Overview

In this chapter

This chapter introduces you to the object-oriented paradigm by comparing and contrasting it with something familiar: standard structured programming.

The object-oriented paradigm grew out of a need to meet the challenges of past practices using standard structured programming. By being clear about these challenges, we can better see the advantages of object-oriented programming, as well as gain a better understanding of this mechanism.

This chapter will not make you an expert on object-oriented methods. It will not even introduce you to all of the basic object-oriented concepts. It will, however, prepare you for the rest of this book, which explains the proper use of object-oriented design methods as practiced by the experts.

In this chapter

• I discuss a common method of analysis, called functional decomposition.
• I address the problem of requirements and the need to deal with change (the scourge of programming!).
• I describe the object-oriented paradigm and show its use in action.
• I point out special object methods.
• I provide a table of important object terminology used in this chapter on page 20.
Before the Object-Oriented Paradigm: Functional Decomposition

Functional decomposition is a natural way to deal with complexity

Let’s start by examining a common approach to software development. If I were to give you the task of writing code to access a description of shapes that were stored in a database and then display them, it would be natural to think in terms of the steps required. For example, you might think that you would solve the problem by doing the following:

1. Locate the list of shapes in the database.
2. Open up the list of shapes.
3. Sort the list according to some rules.
4. Display the individual shapes on the monitor.

You could take any one of these steps and break it down further into the steps required to implement it. For example, you could break down Step 4. For each shape in the list, do the following:

4a. Identify the type of shape.
4b. Get the location of the shape.
4c. Call the appropriate function that will display the shape, giving it the shape’s location.

This is called functional decomposition because the analyst breaks down (decomposes) the problem into the functional steps that compose it. You and I do this because it is easier to deal with smaller pieces than it is to deal with the problem in its entirety. It is the same approach I might use to write a recipe for making lasagna, or instructions to assemble a bicycle. We use this approach so often and so naturally that we seldom question it or ask whether other alternatives exist.

The challenge with this approach: With great power comes great responsibility

One problem with functional decomposition is that it usually leads to having one “main” program that is responsible for controlling its subprograms. It is a natural outcome of decomposing functions into smaller functions. However, it saddles the main program with too much responsibility: ensuring everything is working correctly, coordinating and sequencing functions. Often, therefore, it results in very complicated code. How much easier it would be to make some of those subfunctions responsible for their own behavior, to be able to tell the function to go do something and trust that it will know how to do it. Successful generals in the field and parents in the home have learned this lesson. Programmers are learning it as well. It is called delegation.

The challenge with this approach: Dealing with change

Another problem with functional decomposition is that it does not help us prepare the code for possible changes in the future, for a graceful evolution. When change is required, it is often because I want to add a new variation to an existing theme. For example, I might have to deal with new shapes or new ways to display shapes. If I have put all the logic that implements the steps into one large function or module, virtually any change to the steps will require changes to that function or module. And change creates opportunities for mistakes and unintended consequences. Or, as I like to say,

Many bugs originate with changes to code.

"If I had eight hours to chop down a tree, I'd spend six sharpening my axe.”

-- Abraham Lincoln
Verify this assertion for yourself. Think of a time when you wanted to make a change to your code, but were afraid to put it in because you knew that modifying the code in one place could break it somewhere else. Why might this happen? Must the code pay attention to all of its functions and how they might be used? How might the functions interact with one another? Were there too many details for the function to pay attention to, such as the logic it was trying to implement, the things with which it was interacting, the data it was using? Just as with people, programs trying to focus on too many things simultaneously beg for errors when anything changes. Programming is a complex, abstract, dynamic activity.

And no matter how hard you try, no matter how well you do your analysis, you can never get all of the requirements from the user. Too much is unknown about the future. Things change. They always do…

And nothing you can do will stop change. But you do not have to be overcome by it.

The Problem of Requirements

Requirements always change

Ask software developers what they know to be true about the requirements they get from users. They often answer the following:

- Requirements are incomplete
- Requirements are usually wrong
- Requirements (and users) are misleading
- Requirements do not tell the whole story

One thing you will never hear is, “Not only were our requirements complete, clear, and understandable, but they laid out all of the functionality we were going to need for the next five years!”

In my 30 years of experience writing software, the main thing I have learned about requirements is that requirements always change.

I have also learned that most developers think this is a bad thing. But few of them write their code to handle changing requirements well.

Requirements change for a very simple set of reasons:

- The users’ view of their needs changes as a result of their discussions with developers and from seeing new possibilities for the software
- The developers’ view of the users’ problem domain changes as they develop software to automate it and thus become more familiar with it

About the authors --

Alan Shalloway

Alan Shalloway is the CEO of Net Objectives. Since 1981, he has been both an OO consultant and developer of software in several industries. Alan is a frequent speaker at prestigious conferences around the world, including: SD Expo, Java One, OOP, OOPSLA. He is the primary author of Design Patterns Explained: A New Perspective on Object-Oriented Design and is currently co-authoring three other books in the software development area. He is a certified Scrum Master and has a Masters in Computer Science from M.I.T.

James Trott

James Trott is a knowledge management consultant, collaboration specialist, and knowledge engineer. He has Master of Science in Applied Mathematics, an MBA, and a Master of Arts in Intercultural Studies. He has twenty years experience as a pattern-based analyst, working in the petroleum, aerospace, software, and financial industries. He divides his consulting efforts between corporate work and humanitarian work with Millennium Relief and Development Services.
I May Not Know What Will Change, But I Can Guess Where

Early in my career, I had a mentor who used to say, “There is always time to program it right the second time, so you might as well do it right the first time!” I have often thought about that advice. I used to think it meant trying to anticipate every change that might be made and building my code accordingly. This was overwhelming and usually disappointing because I rarely could predict every possible change that might come my way.

Finally I realized that although I could not predict what changes might occur, I could usually anticipate where the changes might occur. One of the great benefits of object orientation is that I can contain those areas of change and thus insulate my code from the effects of change more easily.

The environment in which the software is being developed changes. (Who anticipated, five years ago, Web development as it is today?)

This does not mean you and I can give up on gathering good requirements. It does mean that we must write our code to accommodate change. It also means we should stop beating ourselves up (or our customers, for that matter) for things that will naturally occur.

Change Happens! Deal With It

In all but the simplest cases, requirements will always change, no matter how well we do the initial analysis!

Rather than complaining about changing requirements, we should change the development process so that we can address change more effectively.

You can design your code so that the impact of changing requirements is much less dramatic. Your code may evolve or new code can be bolted on with little impact.

Dealing with Changes:
Using Functional Decomposition

Using modularity to isolate variation

Look a little closer at the problem of displaying shapes. How can I write the code so that it is easier to handle shifting requirements? Instead of writing one large function, I could make it more modular.

For example, in Step 4c on page 2, where I “call the appropriate function that will display the shape, giving it the shape’s location,” I could write a module like that shown in Example 1-1.

Example 1-1 Using Modularity to Contain Variation

function: display shape
input: type of shape, description of shape
action:
      switch (type of shape)
        case square: put display function for square here
        case circle: put display function for circle here
Seminars We Can Give

Agile Planning Game – The Planning Game was created by Kent Beck and is well described in his excellent book: Extreme Programming Explained. Unfortunately, the Planning Game as described is not complete enough - even for pure, XP teams. This seminar describes the other factors which must often typically be handled.

Comparing RUP, XP, and Scrum: Mixing a Process Cocktail for Your Team – This seminar discusses how combining the best of some popular processes can provide a successful software development environment for your project.

Design Patterns in an Agile (even XP) Environment – The Object Pool – This seminar presents a project done by following the guidelines of Design Patterns, Agile Development, and Refactoring to show how these ideas can inform each other.

Emergent Design: Design Patterns and Refactoring for Agile Development - This seminar illustrates why design patterns and refactoring, which can seem opposed to each other, are actually two sides of the same coin.

Transitioning to Agile – More and more companies are beginning to see the need for Agile Development. In this seminar, we discuss what problems agility will present and how to deal with these.

Use Case Driven Agile Development - Capturing functional requirements with Use Cases is a software development best practice. Agile development processes are proving themselves to be effective. They work very well together by combining the iterative nature of agility with incremental development of use cases, as this seminar explains.

Unit Testing For Emergent Design – This seminar illustrates why design patterns and refactoring are actually two sides of the same coin.

Check www.netobjectives.com/events/pr_main.htm#FreeSeminars for Public Seminars in Western Washington State, Oregon, Arizona, Midwest, and Northern California

If your user group or company is interested in Net Objectives making a free technical presentation for them, or if you would like to be notified of upcoming Net Objectives events, please visit our website, or contact us by the email address or phone number below.

www.netobjectives.com • mike.shalloway@netobjectives.com • 404-593-8375
**Problems with modularity in a functional decomposition approach**

Then, when I receive a requirement to be able to display a new type of shape—a triangle, for instance—I only need to change this module (hopefully!).

This approach presents some problems, however. For example, I said that the inputs to the module were the type of shape and a description of the shape. Depending upon how I am storing shapes, it may or may not be possible to have a consistent description of shapes that will work well for all shapes. What if the description of the shape is sometimes stored as an array of points and sometimes is stored another way? Would that still work?

Modularity definitely helps to make the code more understandable, and understandability takes the code easier to maintain. But modularity does not always help code deal with all the variations it might encounter.

**Weak cohesion, tight coupling**

With the approach that I have used so far, I find that I have two significant problems, which go by the terms weak cohesion and tight coupling. In his book *Code Complete* (Microsoft Press, 1993), Steve McConnell gives an excellent description of both cohesion and coupling. He says:

> **Cohesion refers to how “closely the operations in a routine are related.”**

I have heard other people refer to cohesion as *clarity* because the more that operations are related in a routine (or a class), the easier it is to understand things. Weakly cohesive classes are those that do many, unrelated tasks. The code often appears to be a confused mass. Taken to an extreme, these classes become entangled with most everything in a system. I have heard some people call these *god objects*, because they do everything (or perhaps it’s because only God can understand them).

**Coupling refers to “the strength of a connection between two routines. Coupling is a complement to cohesion. Cohesion describes how strongly the internal contents of a routine are related to each other. Coupling describes how strongly a routine is related to other routines. The goal is to create routines with internal integrity (strong cohesion) and small, direct, visible, and flexible relations to other routines (loose coupling).”**

Changing a function, or even data used by a function, can wreak havoc on other functions

Most programmers have had the experience of making a change to a function or piece of data in one area of the code that then has an unexpected impact on other pieces of code. This type of bug is called an *unwanted side effect*. That is because although we get the impact we want (the change), we also get other impacts we don’t want—bugs! What is worse, these bugs are often difficult to find because we usually don’t notice the relationship that caused the side effects in the first place. (If we had, we wouldn’t have changed it the way we did.)

In fact, bugs of this type lead me to a rather startling observation: *We really do not spend much time fixing bugs.*

I think fixing bugs takes a short period of time in the maintenance and debugging process. The

---

1 McConnell, S. *Code Complete: A Practical Handbook of Software Construction*, Redmond: Microsoft Press, 1993, p. 81. (Note: McConnell did not invent these terms, Yourdon and Constantine did. We just happen to like his definitions best.)

2 Ibid, p. 87.
overwhelming amount of time involved in maintenance and debugging is spent on trying to discover how the code works and on finding bugs and taking the time to avoid unwanted side effects. The actual fix is relatively short!

Because unwanted side effects are often the hardest bugs to find, having a function that touches many different pieces of data makes it more likely that a change in requirements will result in a problem.

### The Devil Is in the Side Effects

- A focus on functions is likely to cause side effects that are difficult to find.
- Most of the time spent in maintenance and debugging is not spent on fixing bugs, but in finding them and seeing how to avoid unwanted side effects from the fix.

### Functional decomposition focuses on the wrong thing

With functional decomposition, changing requirements cause my software development and maintenance efforts to thrash. I am focused primarily on the functions. Changes to one set of functions or data impact other sets of functions and other sets of data, which in turn impact other functions that must be changed. Like a snowball that picks up snow as it rolls downhill, a focus on functions leads to a cascade of changes from which it is difficult to escape.

### Dealing with Changing Requirements

#### How do people do things?

To figure out a way around the problem of changing requirements and to see whether there is an alternative to functional decomposition, let’s look at how people do things. Suppose, for example, that you are an instructor at a conference. People in your class have another class to attend following yours, but don’t know where it is located. One of your responsibilities is to make sure everyone knows how to get to the next class.

If you were to follow a structured programming approach, you might do the following:

1. Get a list of people in the class.
2. For each person on this list, do the following:
   a. Find the next class he or she is taking
   b. Find the location of that class
   c. Find the way to get from your classroom to the person’s next class
   d. Tell the person how to get to his or her next class

To do this would require the following procedures:

1. A way of getting the list of people in the class
2. A way of getting the schedule for each person in the class
3. A program that gives someone directions from your classroom to any other classroom
4. A control program that works for each person in the class and does the required steps for each person

#### Doubtful you’d follow this approach

I doubt that you would actually follow this approach. Instead, you would probably post directions to go from this classroom to the other classrooms and then tell everyone in the class, “I have posted the locations of the classes following this in the back of the room, as well as the locations of the other classrooms. Please use them to go to your next classroom.” You would expect that everyone would know what his or her next class was, and that everyone could find the right classroom from the list and could then follow the directions to get to the correct classrooms.
More Seminars We Can Give

Agility and Ceremony: How Can They Co-Exist? – This seminar provides a short overview of Agility and the challenges one faces in a High Ceremony environment.

C# for Java and C++ Developers – This seminar will give a complete overview of the major language features of C#, and will also examine some best-practice issues.

Design Patterns and Extreme Programming – Patterns are often thought of as an up-front design approach. That is not accurate. This seminar illustrates how the principles and strategies learned from patterns can actually facilitate agile development. This talk walks through a past project of the presenter.

Effective Coding Practices – This seminar introduces three straightforward coding practices that support design patterns, refactoring and test-driven-development: Encapsulating Construction, Coding by Intention, and Considering Testability before Coding.

Effective C# – This seminar is intended for programmers who are not object-oriented experts and who want to learn how to use C# effectively.

Introduction to Use Cases – In this seminar we present different facets of the lowly Use Case; what they are, why they’re important, how to write one, and how to support agile development with them.

The Need for Agility – This seminar is designed to demonstrate how to balance the necessity for up-front analysis and design with the need to get feedback about how the project is going.

Net Objectives Panel Discussion on Agile Software Development - In this special presentation, key members of the Net Objectives training staff give their views on agile software development and take questions from the audience.

Pattern Oriented Development: Design Patterns From Analysis To Implementation – This seminar discusses how design patterns can be used to improve the entire software development process - not just the design aspect of it.

Test-First Techniques Using xUnit and Mock Objects – This seminar explains the basics of unit testing, how to use unit tests to drive coding forward (test-first), and how to resolve some of the dependencies that make unit testing difficult.

The XP Planning Game and Iteration Retrospectives – Tools, techniques and tricks that aid iteration planning and team learning will be discussed.

and 3 on key XML topics:
XML Data Binding With Castor
XSLT - Step By Step
The JDOM Alternative For XML Parsing

If you are interested in any of these offerings, if your user group or company is interested in Net Objectives making a free technical presentation to them, or if you would like to be notified of upcoming Net Objectives events, please visit our website, or contact us by the email address or phone number below:

www.netobjectives.com • mike.shalloway@netobjectives.com • 404-593-8375
What is the difference between these approaches?

- In the first one—giving explicit directions to everyone—you have to pay close attention to a lot of details. No one other than you is responsible for anything. You will go crazy!
- In the second case, you give general instructions and then expect that each person will figure out how to do the task individually.

**Shifting responsibility from yourself to individuals…**

The biggest difference is this shift of responsibility. In the first case, you are responsible for everything; in the second case, students are responsible for their own behavior. In both cases, the same things must be implemented, but the organization is very different.

**What is the impact of this?**

To see the effect of this reorganization of responsibilities, let’s consider what happens when some new requirements are specified.

Suppose I am now told to give special instructions to graduate students who are assisting at the conference. Perhaps they need to collect course evaluations and take them to the conference office before they can go to the next class. In the first case, I would have to modify the control program to distinguish the graduate students from the undergraduates, and then give special instructions to the graduate students. It’s possible that I would have to modify this program considerably.

**…can minimize changes**

However, in the second case—where people are responsible for themselves—I would just have to write an additional routine for graduate students to follow. The control program would still just say, “Go to your next class.” Each person would simply follow the instructions appropriate for himself or herself.

**Why the difference?**

This represents a significant shift in responsibility for the control program. In one case, every time a new category of students needs to be added, the control program itself has to be modified; the control program is responsible for telling the new category of student what to do. In the other case, the control program remains unaffected by the new category of student; the students themselves are responsible for figuring out what to do.

**What makes it happen?**

There are three different things going on that make this happen:

- The people are responsible for their own behavior. There is not a central control program responsible for determining their behavior. (Note that to accomplish this, a person must also be aware of what type of student he or she is.)
- The control program can talk to different types of people (graduate students and regular students) as if they were exactly the same.
- The control program does not need to know about any special steps that students might need to take when moving from class to class.

**Different perspectives**

To fully understand the implications of this, it’s important to establish some terminology. In *UML Distilled* (Addison-Wesley, 1999), Martin Fowler describes three different perspectives in the software development process. These are described in Table 1-1.

---

**Table 1-1 Perspectives in the Software Development Process**

<table>
<thead>
<tr>
<th>Perspective</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conceptual</td>
<td>This perspective “represents the concepts in the domain under study…a conceptual model should be drawn with little or no regard for the software that might implement it…” It answers the question, “What am I responsible for?”</td>
</tr>
<tr>
<td>Specification</td>
<td>“Now we are looking at software, but we are looking at the interfaces of the software, not the implementation.” It answers the question, “How am I used?”</td>
</tr>
<tr>
<td>Implementation</td>
<td>At this point we are at the code itself. “This is probably the most often-used perspective, but in many ways the specification perspective is often a better one to take.” It answers the question, “How do I fulfill my responsibilities?”</td>
</tr>
</tbody>
</table>

**How perspectives help**

Look again at the previous example of “Go to your next class.” Notice that you—as the instructor—are communicating with the people at the conceptual level. In other words, you are telling people what you want, not how to do it. However, the way they go to their next class is very specific. They are following specific instructions and in doing so are working at the implementation level.

Communicating at one level (conceptually) while performing at another level (implementation) results in the requestor (the instructor) not having to know exactly what is happening, only having to know in general—conceptually—what is happening. This can be very powerful: The requestor is insulated from changes in implementation details as long as the concept remains the same. Let’s see how to take these notions and write programs that take advantage of them.

**The Object-Oriented Paradigm**

**Using objects shifts responsibility to a more local level**

The object-oriented paradigm is centered on the concept of the object. Everything is focused on objects. I write code organized around objects, not functions.

What is an object? Objects have traditionally been defined as data with methods (the object-oriented term for functions). Unfortunately, this is a very limiting way of looking at objects. I will look at a better definition of objects shortly (and again in Chapter 8, “Expanding Our Horizons”). When I talk about the data of an object, these can be simple things such as numbers and character strings, or they can be other objects.

The advantage of using objects is that I can define things that are responsible for themselves. (See Table 1-2.) Objects inherently know what type they are. The data in an object allows it to know
what state it is in, and the code in the object allows it to function properly (that is, do what it is supposed to do).

<table>
<thead>
<tr>
<th>This Object...</th>
<th>Is Responsible For...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>Knowing which classroom he or she is in</td>
</tr>
<tr>
<td></td>
<td>Knowing which classroom to go to next</td>
</tr>
<tr>
<td></td>
<td>Going from one classroom to the next</td>
</tr>
<tr>
<td>Instructor</td>
<td>Telling people to go to next classroom</td>
</tr>
<tr>
<td>Classroom</td>
<td>Having a location</td>
</tr>
<tr>
<td>Direction giver</td>
<td>Given two classrooms, giving directions from one classroom to the other</td>
</tr>
</tbody>
</table>

In this case, the objects were identified by looking at the entities in the problem domain. I identified the responsibilities (or methods) for each object by looking at what these entities need to do. This is consistent with the technique of finding objects by looking for the nouns in the requirements and finding methods by looking for verbs. As I get into more complex problems, you will see that this technique is quite limiting, and I will show a better way throughout this book. For now, it is a way to get us started.

**How to think about objects**

The best way to think about what an object is, is to think of it as something with responsibilities. A good design rule is that objects should be responsible for themselves and should have those responsibilities clearly defined. This is why I say one of the responsibilities of a student object is knowing how to go from one classroom to the next.

**Or, taking Fowler’s perspective**

I can also look at objects using the framework of Fowler’s perspectives:

- At the **conceptual level**, an object is a set of responsibilities.

- At the **specification level**, an object is a set of methods (behaviors) that can be invoked by other objects or by itself.

- At the **implementation level**, an object is code and data and computational interactions between them.

Unfortunately, object-oriented design is often taught and talked about only at the implementation level—in terms of code and data—rather than at the conceptual or specification level. But there is great power in thinking about objects in these latter ways as well!

**Objects have interfaces for other objects to use**

Because objects have responsibilities and objects are responsible for themselves, there has to be a way to tell objects what to do. Remember that objects have data to tell the object about itself and methods to implement required functionality. Many methods of an object will be identified as callable by other objects. The collection of these methods is called the object’s **public interface**.

For example, in the classroom example, I could write the **Student** object with the method **gotoNextClassroom()**. I would not need to pass any parameters in because each **Student** object would be responsible for itself. That is, it would know:

---

• What it needs to be able to move
• How to get any additional information it needs to perform this task

Organizing objects around the class
Initially, there was only one kind of student—a regular student who goes from class to class. Note that there would be many of these “regular students” in my classroom (my system). I would have an object for each student, enabling me to track the state of each student easily and independently of other students. However, it seems inefficient to require each Student object to have its own set of methods to tell it what it can do and how to do it, especially for tasks that are common to all students.

A more efficient approach would be to have a set of methods associated with all students that each one could use or tailor to his or her own needs. I want to define a “general student” to contain the definitions of these common methods. Then I can have all manner of specialized students, each of whom has to keep track of his or her own private information.

In object-oriented terms, this general student is called a class. A class is a definition of the behavior of an object. It contains a complete description of the following:
• The data elements the object contains
• The methods the object can do
• The way these data elements and methods can be accessed

Because the data elements an object contains can vary, each object of the same type may have different data but will have the same functionality (as defined in the methods).

Objects are instances of classes
To get an object, I tell the program that I want a new object of this type (that is, the class that the object belongs to). This new object is called an instance of the class. Creating instances of a class is called instantiation.

Working with objects in the example
Writing the “Go to the next classroom” example using an object-oriented approach is much simpler. The program would look like this:
1. Start the control program.
2. Instantiate the collection of students in the classroom.
3. Tell the collection to have the students go to their next class.
4. The collection tells each student to go to his or her next class.
5. Each student
   a. Finds where his next class is.
   b. Determines how to get there.
   c. Goes there.
6. Done.

The need for an abstract type
This works fine until I need to add another student type, such as a graduate student.

I have a dilemma. It appears that I must allow any type of student into the collection (either regular student or graduate student). The problem facing me is how do I want the collection to refer to its constituents? Because I am talking about implementing this in code, the collection will actually be an array or something, of some type of object. If the collection were named something like RegularStudent, I would not be able to put graduate students into the collection. If I say that the collection is just a group of objects, how can I be sure that I do not include the wrong type of object (that is, something that doesn’t do “Go to your next class”)?

The solution is straightforward. I need a general type that encompasses more than one specific type. In this case, I want a Student type that includes both RegularStudent and
GraduateStudent. In object-oriented terms, we call Student an abstract class.\(^5\)

**Abstract classes define what a set of classes can do**

Abstract classes define what other, related, classes can do. These “other” classes are classes that represent a particular type of related behavior. Such a class is often called a concrete class because it represents a specific, or nonchanging, implementation of a concept.

In the example, the abstract class is Student. There are two types of Student represented by the concrete classes, RegularStudent and GraduateStudent. RegularStudent is one kind of Student, and GraduateStudent is also a kind of Student.

This type of relationship is called an is-a relationship. An is-a relationship is an example of something we call inheritance. Thus, the RegularStudent class inherits from Student. Other ways to say this would be, the GraduateStudent derives from, specializes, or is a subclass of Student.

Going the other way, the Student class is the base class, generalizes, or is the superclass of GraduateStudent and of RegularStudent.

**Abstract classes act as placeholders for other classes**

Abstract classes act as placeholders for other classes. I use them to define the methods their derived classes must implement. Abstract classes can also contain common methods that can be used by all derivations.\(^6\) Whether a derived class uses the default behavior or replaces it with its own variation is up to the derivation. (This is consistent with the mandate that objects be responsible for themselves.)

This means that I can have the controller contain Students. The reference type used will be Student. The compiler can check that anything referred to by this Student reference is, in fact, a kind of Student. This gives the best of both worlds:

- The collection only needs to deal with Students (thereby allowing the instructor object just to deal with students).
- But I still get type checking (only Students that can “Go to their next classroom” are included).
- And each kind of Student is left to implement its functionality in its own way.

**Abstract Classes Are More Than Classes That Do Not Get Instantiated**

Abstract classes are often described as classes that do not get instantiated. This definition is accurate—at the implementation level. But that is too limited. It is more helpful to define abstract classes at the conceptual level. At the conceptual level, abstract classes are placeholders for other classes—classes that implement specifics of the concept the abstract class represents.

That is, they give us a way to assign a name to a set of related classes. This enables us to treat this set of related classes as one concept.

In the object-oriented paradigm, you must constantly think about your problem from all three levels of perspective: conceptual, specification, and implementation.

---

5. Interfaces in several languages also can do this. When I refer to abstract classes in this chapter, you can pretend I’ve written abstract class or interface.

6. Here is one difference between abstract classes and interfaces. Interfaces just define what a set of classes can do—they cannot implement default behavior.
Visibility
Because objects are responsible for themselves, there are many things they do not need to expose to other objects. Earlier I mentioned the concept of the public interface—those methods that are accessible by other objects. In object-oriented systems, the main types of accessibility are as follows:7

- **Public**—Anything can see it.
- **Protected**—Only objects of this class and derived classes can see it.
- **Private**—Only objects from this class can see it.

To encapsulate a private variable.

Encapsulation
This leads to the concept of encapsulation. Encapsulation has often been described simply as hiding data. Objects generally do not expose their internal data members to the outside world. (That is, their visibility is protected or private.)

But encapsulation refers to more than hiding data. In general, encapsulation means any kind of hiding.

In the example, the instructor did not know which were the regular students and which were the graduate students. The type of student is hidden from the instructor. (I am encapsulating the “type” of the student.) In an object-oriented language, the abstract class `Student` hides the types of classes derived from it. As you will see later in the book, this is a very important concept.

Polymorphism
Another term to learn is polymorphism.

In object-oriented languages, we often refer to objects with one type of reference that is an abstract class type. However, what we are actually referring to are specific instances of classes derived from their abstract classes.

Thus, when I tell the objects to do something conceptually through the abstract reference, I get different behavior, depending upon the specific type of derived object I have. Polymorphism derives from *poly* (meaning many) and *morph* (meaning form). Thus, it means “many forms.” This is an appropriate name because I have many different forms of behavior for the same call.

In the example, the instructor tells the students to “Go to your next classroom.” However, depending upon the type of student, he or she will exhibit different behavior (hence polymorphism).

---

7 Different languages often have other types of accessibility. However, they are essentially variations of these three.

---

Technical Book Recommendation:

**Lean Software Development by Mary and Tom Poppendieck**

(ISBN: 0321150783) I believe Lean Software Development will turn out to be the main agile approach in the future. It is already understood outside of the software community and greatly appeals to management. Shows them what they should do without making them wrong.
## Review of Object-Oriented Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract class</td>
<td>Abstract classes define what a set of related classes can do.</td>
</tr>
<tr>
<td>Class</td>
<td>Defines the types of objects I have based on the responsibilities these objects have. Responsibilities can be divided into behavior and/or state. These can be implemented using methods and/or data, respectively.</td>
</tr>
<tr>
<td>Concrete Class</td>
<td>A class that implements a particular type of behavior for an abstract class. Concrete classes are specific, nonchanging implementations of a concept.</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>Typically defined as data hiding, but better thought of as any kind of hiding (type, implementation, design, and so on).</td>
</tr>
<tr>
<td>Inheritance</td>
<td>A class inherits from another class when it receives some or all of the qualities of that class. The starting class is called the base, super, parent, or generalized class, whereas the inheriting class is called the derived, sub, child, or specialized class.</td>
</tr>
<tr>
<td>Instance</td>
<td>A particular example of a class. (It is always an object.) A particular instance or entity of a class. Each object has its own state. This enables me to have several objects of the same type (class).*</td>
</tr>
<tr>
<td>Instantiation</td>
<td>The process of creating an instance of a class.</td>
</tr>
<tr>
<td>Interface</td>
<td>An interface is like a class, but only provides a specification—and not an implementation—for its members. It is similar to an abstract class consisting only of abstract members. When programming, you use interfaces when you need several classes to share some characteristics that are not present in a common base class and want to be sure that each class implements the characteristic on its own (because each member is abstract).</td>
</tr>
<tr>
<td>Perspectives</td>
<td>There are three different perspectives for looking at objects: conceptual, specification, and implementation. These distinctions are helpful in understanding the relationship between abstract classes and their derivations. The abstract class defines how to solve things conceptually. It also gives the specification for communicating with any object derived from it. Each derivation provides the specific implementation needed.</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>Being able to refer to different derivations of a class in the same way, but getting the behavior appropriate to the derived class being referred to.</td>
</tr>
</tbody>
</table>

* Some object-oriented analysts speak of everything as an object: Classes are objects, instances are objects. This may be technically correct, but has been the point of confusion and some controversy. For purposes of this book, an object is considered to be an instance of a class.
Object-Oriented Programming in Action

New example
Let’s reexamine the shapes example discussed at the beginning of the chapter. How would I implement it in an object-oriented manner? Remember that it has to do the following:
1. Locate the list of shapes in the database.
2. Open up the list of shapes.
3. Sort the list according to some rules.
4. Display the individual shapes on the monitor.

To solve this in an object-oriented manner, I need to define the objects and the responsibilities they have.

Using objects in the Shape program
The objects I would need are listed in the following table.

<table>
<thead>
<tr>
<th>Class</th>
<th>Responsibilities (Methods)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShapeDataBase</td>
<td>getCollection—Gets a specified collection of shapes</td>
</tr>
<tr>
<td>Shape (an abstract class)</td>
<td>display—Defines interface for Shapes</td>
</tr>
<tr>
<td></td>
<td>getX—Returns X location of Shape (used for sorting)</td>
</tr>
<tr>
<td></td>
<td>getY—Returns Y location of Shape (used for sorting)</td>
</tr>
<tr>
<td>Square (derived from Shape)</td>
<td>display—Displays a square (represented by this object)</td>
</tr>
<tr>
<td>Circle (derived from Shape)</td>
<td>display—Displays a circle (represented by this object)</td>
</tr>
<tr>
<td>Collection</td>
<td>display—Tells all contained shapes to display</td>
</tr>
<tr>
<td></td>
<td>sort—Sorts the collection of shapes</td>
</tr>
<tr>
<td>Display</td>
<td>drawLine—Draws a line on the screen</td>
</tr>
<tr>
<td></td>
<td>drawCircle—Draws a circle on the screen</td>
</tr>
</tbody>
</table>

Running the program
The main program would now look like this:
1. Main program creates an instance of the database object.
2. Main program asks the database object to find the set of shapes I am interested in and to instantiate a collection object containing all the shapes. (Actually, it will instantiate circles and squares that the collection will hold.)
3. Main program asks the collection to sort the shapes.
4. Main program asks the collection to display the shapes.
5. The collection asks each shape it contains to display itself.
6. Each shape displays itself (using the Display object) according to the type of shape I have.

Why this helps—handling new requirements
Let’s see how this helps to handle new requirements. (Remember, requirements always change.) For example, consider the following new requirements:

- Add new kinds of shapes (such as a triangle). To introduce a new kind of shape, only two steps are required:
  - Create a new derivation of Shape that defines the shape.
  - In the new derivation, implement a version of the display method that is appropriate for that shape.
• **Change the sorting algorithm.** To change the method for sorting the shapes, only one step is required:
  – Modify the method in *Collection*. Every shape will use the new algorithm.

**Bottom line:** The object-oriented approach has limited the impact of changing requirements.

**Encapsulation revisited**

There are several advantages to encapsulation. The fact that it hides things from the user directly implies the following:

- Using things is easier because the user does not need to worry about implementation issues.
- Implementations can be changed without worrying about the caller. (Because the caller didn’t know how it was implemented in the first place, there shouldn’t be any dependencies. Remember that it is often the learning and keeping aware of these dependencies that take time in maintenance—more than actually adding a new function.)
- The internals of an object are unknown to other objects—they are used by the object to help implement the function specified by the object’s interface.

**Benefit: Reduced side effects**

Finally, consider the problem of unwanted side effects that arise when functions are changed. This kind of bug is addressed effectively with encapsulation. The internals of objects are unknown to other objects. If I use encapsulation and follow the strategy that objects are responsible for themselves, the only way to affect an object will be to call a method on that object. The object’s data and the way it implements its responsibilities are shielded from changes caused by other objects.

**Encapsulation Saves Us**

- The more I make my objects responsible for their own behaviors, the less the controlling programs have to be responsible for.
- Encapsulation makes changes to an object’s internal behavior transparent to other objects.
- Encapsulation helps to prevent unwanted side effects.

It is worth noting how encapsulation relates to coupling. When I encapsulate something, I am necessarily loosely coupled to it. Hiding implementations (encapsulating them) thus promotes loosecoupling.

**Special Object Methods**

**Creating and destroying**

I have talked about methods that are called by other objects or possibly used by an object itself. But what happens when objects are created? What happens when they go away? If objects are self-contained units, it would be a good idea to have methods to handle these situations.

**Constructors initialize, or set up, an object**

These special methods do, in fact, exist. They are called constructors and destructors or finalizers.

A constructor is a special procedure that is automatically called when the object is created. Its purpose is to handle starting up the object. This is part of an object’s mandate to be responsible for itself. The constructor is the natural place to do initializations, set default information, set up relationships with other objects, or do anything else that is needed to make a well-defined object. All object-oriented languages look for a constructor and execute it when the object is created.
By using constructors properly, it is easier to eliminate (or at least minimize) uninitialized variables. This type of error usually occurs from carelessness on the part of the developer. By having a set, consistent place for all initializations throughout your code (that is, the constructors of your objects), it is easier to ensure that initializations take place. Errors caused by uninitialized variables are easy to fix but hard to find, so this convention (with the automatic calling of the constructor) can increase the efficiency of programmers.

**Destructors (finalizers) clean up an object when it is no longer needed (when it has been deleted)**

Most object-oriented languages provide a way for an object to clean up after itself when the object goes out of existence; that is, when the object is destroyed. In C++ and C#, this is called a destructor, in Java, it is called a finalizer. In this book, I refer to it by the generic term destructor.

All object-oriented languages look for a destructor and execute it when the object is being deleted. As with the constructor, the use of the destructor is part of the object’s mandate to be responsible for itself.

Destructors are typically used for releasing resources when objects are no longer needed. Because Java has garbage collection (auto-cleanup of objects no longer in use), destructors are not as important in Java as they are in C++. In C++, it is common for an object’s destructor also to destroy other objects that are used only by this object.
Summary

In this chapter

This chapter has shown how object orientation helps us minimize consequences of shifting requirements on a system and how it contrasts with functional decomposition.

I have covered a number of the essential concepts in object-oriented programming and have introduced and described the primary terminology. Table 1-3 summarizes these concepts, and Table 1-4 summarizes the primary terminology of object-oriented programming.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Review</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional decomposition</td>
<td>Structured programmers usually approach program design with functional decomposition. Functional decomposition is the method of breaking down a problem into smaller and smaller functions. Each function is subdivided until it is manageable.</td>
</tr>
<tr>
<td>Changing requirements</td>
<td>Changing requirements are inherent to the development process. Rather than blaming users or ourselves about the seemingly impossible task of getting good and complete requirements, we should use development methods that deal with changing requirements more effectively.</td>
</tr>
<tr>
<td>Objects</td>
<td>Objects are defined by their responsibilities. Objects simplify the tasks of programs that use them by being responsible for themselves.</td>
</tr>
<tr>
<td>Constructors and destructors</td>
<td>An object has special procedures that are called when it is created and deleted. These special procedures are:</td>
</tr>
<tr>
<td></td>
<td>• Constructors, which initialize or set up an object.</td>
</tr>
<tr>
<td></td>
<td>• Destructors, which clean up an object when it is deleted. All object-oriented languages use constructors and destructors to help manage objects.</td>
</tr>
</tbody>
</table>
### Table 1-4 Object-Oriented Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract class</td>
<td>Defines the methods and common attributes of a set of classes that are conceptually similar. Abstract classes are never instantiated.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Data associated with an object (also called a data member).</td>
</tr>
<tr>
<td>Class</td>
<td>Blueprint of an object—defines the methods and data of an object of its type.</td>
</tr>
<tr>
<td>Constructor</td>
<td>Procedure that is invoked when an object is created.</td>
</tr>
<tr>
<td>Derived class</td>
<td>A class that is specialized from a base class. Contains all of the attributes and methods of the base class but may also contain other attributes or different method implementations.</td>
</tr>
<tr>
<td>Destructor</td>
<td>Procedure that is invoked when an object is deleted. (Note: In Java, this is called a finalizer.)</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>Any kind of hiding. Objects encapsulate their data. Abstract classes encapsulate their derived concrete classes.</td>
</tr>
<tr>
<td>Functional</td>
<td>A method of analysis in which a problem is broken into smaller and smaller functions.</td>
</tr>
<tr>
<td>decomposition</td>
<td></td>
</tr>
<tr>
<td>Inheritance</td>
<td>The way that a class is specialized, used to relate derived classes with their base classes.</td>
</tr>
<tr>
<td>Instance</td>
<td>A particular object of a class.</td>
</tr>
<tr>
<td>Instantiation</td>
<td>The process of creating an instance of a class.</td>
</tr>
<tr>
<td>Member</td>
<td>Either data or a procedure of a class.</td>
</tr>
<tr>
<td>Method</td>
<td>Procedures that are associated with a class.</td>
</tr>
<tr>
<td>Object</td>
<td>An entity with responsibilities. A special, self-contained holder of both data and procedures that operate on that data. An object's data is protected from external objects.</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>The ability of related objects to implement methods that are specialized to their type.</td>
</tr>
<tr>
<td>Superclass</td>
<td>A class from which other classes are derived. Contains the master definitions of data And procedures that all derived classes will use (and for procedures, possibly override).</td>
</tr>
</tbody>
</table>
Review Questions

Observations
1. Describe the basic approach used in functional decomposition.
2. What are three reasons that cause requirements to change?
3. I advocate thinking about responsibilities rather than functions. What is meant by this? Give an example.
4. Define coupling and cohesion. What is tight coupling?
5. What is the purpose of an interface to an object?
6. Define instance of a class.
7. A class is a complete definition of the behavior of an object. What three aspects of an object does it describe?
8. What does an abstract class do?
9. What are the three main types of accessibility that objects can have?
10. Define encapsulation. Give one example of encapsulation of behavior
11. Define polymorphism. Give one example of polymorphism
12. What are the three perspectives for looking at objects?

Interpretations
1. Sometimes, programmers use “modules” to isolate portions of code. Is this an effective way to deal with changes in requirements? Why or why not?
2. It is too limited to define an abstract class as a class that does not get instantiated. Why is this definition too limited? What is a better (or at least alternative) way to think about abstract classes?
3. How does encapsulation of behavior help to limit the impact of changes in requirements? How does it save programmers from unintended side effects?
4. How do interfaces help to protect objects from changes that are made to other objects?
5. A classroom is used to describe objects in a system. Describe this classroom from the conceptual perspective.

Opinions and Applications
1. Changing requirements is one of the greatest challenges faced by systems developers. Give one example from your own experience where this has been true.
2. There is a fundamental weakness in functional decomposition when it comes to changes in requirements. Do you agree? Why or why not?
3. What do you think is the best way to deal with changing requirements?

To join a discussion about this article, please go to http://www.netobjectivesgroups.com/6/ubb.x and look under the E-zines category.
Net Objectives - What We Do

Net Objectives provides enterprises with a full selection of training, coaching and consulting services. Our Vision is “Effective software development without suffering”. We facilitate software development organizations migration to more effective and efficient processes. In particular, we are specialists in agility, effective analysis, design patterns, refactoring and test-driven development.

We provide a blend of training, follow up coaching and staff supplementation that enables your team to become more effective in all areas of software development. Our engagements often begin with an assessment of where you are and detail a plan of how to become much more effective. Our trainers and consultants are experts in their fields (many of them published authors).

When you've taken a course from Net Objectives, you will see the world of software development with new clarity and new insight. Our graduates often tell us they are amazed at how we can simplify confusing and complicated subjects, and make them seem so understandable, and applicable for everyday use. Many of our students remark that it is the best training they have ever received.

The following courses are among our most often requested. This is not a complete list, though it is representative of the types of courses we offer

**Agile Project Management** - This course analyzes what it means to be an agile project, and provides a number of best practices that provide and/or enhance agility. Different agile practices from different processes (including RUP, XP and Scrum) are discussed.

**Agile Use Case Analysis** - This 3-day course provides theory and practice in deriving software requirements from use cases. This course is our recommendation for almost all organizations, and is intended for audiences that mix both business and software analysts.

**Design Patterns Explained: A New Perspective on Object-Oriented Design** - This 3-day course teaches several useful design patterns as a way to explain the principles and strategies that make design patterns effective. It also takes the lessons from design patterns and expands them into both a new way of doing analysis and a general way of building application architectures.

**Test-Driven Development: Iterative Development, Refactoring, and Unit Testing** - This course teaches how to use either Junit, NUnit or CppUnit to create unit tests. Lab work is done in both Refactoring and Unit Testing together to develop very solid code by writing tests first. The course explains how the combination of Unit Testing Refactoring can result in emerging designs of high quality.

**Effective Object-Oriented Analysis, Design and Programming In C++, C#, Java, or VB.net** - This 5-day course goes beyond the basics of object-oriented design and presents 14 practices which will enable your developers to do object-oriented programming effectively. These practices are the culmination of the best coding practices of eXtreme Programming and of Design Patterns.

If you are interested in any of these offerings, if your user group or company is interested in Net Objectives making a free technical presentation to them, or if you would like to be notified of upcoming Net Objectives events, please visit our website, or contact us by the email address or phone number below:

www.netobjectives.com • mike.shalloway@netobjectives.com • 404-593-8375

For a complete schedule of upcoming Public Courses and information on how to register, go to

www.netobjectives.com/events/pr_main.htm#UpcomingPublicCourses