resilient packages in our system, because of the far-reaching consequences of changing them.

As we've discussed, one of the greatest benefits of object orientation is the ability to easily maintain our systems. The high degree of resiliency and maintainability is achieved through abstract coupling. By coupling concrete classes to abstract classes, we can extend these abstract classes and provide new system functions without having to modify existing system structure. Consequently, the means through which we can depend in the direction of stability, and help ensure that these more depended-upon packages exhibit a higher degree of stability, is to place abstract classes, or interfaces, in the more stable packages. We can now state the following:

- More stable packages, containing a higher number of abstract classes, or interfaces, should be heavily depended upon.

- Less stable packages, containing a higher number of concrete classes, should not be heavily depended upon.

A simple metric can help determine the degree of abstractness associated with a package. Consider the following formula:

$$A = \frac{Na}{Nc}$$

where

A is the abstractness of the package.
Na is the number of abstract classes and interfaces.
Nc is the number of overall classes and interfaces.

Values of A approaching 0 imply a package has few abstract classes. Values of A approaching 1 imply a package consists of almost entirely abstract classes and interfaces.

It is ideal if the abstractness of a package is either 1 or 0 and as far away from 0.5 as possible. A value of 0.5 implies that a package contains both abstract and concrete classes and, therefore, is neither stable nor instable. A goal of all packages should be a high degree or low degree of abstractness, depending heavily upon its role within the application.

It now should be apparent that any packages containing all abstract classes with no incoming dependencies are utterly useless. On the other hand, packages containing all concrete classes with many incoming dependencies are extremely difficult to maintain. Therefore, in terms of SDP and the Stable Abstractions Principle (SAP), we can only conclude that as abstractness (A) increases, instability (I) decreases.

1.3 Patterns

Any discussion of patterns could easily fill multiple texts. This section doesn't even attempt to define a fraction of the patterns that can be useful during development. Instead, we emphasize the common components of a pattern, as well as introduce a few common patterns that have multiple uses. As the discussion continues throughout this book, additional patterns are introduced as the need warrants. The discussion in this section serves two purposes. First, we describe the intent of the patterns, a few problems that they might help resolve, and some consequences of using the pattern. This discussion should help in understanding how patterns can be used and the context in which they might be useful. Second, and most important for this discussion, we explore the consistent nature with which the principles previously discussed resurface within these patterns.

This topic is important because, as mentioned previously, patterns may not always be available for the specific problem domain or, if available, may possibly be unknown. In these situations, a reliance upon some other fundamental principles is important.

1.3.1 Strategy

Undoubtedly, the Strategy pattern is one of the simplest patterns and is a direct implementation of OCP. The purpose of a Strategy, as stated in [GOF95], is to

> *Define a family of algorithms, encapsulate each one, and make them interchangeable.*

In fact, the `InterestCalculator` class in Figure 1.7 is a Strategy. The individual `Algorithm` classes encapsulate the various interest calculations. These are our family of algorithms, and they are made interchangeable by implementing the `InterestCalculator` interface. This is the structural aspect of the Strategy. The behavioral aspects of a Strategy are a bit more interesting and are typically discussed in the context of the consequences that result as the application of that pattern. For instance, where does the concrete Strategy instance get created? Creating it within the client class removes many of the advantages of using Strategy, which becomes more apparent when considering the coupling that exists between the client class and the concrete Strategy classes. In Figure 1.7, if the `AccountType` class actually created the `InterestCalculator` Strategy, the `AccountType` class would have to be modified each time a new `Algorithm` class was added to our system. This solution isn't ideal and, in fact, doing so violates OCP, even though Strategy attempts to achieve OCP. A better approach may be for a separate class to create the concrete Strategy. Structuring the system in this manner is common, and the end result is the incorporation of a Factory pattern into the system. The sole responsibility of the Factory pattern, introduced in Chapter 9, is to create instances of objects. At this point, it could be stated that even though a Factory is used, OCP still is violated because any new concrete Strategy classes now require a modification of the Factory. While this statement is true, careful consideration should be given to the ease with which this maintenance has been achieved versus the strict adherence to a principle. While there are many alternatives to this approach, we've found that using a Factory in this situation is not only easily understood, but easily maintainable as well. In fact, while it may be a small violation of OCP, the points within our application that refer directly to the concrete Strategy classes are so small in number (one) that we don't even consider it a violation of OCP.

Also note that when using a Strategy, we have to determine when and where to use it. Obviously, numerous places could take advantage of a Strategy. The trick is to keep OCP in mind. Does the system need to have this flexibility at this point? The intent is not to use Strategy, or any other pattern for that matter, anywhere that it could be used, but to use the appropriate pattern in the appropriate context. Should the context call for this degree of flexibility, a Strategy should be considered. Were we not familiar with the Strategy pattern, nor any other pattern that accommodated the need, reliance upon the fundamental principles would have yielded similar results. In fact, our discussion in Section 1.1.5 resulted in the derivation of the Strategy pattern, prior to ever having heard of Strategy.

1.3.2 Visitor

The Visitor pattern is not widely used, yet it serves as an excellent example illustrating the true power of the object-oriented paradigm. The discussion up to this point has focused on the fact that the object-oriented paradigm allows a system to be easily extended with new classes, without having to modify the existing system. The structured paradigm didn't accommodate this need in the same flexible manner. In fact, it already has been clearly illustrated in Figure 1.7 that a system can be extended without having to modify the existing system. What if, however, we want to add new operations to an existing set of classes? This task is a more difficult one because defining a new method on a class presents huge obstacles if the system has made any attempt whatsoever to adhere to the previously discussed principles. The problem is that most classes are reliant upon interfaces, and the operations defined on the interface serve as the contract between the client class and the service class. This is especially true with DIP, and changing the interface results in a broken contract that can be remedied only by correcting either the client or service class and conforming to the new contract. A mechanism enabling us to attach new responsibilities to a class without having to change the class on which those responsibilities operate would be extremely valuable.

In the most rudimentary sense, consider a class that has a dynamic interface. It may be extremely difficult to determine what methods should actually reside on that class. As development progresses, and new operations are discovered, the system requires constant maintenance to change the interface. In Figure 1.11, such a class is presented. The `DynamicClass`, however, has only a single generic method named `accept`. This method accepts a parameter of type `Visitor`. Consequently, the `DynamicClass` receives an instance of a concrete `Visitor` at runtime. Each of the concrete `Visitor` classes represents a method that would have normally been found on the `DynamicClass`. We can easily extend the func-

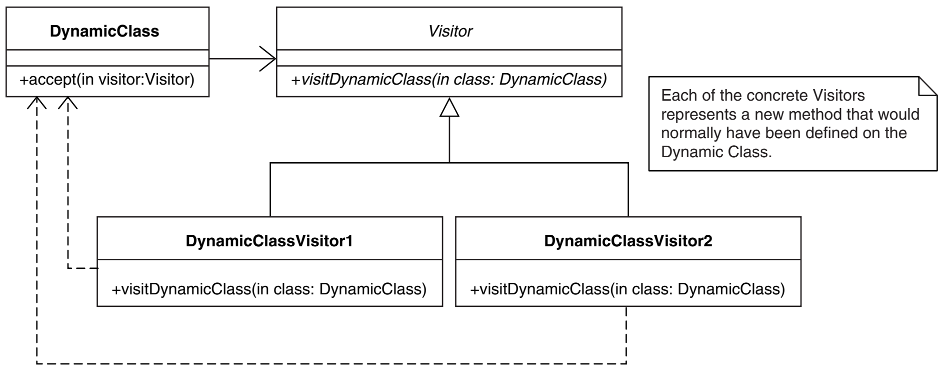
DynamicClass
+accept(in visitor:Visitor)
Visitor
+visitDynamicClass(in class: DynamicClass)
Each of the concrete Visitors represents a new method that would normally have been defined on the Dynamic Class.
DynamicClassVisitor1
+visitDynamicClass(in class: DynamicClass)
DynamicClassVisitor2
+visitDynamicClass(in class: DynamicClass)

Figure 1.11 Visitor Pattern

tionality provided by `DynamicClass` by defining a new concrete `Visitor` for each method. In addition, any client of `DynamicClass` always will be dependent only on the generic `accept` method. As is now evident, it's easy to add new operations in the form of concrete `Visitor` classes without having to modify existing system components.

In fact, the Visitor is structurally similar to the Strategy. However, the intent as discussed is radically different. In fact, this is a major point that should be addressed in any discussion on patterns. While some patterns may appear to be structurally similar, the behavioral aspects of different patterns typically differ radically. Therefore, it becomes extremely important to understand the dynamic aspects of the challenge presented. The behavioral and structural differences in the context of a system as a whole are introduced in Chapter 4 and discussed in detail in Chapters 8 and 9.

Examining the Visitor pattern a bit further, a number of principles do surface. For instance, OCP has been adhered to; however, it comes in a different form. In this situation, it's used to support adding new methods to a class, not to support adding new data structures to a system. In addition to OCP, CRP is also used because the `DynamicClass` is composed of the various `Visitor` classes. The identification of additional principles is left as an exercise to the reader.

Up to this point, the Visitor pattern sounds fairly useful. However, as mentioned previously, the Visitor pattern isn't often used because of one major implication that it has upon our system. While the Visitor makes it easy to add new methods to the interface of a class, it makes it extremely difficult to add new classes. For instance, in Figure 1.11, consider the modifications required to the system should a new `DynamicClass` class be required. Creating the new `DynamicClass` class would be easy. It involves simply creating a new class with an `accept` method. The problem resides in the proliferation of changes that exist within our `Visitor` hierarchy. These modifications demand that a new method be added to all `Visitor` classes in the hierarchy, including the abstract `Visitor` class. Because of this, the Visitor pattern is somewhat limited in use. In fact, caution should be used any time an implementation of the Visitor pattern is considered. Interested readers should refer to [GOF95] for further reading on Visitor.

1.3.3 Layers

In Java, a class can belong to only a single package. Therefore, if classes in different packages have relationships to each other, this implies that packages have structural relationships among them as well. Stated more precisely, if a class `C1` in package `P1` has a relationship to a class `C2` in package `P2`, we can say that the package `P1` has a relationship to package `P2`. The Layers pattern focuses on the relationships that exist between packages. In layering an application, a goal is to create packages that exist at higher-level layers and are dependent on packages that exist in lower-level layers.

For instance, a common approach to layering an application is to define a package containing all presentation, or user interface, classes; another containing domain classes, or business objects; and another containing an application kernel, which may consist of database access classes. Each of these packages exists at different layers, and the relationships between the classes contained within each package are driven by the relationships allowed between the individual layers. The caveat of layering our application is that no package existing at a lower-level layer can be dependent on any package existing at a higher-level layer. This is important and is the defining characteristic of a layered application. In fact, this pattern is an implementation of ADP. However, while layering an application may seem obvious, successfully doing so can be tedious. For now, we defer any in-depth discussion on layering to Chapter 10, where we discuss architectural modeling and its various consequences. Our purpose has been served in this chapter by illustrating that the Layers pattern is supportive of the aforementioned ADP.

1.4 Conclusion

The object-oriented paradigm goes beyond the fundamental concepts and terms associated with it. Of course, while understanding core terms such as *polymorphism*, *encapsulation*, and *inheritance* is important, understanding the essence of the paradigm and pragmatically applying fundamental concepts is vital. In this chapter, we introduced a set of principles that will serve as a guide throughout the various phases of the software lifecycle, and the remainder of this book, helping to ensure our designs are more resilient, robust, and maintainable.

Whereas principles provide a reliable foundation for our design efforts, patterns can raise the level of abstraction, providing a more concrete starting point, and allowing our designs to grow more quickly. Many benefits are associated with taking advantage of design patterns. Because of this popularity, however, a proliferation of patterns has saturated the marketplace, making it difficult to separate the more useful patterns from those less so. In such situations, emphasizing the principles can help produce a more reliable and desired outcome. Regardless, both principles and patterns will be given the majority of our attention as we move through our chapters.