Chapter 1

In OO design, the attributes and behaviors are contained within a single object, whereas in procedural, or structured, design the attributes and behaviors are normally separated.

Encapsulation is defined by the fact that objects contain both attributes and behaviors. Data hiding is a major part of encapsulation.

For data hiding to work properly, all attributes should be declared as private. Thus, attributes are never part of the interface. Only the public methods are part of the class interface. Declaring an attribute as public breaks the concept of data hiding

---

If you control the access to the attribute, when a problem arises, you do not have to worry about tracking down every piece of code that might have changed the attribute—it can be changed in only one place (the setter).

From a security perspective, you don't want uncontrolled code to change or retrieve sensitive data. For example, when you use an ATM, access to data is controlled by asking for a PIN.

When you tell somebody to draw a shape, the first question asked is, "What shape?" No one can draw a shape, because it is an abstract concept (in fact, the draw method in the Shape code following contains no implementation). You must specify a concrete shape. To do this, you provide the actual implementation in Circle. Even though Shape has a draw method, Circle overrides this method

and provides its own draw method. Overriding basically means replacing an implementation of a parent with one from a child.

• **Encapsulation**—Encapsulating the data and behavior into a single object is of primary importance in OO development. A single object contains both its data and behaviors and can hide what it wants from other objects.
• **Inheritance**—A class can inherit from another class and take advantage of the attributes and methods defined by the superclass.
• **Polymorphism**—Polymorphism means that similar objects can respond to the same message in different ways. For example, you might have a system with many shapes. However, a circle, a square, and a star are each drawn differently. Using polymorphism, you can send each of these shapes the same message (for example, Draw), and each shape is responsible for drawing itself.
• **Composition**—Composition means that an object is built from other objects.

Chapter 2

Three important things you can do to develop a good sense of the OO thought process are covered in this chapter:

• Knowing the difference between the interface and implementation

• Thinking more abstractly

• Giving the user the minimal interface possible

Knowing your end users is always the most important issue when doing any kind of design.

When designing a class, the general rule is to always provide the user with as little knowledge of the inner workings of the class as possible. To accomplish this, follow these simple rules:

• Give the users only what they absolutely need. In effect, this means the class has as few interfaces as possible. When you start designing a class, start with a minimal interface. The design of a class is iterative, so you will soon discover that the minimal set of interfaces might not suffice. This is fine.

• It is better to have to add interfaces because users really need it than to give the users more interfaces than they need. At times it is highly problematic for the user to have access to certain interfaces. For example, you don't want an interface that provides salary information to all users—only the ones who need to know.

• For the moment, let's use a hardware example to illustrate our software example. Imagine handing a user a PC box without a monitor or a keyboard. Obviously, the PC would be of little use. You have just provided the user with the minimal set of interfaces to the PC. However, this minimal set is insufficient, and it immediately becomes necessary to add interfaces.

• Public interfaces define what the users can access. If you initially hide the entire class from the user by making the interfaces private, when programmers start using the class, you will be forced to make certain methods public—these methods thus become the public interface.

• It is vital to design classes from a user's perspective and not from an information systems viewpoint. Too often designers of classes (not to mention any other kind of software) design the class to make it fit into a specific technological model. Even if the designer takes a user's perspective, it is still probably a technician user's perspective, and the class is designed with an eye on getting it to work from a technology standpoint and not from ease of use for the user.

• Make sure when you are designing a class that you go over the requirements and the design with the people who will actually use it—not just developers (this includes all levels of testing). The class will most likely evolve and need to be updated when a prototype of the system is built.

Initially, you think about how the object is used and not how it is built

---

Doing things in an OO way is more of an art than a science

Chapter 3

In Chapter 3, “More Object-Oriented Concepts,” we discuss the object life cycle: it is born, it lives, and it dies. While it is alive, it might transition through many states. For example, a Database Reader object is in one state if the database is open and another state if the database is closed. How this is represented depends on the design of the class.

The general rule is that you should always provide a constructor, even if you do not plan to do anything inside it. You can provide a constructor with nothing in it and then add to it later. Although there is technically nothing wrong with using the default constructor provided by the compiler, for documentation and maintenance purposes, it is always nice to know exactly what your code looks like.

It is extremely rare for a class to be written perfectly the first time. In most, if not all, situations, things *will* go wrong. Any developer who does not plan for problems is inviting disaster.

A Shared Method

A constructor is a good example of a method that is shared by all instances of a class.

Methods represent the behaviors of an object; the state of the object is represented by attributes. There are three types of attributes:

- Local attributes
- Object attributes
- Class attributes

Deep Versus Shallow Copies

A *deep copy* occurs when all the references are followed, and new copies are created for all referenced objects. Many levels might be involved in a deep copy. For objects with references to many objects, which in turn might have references to even more objects, the copy itself can create significant overhead. A shallow copy would simply copy the reference and not follow the levels. Gilbert and McCarty have a good discussion about what shallow and deep hierarchies are in *Object-Oriented Design in Java* in a section called “Prefer a Tree to a Forest.

Chapter 4

you do not want object A to have the capability to inspect or change the attributes of object B without object B having control. There are several reasons for this; the most important reasons boil down to data integrity and efficient debugging.

Sometimes accessors are referred to as getters and setters, and sometimes they’re simply called get() and set().

In general, the setters are used to ensure a level of data integrity.

Chapter 5

Extending the Interface

Even if the public interface of a class is insufficient for a certain application, object technology easily allows the capability to extend and adapt this interface. In short, if properly designed, a new class can utilize an existing class and create a new class with an extended interface.

A class is most useful if the implementation can change without affecting the users. Basically, a change to the implementation should not necessitate a change in the user's application code. Again, the best way to enable change of behaviors is via the use of interfaces and composition.

---

a constructor should put an object into an initial, safe state

---

Memory Leaks

When an object fails to properly release the memory that it acquired during an object's life cycle, the memory is lost to the entire operating system as long as the application that created the object is executing. For example, suppose multiple objects of the same class are created and then destroyed, perhaps in some sort of loop. If these objects fail to release their memory when they go out of scope, this memory leak slowly depletes the available pool of system memory. At some point, it is possible that enough memory will be consumed that the system will have no available memory left to allocate. This means that any application executing in the system would be unable to acquire any memory. This could put the application in an unsafe state and even lock up the system.

The general rule is that the application should never crash. When an error is encountered, the system should either fix itself and continue, or at minimum, exit gracefully without losing any data that's important to the user.

We can safely say that almost no class lives in isolation. In most cases, there is little reason to build a class if it is not going to interact with other classes, unless the class will be used only once.

The process of designing classes forces you to organize your code into many (ideally) manageable pieces. Separate pieces of code tend to be more maintainable than larger pieces of code (at least that's the idea). One of the best ways to promote maintainability is to reduce interdependent code—that is, changes in one class have no impact or minimal impact on other classes.

To send an object over a wire (for example, to a file, over a network), the system must deconstruct the object (flatten it out), send it over the wire, and then reconstruct it on the other end of the wire. This process is called *serializing* an object. The act of sending the object across a wire is called *marshaling* an object. A serialized object, in theory, can be written to a flat file and retrieved later, in the same state in which it was written. The major issue here is that the serialization and deserialization must use the same specifications. It is sort of like an encryption algorithm. If one object encrypts a string, the object that wants to decrypt it must use the same encryption algorithm.

Chapter 6

This chapter focuses on designing good systems. A *system* can be defined as classes that interact with each other.

One fallacy is that there is one true (best) design methodology. This is certainly not the case. There is no right or wrong way to create a design. Many design methodologies are available today, and they all have their proponents. However, the primary issue is not which design method to use, but whether to use a method at all. This can be expanded beyond design to encompass the entire software development process. Some organizations do not follow a standard software development process, or they have one and don't adhere to it. The most important factor in creating a good design is to find a process that you and your organization feel comfortable with, stick to it, and keep refining it. It makes no sense to implement a design process that no one will follow.

Generally, a solid OO design process includes the following steps:

1. Doing the proper analysis
2. Developing a statement of work that describes the system
3. Gathering the requirements from this statement of work
4. Developing a prototype for the user interface
5. Identifying the classes
6. Determining the responsibilities of each class
7. Determining how the various classes interact with each other
8. Creating a high-level model that describes the system to be built

---

Simply put, the reasons to identify requirements early and keep design changes to a minimum are as follows:

- The cost of a requirement/design change in the design phase is relatively small.
- The cost of a design change in the implementation phase is significantly higher.
- The cost of a design change after the deployment phase is astronomical when compared to the first item

---

The *statement of work* (SOW) is a document that describes the system. Although determining the requirements is the ultimate goal of the analysis phase, at this point the requirements are not yet in a final format. The SOW should give anyone who reads it a complete, high level understanding of the system. Regardless of how it is written, the SOW must represent the complete system and be clear about how the system will look and feel.

Chapter 7

The good news is that the discussions about whether to use inheritance or composition are a natural progression toward some seasoned middle ground. As in all philosophical debates, there are passionate arguments on both sides. Fortunately, as is normally the case, these heated discussions have led to a more sensible understanding of how to utilize the technologies.

The bottom line is that inheritance and composition are both important techniques in building OO systems. Designers and developers need to take the time to understand the strengths and weaknesses of both and to use each in the proper contexts.

Is-a

One of the primary rules of OO design is that public inheritance is represented by an is-a relationship. In the case of interfaces you might add “behaves like” (implements). The data (attributes) that are inherited are the “is,” the interfaces describing encapsulated behaviors are “acts like,” and composition is “has a.” The lines get pretty blurry, however.

This provides us with some significant benefits. First, when we wrote the GoldenRetriever class, we did not have to reinvent part of the wheel by rewriting the bark and pant methods. Not only does this save some design and coding time, but it saves testing and maintenance time as well. The bark and pant methods are written only once and, assuming that they were properly tested when the Dog class was written, they do not need to be heavily tested again; but it does need to be retested because there are new interfaces, and so on.

Consider the object model of the Dog class hierarchy. We started with a single class, called Dog, and we factored out some of the commonality between various breeds of dogs. This concept, sometimes called *generalization-specialization*, is yet another important consideration when using inheritance. The idea is that as you make your way down the inheritance tree, things get more specific. The most general case is at the top of the tree. In our Dog inheritance tree, the class Dog is at the top and is the most general category. The various breeds— the GoldenRetriever, LhasaApso, and Basenji classes—are the most specific. The idea of inheritance is to go from the general to the specific by factoring out commonality.

In theory, factoring out as much commonality as possible is great. However, as in all design issues, sometimes it really is too much of a good thing. Although factoring out as much commonality as possible might represent real life as closely as possible, it might not represent your model as closely as possible. The more you factor out, the more complex your system gets. So you have a conundrum: Do you want to live with a more accurate model or a system with less complexity? You must make this choice based on your situation, for there are no hard guidelines to make the decision

---

Making Design Decisions with the Future in Mind

You might at this point say, “Never say never.” Although you might not breed yodeling dogs now, sometime in the future you might want to do so. If you do not design for the possibility of yodeling dogs now, it will be much more expensive to change the system later to include them. This is yet another of the many design decisions that you have to make. You could possibly override the bark() method to make it yodel; however, this is not intuitive, and some people will expect a method called bark() to actually bark

---

Stephen Gilbert and Bill McCarty define encapsulation as “the process of packaging your program, dividing each of its classes into two distinct parts: the interface and the implementation.” This is the message that has been presented over and over in this book.

How Inheritance Weakens Encapsulation ??

The premise of polymorphism is that you can send messages to various objects, and they will respond according to their object's type.

Many well-respected OO designers have stated that composition should be used whenever possible, and inheritance should be used only when necessary.

Chapter 8

The online dictionary, Merriam-Webster (https://www.merriam-webster.com), defines a contract as a "binding agreement between two or more persons or parties, especially: one legally enforceable.

One way a contract is implemented is via an abstract class. An *abstract class* is a class that contains one or more methods that do not have any implementation provided. Suppose that you have an abstract class called Shape. It is abstract because you cannot instantiate it. If you ask someone to draw a shape, the first thing the person will most likely ask you is, "What kind of shape?" Thus, the concept of a shape is abstract. However, if someone asks you to draw a circle, this does not pose quite the same problem, because a circle is a concrete concept. You know what a circle looks like. You also know how to draw other shapes, such as rectangles.

One way a contract is implemented is via an abstract class. An *abstract class* is a class that contains one or more methods that do not have any implementation provided. Suppose that you have an abstract class called Shape. It is abstract because you cannot instantiate it. If you ask someone to draw a shape, the first thing the person will most likely ask you is, "What kind of shape?" Thus, the concept of a shape is abstract. However, if someone asks you to draw a circle, this does not pose quite the same problem, because a circle is a concrete concept.

You know what a circle looks like. You also know how to draw other shapes, such as rectangles.

Although the concept of abstract classes revolves around abstract methods, nothing is stopping Shape from providing some implementation. Remember that the definition for an abstract class is that it contains *one or more* abstract methods—this implies that an abstract class can also provide concrete methods. For example, although Circle and Rectangle implement the draw() method differently, they share the same mechanism for setting the color of the shape. So, the Shape class can have a color attribute and a method to set the color. This setColor() method is a concrete implementation and would be inherited by both Circle and Rectangle. The only methods that a subclass must implement are the ones that the superclass declares as abstract. These abstract methods are the contract.

Be aware that in the cases of Shape, Circle, and Rectangle, we are dealing with a strict inheritance relationship, as opposed to an interface, which we discuss in the next section. Circle *is-a* Shape, and Rectangle *is-a* Shape.

Interface Terminology

This is another one of those times when software terminology gets confusing—very confusing. Be aware that you can use the term interface in several ways, so be sure to use each in the proper context. First, the graphical user interface (GUI) is widely used when referring to the visual interface that a user interacts with—often on a monitor. Second, the interface to a class is basically the signatures of its methods. Third, in Objective-C and Swift, you break up the code into physically separate modules called the interface and implementation.

Fourth, an interface and a protocol are basically a contract between a parent class and a child class. Can you think of any others?

The obvious question is this: If an abstract class can provide the same functionality as an interface, why do Java and .NET bother to provide this construct called an interface? And why does Objective-C and Swift provide the protocol?

For one thing, C++ supports multiple inheritance, whereas Java, Objective-C, Swift, and .NET do not. Although Java, Objective-C, Swift, and .NET classes can inherit from only one parent class, they can implement many interfaces. Using more than one abstract class constitutes multiple inheritance; thus, Java and .NET cannot go this route. In short, when using an interface, you do not have to concern yourself with a formal inheritance structure—you can theoretically add an interface to any class if the design makes sense. However, an abstract class requires you to inherit from that abstract class and, by extension, all of its potential parents.

Because of these considerations, interfaces are often thought to be a workaround for the lack of multiple inheritance. This is not technically true. Interfaces are a separate design technique, and although they can be used to design applications that could be done with multiple inheritance, they do not replace or circumvent multiple inheritance.

---

notice that Nameable is not declared as a class but as an interface. Because of this, both methods, getName() and setName(), are considered abstract and no implementation is provided. An interface, unlike an abstract class, can provide *no* implementation at all. As a result, any class that implements an interface must provide the implementation for all methods

---

if both abstract classes and interfaces provide abstract methods, what is the real difference between the two? As we saw before, an abstract class can provides both abstract and concrete methods, whereas an interface provides only abstract methods Why is there such a difference?
Assume that we want to design a class that represents a dog, with the intent of adding more mammals later. The logical move would be to create an abstract class called Mammal:

```
public abstract class Mammal {
public void generateHeat() {System.out.println("Generate heat");}
public abstract void makeNoise();
}
```

This class has a concrete method called generateHeat()and an abstract method called makeNoise(). The method generateHeat()is concrete because all mammals generate heat. The method makeNoise()is abstract because each mammal will make noise differently.Let's also create a class called Head that we will use in a composition relationship:

```
public class Head {
String size;
public String getSize() {
return size;
}
public void setSize(String aSize) { size = aSize; }
}
```

Head has two methods: getSize() and setSize(). Although composition might not shed much light on the difference between abstract classes and interfaces, using composition in this example does illustrate how composition relates to abstract classes and interfaces in the overall design of an object-oriented system. I feel that this is important because the example is more complete. Remember that there are two ways to build object relationships: the *is-a* relationship, represented by inheritance, and the *has-a* relationship, represented by composition. The question is: Where does the interface fit in?

To answer this question and tie everything together, let's create a class called Dog that is a subclass of Mammal, implements Nameable, and has a Head object

In a nutshell, Java and .NET build objects in three ways: inheritance, interfaces, and composition. Note the dashed line in Figure 8.5 that represents the interface. This example illustrates when you should use each of these constructs. When do you choose an abstract class? When do you choose an interface? When do you choose composition? Let's explore further. You should be familiar with the following concepts:

- Dog is a Mammal, so the relationship is inheritance.
- Dog implements Nameable, so the relationship is an interface.
- Dog has a Head, so the relationship is composition.

The key here is that classes in a strict inheritance relationship must be related. For example, in this design, the Dog class is directly related to the Mammal class. A dog is a mammal. Dogs and lizards are not related at the mammal level because you can't say that a lizard is a mammal. However, interfaces can be used for classes that are not related. You can name a dog just as well as you can name a lizard. This is the key difference between using an abstract class and using an interface.

Chapter 9

Another major advantage in using composition is that systems and subsystems can be built independently, and perhaps more importantly, tested and maintained independently.

There is no question that software systems are quite complex. To build quality software, you must follow one overriding rule to be successful: Keep things as simple as possible. For large software systems to work properly and be easily maintained, they must be broken into smaller, more manageable parts.

• **"Stable complex systems usually take the form of a hierarchy, where each system is built from simpler subsystems, and each subsystem is built from simpler subsystems still.**—You might already be familiar with this principle because it forms the basis for functional decomposition, the method behind procedural software development. In object-oriented design, you apply the same principles to composition—building complex objects from simpler pieces.
• **"Stable, complex systems are nearly decomposable."**—This means you can identify the parts that make up the system and can tell the difference between interactions between the parts and inside the parts. Stable systems have fewer links between their parts than they have inside their parts. Thus, a modular stereo system, with simple links between the speakers, turntable, and amplifier, is inherently more stable than an integrated system, which isn't easily decomposable.
• **"Stable complex systems are almost always composed of only a few different kinds of subsystems, arranged in different combinations."**—Those subsystems, in turn, are generally composed of only a few different kinds of parts.
• **"Stable systems that work have almost always evolved from simple systems that worked."**—Rather than build a new system from scratch—reinventing the wheel— the new system builds on the proven designs that went before it.

Composition is another area in OO technologies where there is a question of which came first, the chicken or the egg. Some texts say that composition is a form of association, and some say that an association is a form of composition. In any event, in this book, we consider inheritance and composition the two primary ways to build classes. Thus, in this book, association is considered a form of composition.

Aggregation Versus Association

An aggregation is a complex object composed of other objects. An association is used when one object wants another object to perform a service for it.

No One Right Answer

As usual, there isn't a single, absolutely correct answer when it comes to making a design decision. Design is not an exact science. Although we can make general rules to live by, these rules are not hard
and fast.

Chapter 10

The concept of design patterns did not necessarily start with the need for reusable software. In fact, the seminal work on design patterns is about constructing buildings and cities. As Christopher Alexander noted in *A Pattern Language: Towns, Buildings, Construction*, “Each describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice.

The GoF describe a pattern as having four essential elements:

• The *pattern name* is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves. It makes it easier to think about designs and to communicate them and their trade-off to others. Finding good names has been one of the hardest parts of developing our catalog.?

• The *problem* describes when to apply the pattern. It explains the problem and its content. It might describe specific design problems, such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.
• The *solution* describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many situations. Instead, the pattern provides an abstract description of a design problem, and how a general arrangement of elements (classes and objects in our case) solves it.
• The *consequences* are the results and trade-offs of applying the pattern. Although consequences are often unvoiced, when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of the applying pattern. The consequences for software often concern space and time trade-offs. They might address language and implementation issues as well. Because reuse is often a factor in object oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing the consequences explicitly helps you understand and evaluate them.

• *Creational patterns* create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
• *Structural patterns* help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.
• *Behavioral patterns* help you define the communication between objects in your system and how the flow is controlled in a complex program.

The creational patterns consist of the following categories:

- Abstract factory
- Builder
- Factory method
- Prototype
- Singleton
