



Chapter 1

Program Development

Coming Up

First, there is a need for a computer-based solution to a problem. The need may be expressed in a few sentences as a **problem specification**. The progression from understanding a problem specification to achieving a working computer-based solution is known as “program development.” After studying this chapter, you will be able to

- follow a case study using a simple program development strategy
- understand the characteristics of a good algorithm
- understand how algorithmic patterns help in program design
- provide a deliverable from the analysis phase of program development
- provide a deliverable from the design phase of program development

Analysis and design projects reinforce your ability to develop programs.

1.1 Program Development

There are many approaches to program development. This chapter begins by examining a strategy with these three steps: analysis, design, and implementation.

Phase of Program Development	Activity
Analysis	Understand the problem.
Design	Develop a solution.
Implementation	Make the solution run on a computer.

Our study of computing fundamentals begins with an example of this particular approach to program development.

Each of these three phases will be exemplified with a simple case study—one particular problem. Emphasis is placed on the **deliverables**—the tangible results—of each phase. Here is a preview of the deliverables for each of the three stages:

Phase	Deliverable
Analysis	A document that lists the data that store relevant information
Design	An algorithm that outlines a solution
Implementation	An executable program ready to be used by the customer (or ready to be graded)

Self-Check

- 1-1 What deliverable can you expect at the end of a successfully completed college career?

1.2 Analysis

Synonyms for analysis: inquiry, examination, study

Program development may begin with a study, or **analysis**, of a problem. Obviously, to determine what a program is to do, you must first understand the problem. If the problem is written down, you can begin the analysis phase by simply reading the problem.

While analyzing the problem, it proves helpful to name the data that represent information. For example, you might be asked to compute the maximum weight allowed for a successful liftoff of a particular airplane from a given runway under certain thrust-affecting weather conditions such as temperature and wind direction. While analyzing the problem specification, you might name the desired information `maximumWeight`. The data required to compute that information could have names such as `temperature` and `windDirection`.

Although such data do not represent the entire solution, they do represent an important piece of the puzzle. The data names are symbols for what the program will need and what the program will compute. One value needed to compute `maximumWeight` might be 19.0 for `temperature`. Such data values must often be manipulated—or processed—in a variety of ways to produce the desired result. Some values must be obtained from the user, other values must be multiplied or added, and still other values must be displayed on the computer screen.

At some point, these data values will be stored in computer memory. The values in the same memory location can change while the program is running. The values also have a type, such as integers or numbers with decimal points (these two different types of values are stored differently in computer memory). These named pieces of memory that store a specific type of value that can change while a program is running are known as **variables**.

You will see that there also are operations for manipulating those values in meaningful ways. It helps to distinguish the data that must be displayed—**output**—from the data required to compute that result—**input**. These named pieces of memory that store values are the variables that summarize what the program must do.

Input and Output

Output: Information the computer must display.

Input: Information a user must supply to solve a problem.

A problem can be better understood by answering this question: What is the output given certain input? Therefore, it is a good idea to provide an example of the problem with pencil and paper. Here are two problems with variable names selected to accurately describe the stored values.

Problem 1 Analysis Deliverable

Problem	Data Name	Input or Output	Sample Problem
Compute a monthly loan payment	<code>amount</code>	Input	12500.00
	<code>rate</code>	Input	0.08
	<code>months</code>	Input	48
	<code>payment</code>	Output	303.14

Problem 2 Analysis Deliverable

Problem	Data Name	Input or Output	Sample Problem
Count how often Shakespeare wrote a particular word in a particular play	<code>aBardsWork</code>	Input	Much Ado About Nothing
	<code>theWord</code>	Input	the
	<code>howOften</code>	Output	220

In summary, problems are analyzed by doing these things:

1. Reading and understanding the problem specification.
2. Deciding what data represent the answer—the output.
3. Deciding what data the user must enter to get the answer—the input.
4. Creating a document (like those above) that summarizes the analysis. This document is input for the next phase of program development—design.

In textbook problems, the variable names and type of values (such as integers or numbers with a decimal point) that must be input and output are sometimes provided. If not, they are relatively easy to recognize. In real-world problems of significant scale, a great deal of effort is expended during the analysis phase. The next subsection provides an analysis of a small problem.

Self-Check

- 1-2** Given the problem of converting British pounds to U.S. dollars, name a value that must be input by the user and a value that must be output.
- 1-3** Given the problem of selecting one CD from a 200-compact-disc player, what name would represent all of the CDs? What name would be appropriate to represent one particular CD selected by the user?

An Example of Analysis

Problem: Using the grade assessment scale to the right, compute a course grade as a weighted average of two tests and one final exam. The dialogue must look exactly like this for the given input of 74.0, 79.0, and 84.0:

```
Enter first test: 74.0
Enter second test: 79.0
Enter final exam: 84.0
Course Grade: 80.25%
```

	Percentage of Final Grade
Test 1	25%
Test 2	25%
Final Exam	50%

Analysis begins by reading the problem specification and establishing the desired output and the required input to solve the problem. Determining and naming the output is a good place to start. The output stores the answer to the problem. It provides insight into what the program must do. Once the need for a data value is discovered and given a meaningful name, the focus can shift to what must be accomplished. For this particular problem, the desired output is the actual course grade (80.25% above). The name `courseGrade` represents the requested information to be output to the user.

A complete analysis might answer other questions, such as:

The problem has now been analyzed, the input and output have been named, it is understood what the computer-based solution is to do, and one sample problem has been given. The following deliverable from the analysis phase summarizes these activities:

Course-Grade Problem Analysis Deliverable

Problem	Data Name	Input or Output	Sample Problem
Compute a course grade	test1	Input	74.0
	test2	Input	79.0
	finalExam	Input	84.0
	courseGrade	Output	80.25

This is the first deliverable. The next section presents a method for designing a solution. The emphasis during design is on placing the appropriate activities in the proper order to solve the problem.

Self-Check

- 1-4** Complete an analysis deliverable for the following problem:
Problem: Compute the distance traveled for any moving vehicle. Allow the user to choose the unit of measurement (meters per second, miles per hour, or kilometers per hour, for example).
- 1-5** Complete an analysis deliverable for the following problem. You will need a calculator to determine the output.
Problem: Show the future value of an investment given its present value, the number of periods (years, perhaps), and the interest rate. Be consistent with the interest rate and the number of periods; if the periods are in years, then the annual interest rate must be supplied (0.085 for 8.5%, for example). If the period is in months, the monthly interest rate must be supplied (0.0075 per month for 9% per year, for example). The formula to compute the future value of money is $\text{future value} = \text{present value} * (1 + \text{rate})^{\text{periods}}$.

1.3 Design

Synonyms of design: model, think, plan, devise, pattern, outline

Design refers to the set of activities that includes (1) defining an architecture for the program that satisfies the requirements and (2) specifying an algorithm for each program component in the architecture.¹ In later chapters, you will see

1. Alan M. Davis, *201 Principles of Software Development* (IEEE Computer Society, 1995).

functions used as the basic building blocks of programs. Then you will see classes used as the basic building blocks of programs. A class is a collection of functions, typically called “methods.” In this chapter, the architecture is intentionally constrained to a component known as a **program**. Therefore, the design activity that follows is limited to specifying an algorithm for this program.

An **algorithm** is a step-by-step procedure for solving a problem or accomplishing some end, especially by a computer.² A good algorithm must

- list the activities that need to be carried out
- list those activities in the proper order

Consider an algorithm to bake a cake:

1. Preheat the oven.
2. Grease the pan.
3. Mix the ingredients.
4. Pour the ingredients into the pan.
5. Place the cake pan in the oven.
6. Remove the cake pan from the oven after 35 minutes.

If the order of the steps is changed, the cook might get a very hot cake pan with raw cake batter in it. If one of these steps is omitted, the cook probably won’t get a baked cake—or there might be a fire. An experienced cook may not need such an algorithm. However, cake-mix marketers cannot and do not presume that their customers have this experience. Good algorithms list the proper steps in the proper order and are detailed enough to accomplish the task.

Self-Check

- 1-6** Cake recipes typically omit a very important activity. Describe an activity that is missing.

An algorithm often contains a step without much detail. For example, step 3, “Mix the ingredients,” isn’t very specific. What are the ingredients? If the problem is to write a recipe algorithm that humans can understand, step 3 should be refined a bit to instruct the cook on how to mix the ingredients. The refinement to step 3 could be something like this:

3. Empty the cake mix into the bowl and mix in the milk until smooth.

or for scratch bakers:

- 3a. Sift the dry ingredients.
- 3b. Place the liquid ingredients in the bowl.
- 3c. Add the dry ingredients a quarter-cup at a time, whipping until smooth.

2. Merriam-Webster, *Merriam-Webster’s Collegiate Dictionary*, 10th ed. (International Thomson Publishing, 1998).

Algorithms may be expressed in **pseudocode**—instructions expressed in a language that even nonprogrammers could understand. Pseudocode is written for humans, not for computers. Pseudocode algorithms are an aid to program design.

Pseudocode is very expressive. One pseudocode instruction may represent many computer instructions. Pseudocode algorithms are not concerned about issues such as misplaced punctuation marks or the details of a particular computer system. Pseudocode solutions make design easier by allowing details to be deferred. Writing an algorithm can be viewed as planning. A program developer can design with pencil and paper and sometimes in her or his head.

Algorithmic Patterns

Problems often require input from the user in order to compute and display the desired information. This particular flow of three activities—input/process/output—occurs so often, in fact, that it can be viewed as a pattern. It is one of several algorithmic patterns acknowledged in this textbook. These patterns will help you design programs.

A **pattern** is anything shaped or designed to serve as a model or a guide in making something else [Funk/Wagnalls 1968]. An **algorithmic pattern** serves as a guide to help develop programs. For instance, the following **Input/Process/Output (IPO) pattern** can be used to help design your first programs. In fact, this pattern will provide a guideline for many programs.

Algorithmic Pattern 1.1

Pattern: Input/Process/Output (IPO)
 Problem: The program requires input from the user in order to compute and display the desired information.
 Outline: 1. Obtain the input data.
 2. Process the data in some meaningful way.
 3. Output the results.

This algorithmic pattern is the first of several. In subsequent chapters, you'll see other algorithmic patterns, such as Guarded Action, Alternative Action, and Multiple Selection. To use an algorithmic pattern effectively, you should first become familiar with it. Look for the Input/Process/Output algorithmic pattern while developing programs. This could allow you to design your first programs more easily. For example, if you discover you have no meaningful values for the input data, it may be because you have placed the process step *before* the input step. Alternately, you may have skipped the input step altogether.

Patterns help solve other kinds of problems. Consider this quote:³

3. Christopher Alexander, *A Pattern Language* (Oxford University Press, 1977).

Each pattern 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 this solution a million times over, without ever doing it the same way twice.

Alexander is describing patterns in the design of furniture, gardens, buildings, and towns, but his description of a pattern can also be applied to program development. The IPO pattern frequently pops up during program design. It guides the solution to many problems—especially in the first five chapters of this textbook.

An Example of Algorithm Design

The deliverable from the design phase is an algorithm that solves the problem. The Input/Process/Output pattern guides the design of the algorithm that relates to our `courseGrade` problem.

<u>Three-Step Pattern</u>	<u>Pattern Applied to a Specific Algorithm</u>
1. Input	1. Obtain <code>test1</code> , <code>test2</code> , and <code>finalExam</code>
2. Process	2. Compute <code>courseGrade</code>
3. Output	3. Display <code>courseGrade</code>

Although algorithm development is usually an iterative process, a pattern helps to quickly provide an outline of the activities necessary to solve the `courseGrade` problem.

Self-Check

- 1-7** Read the three activities of the algorithm above. Do you detect a missing activity?
- 1-8** Read the three activities of the algorithm above. Do you detect any activity out of order?
- 1-9** Would this previous algorithm work if the first two activities were switched?
- 1-10** Is there enough detail in this algorithm to correctly compute `courseGrade`?

There currently is not enough detail in the process step of the `courseGrade` problem. The algorithm needs further refinement. Specifically, exactly how should the input data be processed to compute the course grade? The algorithm omits the weighted scale specified in the problem specification. The process step should be refined a bit more. Currently, this pseudocode algorithm does not describe how `courseGrade` must be computed.

The refinement of this algorithm (below) shows a more detailed process step. The step “Compute `courseGrade`” is now replaced with a **refinement**—a more detailed and specific activity. The input and output steps have also been refined.

This is the design phase deliverable—an algorithm with enough detail to pass on as the input into the next phase, implementation.

Refinement of a Specific Input/Process/Output (IPO) Algorithm

1. Obtain `test1`, `test2`, and `finalExam` from the user.
2. Compute `courseGrade = (25% of test1) + (25% of test2) + (50% of finalExam)`.
3. Display the value of `courseGrade`.

Try to think of program development in terms of the deliverables. This provides a checklist. What deliverables exist so far?

1. From the analysis phase there is a document with a list of data (variables) and a sample problem.
2. From the design phase there is an algorithm.

Algorithm Walkthrough

Programs can be developed more quickly and with fewer errors by reviewing algorithms before moving on to the implementation phase. Are the activities in the proper order? Are all the necessary activities present? This review is accomplished with an algorithm walkthrough. An **algorithm walkthrough** simulates what a computer would do by stepping through the instructions of the algorithm.

A **computer** is a programmable electronic device that can store, retrieve, and process data. Programmers can simulate an electronic version of the algorithm by following the algorithm and manually performing the activities of storing, retrieving, and processing data using pencil and paper. The following algorithm walkthrough is a human (non-electronic) execution of the algorithm:

1. Retrieve some example values from the user and store them as shown:

```
test1:      80
test2:      90
finalExam: 100
```

2. Retrieve the values and compute `courseGrade` as follows:

```
courseGrade = (0.25 x test1) + (0.25 x test2) + (0.50 x finalExam)
              (0.25 x 80.0)  + (0.25 x 90.0)  + (0.50 x 100.0)
              20.0          + 22.5          + 50.0
courseGrade = 92.5
```

3. Show the course grade to the user by retrieving the data stored in `courseGrade` to show 92.5%.

It has been said that good artists know when to put down the brushes. Deciding when a painting is done is critical for its success. By analogy, a designer must decide when to stop designing. This is a good time to move on to the third phase of program development. In summary, here is what has been accomplished so far:

- The problem is understood.
- Data have been identified and named.
- Output for two sample problems is known (80.25% and now 92.5%).
- An algorithm has been developed.
- Walking through the algorithm has simulated computer activities.

Self-Check

1-11 Walk through the previous algorithm when `test1` is input as 0.0, `test2` is input as 50.0, and `finalExam` is input as 100.0. What value is stored in `courseGrade`?

1.4 Implementation

Synonyms for implementation: accomplishment, fulfilling, making good, execution

The analysis and design of simple problems could be done with pencil and paper. The implementation phase of program development requires both software and hardware to obtain the deliverable. The deliverable of the implementation phase is a program that runs correctly on a computer. **Implementation** is the collection of activities required to complete the program so someone else can use it. Here are some implementation phase activities and associated deliverables:

Activity	Deliverable
Translate an algorithm into a programming language.	Source code
Compile source code into byte code.	Byte code
Run the program.	A running program
Verify that the program does what it is supposed to do.	A grade

This is neither the time nor place to describe all of the details of how programs actually run on computers. However, the gist of it can be understood by viewing the computer and its programs in a hierarchy of abstractions referred to as “virtual machines” (see the table below) [Aho/Ullman 1992].

Level	Virtual Machine (Abstraction)
6	Application software
5	High-level programming language
4	Assembly language or Java byte code
3	Operating system
2	Machine-level programming language
1	Microprogram
0	Digital logic

Most computer users view the machine at its highest level—the application software. Examples of software at this level include word processors, games, Web browsers, appointment calendars, and realtime applications that control electromechanical devices.

Programmers view the computer as a machine that responds to messages written at level 5. They use a high-level programming language such as Java, C++, or Smalltalk to write the messages that will be translated by other software that eventually will control the computer.

Our view of the computer in the implementation phase will be at the level-5 abstraction—the high-level programming language. This implies that the computer is a machine controlled by programming language instructions written in a high-level programming language. There are hundreds of level-5 programming languages that could be used. In this textbook, you will be using a programming language named Java. Java is the tool you will learn to use in order to control what happens at the computer.

Whereas the design phase provided a solution in the form of a pseudocode algorithm, the implementation phase requires nitty-gritty details. The programming language translation must be written in a precise manner according to the syntax rules of that programming language. Attention must be paid to the placement of semicolons, commas, and periods. For example, an algorithmic statement such as this:

3. Display the value of `courseGrade`.

could be translated into Java source code that might look like this:

```
System.out.println( "Course Grade: " + courseGrade + "%" );
```

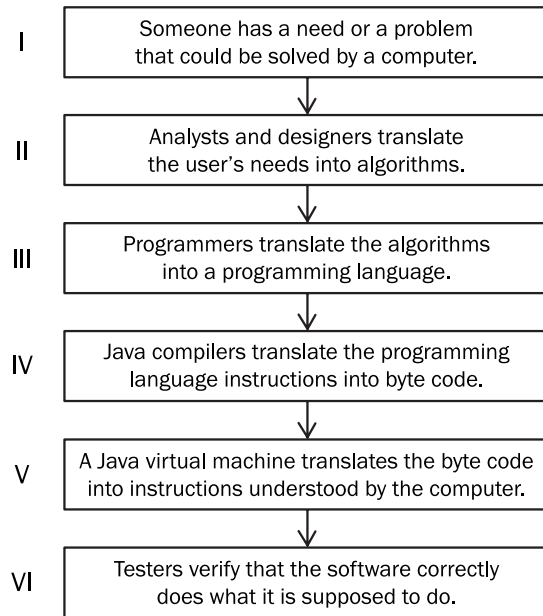
This output step generates output to the computer screen that might look like this (assuming the state of `courseGrade` is 92.5):

```
Course Grade: 92.5%
```

Once a programmer has translated the user's needs into pseudocode and then into a programming language, software is utilized to translate level-5 instructions into the lower levels of the computer. Fortunately, there is a tool for performing these translations. Programmers use a compiler to translate the high-level programming language source code (such as Java) into its byte code equivalent. This byte code can then be sent to any machine with a Java virtual machine (VM) on it. The **Java virtual machine** then converts the byte code into the machine language of that particular machine. In this way, the Java program works on a variety of platforms, such as Unix, Mac OS, and Microsoft Windows. Finally, to verify that the program works, the behavior of the executable program must be observed. Input data may be entered, and the corresponding output is observed. The output is compared to what was expected. If the two match, the program works for at least one particular set of input data.

Other sets of input data can be entered while the program is running to build confidence that the program works as defined by the problem specification. Program development is summarized in Figure 1.1.

Figure 1.1: A review of program development



Although you will likely use the same compiler as in industry, the roles of people will differ. In large software organizations, many people—usually in teams—perform analysis, design implementation, and testing. In many of these simple textbook problems, the user needs are what your instructor requires, usually for grade assessment. You will often play the role of analyst, designer, programmer, *and* tester—perhaps as part of a team, but for the most part by yourself.

Self-Check

- 1-12** Review Figure 1.1 above and list the phases that are **-a** primarily performed by humans and **-b** primarily performed by software. Select your answers from the set of I, II, III, IV, V, and VI.
- 1-13** Must the level-6 user of software applications understand the level-5 programming language to use the computer?
- 1-14** Must the programmer implement the level-5 compiler that translates source code into lower levels?

An Example of Implementation

The following program—a complete Java translation of the algorithm—previews many programming language details. You are not expected to understand this Java code. The details are presented later. For now, just peruse the Java code as an implementation of the pseudocode algorithm. The three variables `test1`, `test2`, and `finalExam` represent user input. The output variable is named `courseGrade`. User input is made possible through an object named `keyboard` (objects and the `TextReader` class are discussed in more detail in Chapters 2 and 3). To get this code to work, you would need to copy the file named `TextReader.java`⁴ into the same folder as this file.

```
// This program computes and displays a final course grade as a
// weighted average after the user enters the appropriate input.

import TextReader;    // An author-supplied class for easy input

public class TestCourseGrade
{
    public static void main( String[] args )
    {
        // Declare the input and output variables
        double test1 = 0.0;
        double test2 = 0.0;
        double finalExam = 0.0;
        double courseGrade = 0.0;

        System.out.println( "This program computes a course grade when" );
        System.out.println( "you have entered three requested values." );
        System.out.println( );

        // Construct an object for reading input easily
        TextReader keyboard = new TextReader( );

        // Input test1, test2, and finalExam
        System.out.print( "Enter first test: " );
        test1 = keyboard.readDouble( );

        System.out.print( "Enter second test: " );
        test2 = keyboard.readDouble( );

        System.out.print( "Enter final exam: " );
        finalExam = keyboard.readDouble( );
    }
}
```

4. `TextReader` provides an easy way to read input from the user. However, `TextReader` is not part of Java. `TextReader.java` must be copied from the disk that accompanies this text-book.

```

// P)rocess
courseGrade = ( 0.25 * test1 )
              + ( 0.25 * test2 )
              + ( 0.50 * finalExam );

// O)utput the results
System.out.println( "Course Grade: " + courseGrade + "%" );

} // End of the main method

} // End class TestCourseGrade

```

Dialogue

This program computes a course grade when you have entered three requested values.

```

Enter first test: 80.0
Enter second test: 90.0
Enter final exam: 100.0
Course Grade: 92.5%

```

At the end of most programs in this textbook, you will find a section titled either “Dialogue” or “Output.” A “Dialogue” section shows the program output along with user input in boldface italic. If there is no user input, the section will simply be titled “Output.” These dialogues help explain how programs execute so you can more easily learn how to read Java code. Read before you write.

Testing

Although this “Testing” section appears at the end of our first example of program development, don’t presume that testing is deferred until implementation. The important process of **testing** may, can, and should occur at any phase of program development. The actual work can be minimal, and it’s worth the effort. However, you may not agree until you have felt the pain of *not* testing.

Testing During All Phases of Program Development

- During analysis, establish sample problems to confirm your understanding of the problem.
- During design, walk through the algorithm to ensure that it has the proper steps in the proper order.
- During testing, run the program (or method) several times with different sets of input data. Confirm that the results are correct.
- Review the problem specification. Does the running program do what was requested?

You should have a sample problem before the program is coded—not after. However, if you do decide to wait, it’s still better late than never. Work out an

example now. If nothing else, at least look at the input and the result of one program run and convince yourself that the output is correct.

When the Java implementation finally does generate output, the predicted results are then compared to the output of the running program. Adjustments must be made any time the predicted output does not match the program output. Such a conflict indicates that the problem example, the program output, or perhaps both are incorrect. Using problem examples helps avoid the misconception that a program is correct just because the program runs successfully and generates output. The output could be wrong! Simply executing doesn't make a program right.

Even exhaustive testing does not prove a program is correct. E. W. Dijkstra has argued that testing only reveals the presence of errors, not the absence of errors. Even with correct program output, the program is not proven correct. However, testing increases confidence that the algorithm, now implemented as a program, at least appears to be reliable.

Self-Check

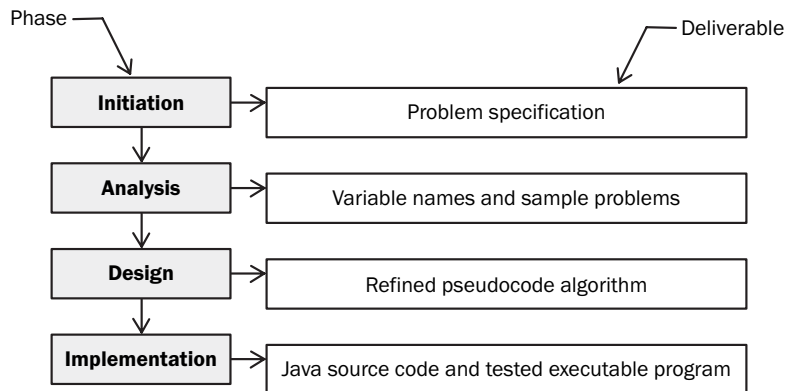
- 1-15** If the programmer predicts `courseGrade` should be `100.0` when all three inputs are `100.0` and the program displays `courseGrade` as `75.0`, what is wrong: the prediction, the program, or both?
- 1-16** If the programmer predicts `courseGrade` should be `90.0` when `test1` is `80`, `test2` is `90.0`, and `finalExam` is `100.0` and the program outputs `courseGrade` as `92.5`, what is wrong: the prediction, the program, or both?
- 1-17** If the programmer predicts `courseGrade` should be `92.5` when `test1` is `80`, `test2` is `90.0`, and `finalExam` is `100.0` and the program outputs `courseGrade` as `90.0`, what is wrong: the prediction, the program, or both?

Chapter Summary

- This chapter presented a three-step program development strategy of analysis, design, and implementation. The table below shows some of the activities performed during each of these three phases. The maintenance phase has been added to show how the three steps fit into the complete program life cycle. The maintenance phase requires the majority of the time, energy, and money of the program's life cycle.

Phase	Activities You Might Perform
Initiation	Specify the problem. Identify the requirements.
Analysis	Read and understand the problem specification. Determine the input and output. Solve a few sample problems.
Design	Look for patterns to guide algorithm development. Write an algorithm—the steps needed to solve the problem. Refine the steps in the algorithm and walk through it.
Implementation	Translate the design into a programming language. Fix errors. Create an executable program. Test the program.
Maintenance	Update the program to keep up with a changing world. Enhance it. Correct bugs (errors) as they're found.

- Each phase of program development can be viewed in terms of the deliverables.










- Some useful analysis and design tools were introduced:
 1. naming the variables that help solve a problem
 2. developing algorithms
 3. refining one or more steps of an algorithm
 4. using the Input/Process/Output algorithmic pattern

Key Terms

algorithm	implementation	problem specification
algorithm walkthrough	input	program
algorithmic pattern	Input/Process/Output (IPO) pattern	pseudocode
analysis	Java virtual machine	refinement
computer	output	testing
deliverable	pattern	variable
design		

Exercises

The answers to exercises marked with  are available in Appendix C.

-  1. What is the difference between variables that store output values and variables that store values input by the user?
2. What activities are performed when you design programs?
-  3. Describe the deliverable of the design phase of program development.
4. What is the deliverable from the implementation phase of program development?
-  5. What are the characteristics of a good algorithm?
6. Does a running program always work correctly? Explain your answer.
-  7. Explain the importance of testing a program before a user gets it.
-  8. Write an algorithm for finding any phone number in the phone book. Will the search always be successful?
9. Write an algorithm that would direct someone from your classroom to the nearest grocery store.
-  10. Write an algorithm that would arrange a collection of CDs (or tapes or record albums) in alphabetical order.
11. Prepare for programming projects at the end of Chapter 2 by procuring a Java development environment. If you are using your school computer lab, Java may already be set up and your instructor should direct you how to use it. If you are working at home, consider downloading Java from Sun Microsystem's Web site: <http://java.sun.com/j2se/>.

Analysis/Design Projects

The following projects do not require a computer. Use a table like this to complete the analysis deliverable for each of the following analysis and design projects:

Problem	Data Name	Input or Output	Sample Problem
---------	-----------	-----------------	----------------

1A Simple Arithmetic



Problem: For any two numbers a and b , compute the product ($a * b$) and the sum ($a + b$). Also, display the difference between the product and the sum. The difference may be computed as positive or negative depending on the input values for a and b .

1. Complete an analysis deliverable (like the table above) for this problem with at least one sample problem.
2. Complete a design deliverable by writing an algorithm that solves the simple arithmetic problem. Include the arithmetic expressions.

1B Simple Average

Problem: Find the average of three tests of equal weight.

1. Complete an analysis deliverable (like the table above) for this problem with at least one sample problem.
2. Complete a design deliverable by writing an algorithm (with the formula) that solves the simple average problem.

1C Weighted Average

Problem: Determine a course grade using this grade assessment scale:

Quiz average	20%
Midterm	20%
Lab grade	35%
Final exam	25%

1. Complete an analysis deliverable (like the table above) for this problem with at least two sample problems.
2. Complete a design deliverable by writing an algorithm (with the formula) that solves the weighted average problem.

1D Wholesale Cost

Problem: You happen to know that a store has a 25% markup on compact-disc players. If the retail price (what you pay) of a CD player is \$189.98, how much did the store pay for that item (the wholesale price)? In general, what is the wholesale price for any item given its retail price and markup? Analyze the problem and design an algorithm that computes the wholesale price for *any* given retail price and *any* given markup. *Clue:* If you can't determine the equation, use this formula and a little algebra to solve for wholesale price:

retail price = wholesale price * (1 + markup)

1. Complete an analysis deliverable (like the table above) for this problem with at least two sample problems.
2. Complete a design deliverable by writing an algorithm (with the formula) that solves the wholesale cost problem.

1E Grade Point Average



Problem: Compute a student's cumulative grade point average (GPA) for three courses. Credits range from 0.5 to 15.0. Grades can be 0.0, 1.0, 2.0, 3.0, or 4.0.

1. Complete an analysis deliverable (like the table above) for the GPA problem with at least one sample problem that has courses with different credits.
2. Complete a design deliverable by writing an algorithm that solves the GPA problem.

1F F to C

Problem: Use the following formula to convert Fahrenheit (F) temperatures to Celsius (C):

1. Complete an analysis deliverable (like the table above) for this problem with at least one sample problem.
2. Complete a design deliverable by writing an algorithm (with the formula) that solves the F to C problem.

1G C to F

Problem: Use algebra and the formula of the preceding exercise to convert Celsius (C) temperatures to Fahrenheit (F).

1. Complete an analysis deliverable (like the table above) for this problem with at least one sample problem.
2. Complete a design deliverable by writing an algorithm (with the formula) that solves the C to F problem.

1H Seconds

Problem: Determine the number of hours, minutes, and seconds represented by a total number of seconds. For example, 3,661 seconds is 1 hour, 1 minute, and 1 second.

1. Complete an analysis deliverable (like the table above) for this problem with at least one sample problem.
2. Complete a design deliverable by writing an algorithm (with the formula) that solves the seconds problem.

1I U.S. Minimum Coins

Problem: Determine the number of each denomination of coins required to make change for cents in the range of 0 through 99. The number of coins should be the lowest possible. The available coins are half dollars (50 cents), quarters (25 cents), dimes (10 cents), nickels (five cents), and pennies (one cent). For example, if there is 91 cents change, you will have one of each coin.

1. Complete an analysis deliverable (like the table above) for this problem with at least one sample problem other than 91 cents.
2. Complete a design deliverable by writing an algorithm (with the formula) that solves the U.S. minimum coins problem.

1J U.K. Minimum Coins

Problem: Determine the coins required to make change for any number of pence from 0 to 199. The number of coins should be the lowest possible. The available coins are 1p (p represents pence), 2p, 5p, 10p, 20p, 50p, and 100p (the one-pound coin). For example, if there is 188 pence, you will have one of each coin.

1. Complete an analysis deliverable (like the table above) for this problem with at least one sample problem other than 188 pence.
2. Complete a design deliverable by writing an algorithm (with the formula) that solves the U.K. minimum coins problem.

Answers to Self-Check Questions

- 1-1 A college degree perhaps (other deliverables are possible)
 1-2 Input: pounds and perhaps todaysConversionRate
 Output: USDollars
 1-3 CDCollection, currentSelection

1-4	Problem	Data Name	Input or Output	Sample Problem
	Compute distance traveled	metersPerSecond secondsInFlight distance	Input Input Output	28.0 3.1 86.8
1-5	Problem	Data Name	Input or Output	Sample Problem
	Compute the future value of an investment	presentValue periods monthlyInterestRate futureValue	Input Input Input Output	1000.00 360 0.0075 14730.58

- 1-6 Turn the oven off (or you might recognize some other activity or detail that was omitted).
- 1-7 No (at least the author thinks it's okay)
- 1-8 No (at least the author thinks it's okay)
- 1-9 No. The `courseGrade` would be computed using undefined values for `test1`, `test2`, and `finalExam`.
- 1-10 No. The details of the process step are not present. The formula is missing.
- 1-11 Input: Retrieve values from the user for the three inputs:
`test1 0`
`test2 50`
`finalExam 100`
- Process: Retrieve the values of the tests and compute `courseGrade` as follows:
- $$\text{courseGrade} = (0.25 \times \text{test1}) + (0.25 \times \text{test2}) + (0.50 \times \text{finalExam})$$
- $$\begin{array}{r} (0.25 \times 0.0) + (0.25 \times 50.0) + (0.50 \times 100.0) \\ (0.0 + 12.5 + 50.0) \end{array}$$
- `courseGrade = 62.5`
- Output: Show `courseGrade = 62.5%` to the user.
- 1-12 -a I, II, III, and VI
 -b IV and V
- 1-13 No
- 1-14 No
- 1-15 The program is wrong.
- 1-16 The prediction is wrong. The problem asked for a weighted average, not a simple average.
- 1-17 The program is wrong.