Chapter 1

Program Development

First, there is a need for a computer-based solution to a problem. The need may be expressed in a few sentences like the first examples in this book. The progression from understanding a problem specification to achieving a working computer-based solution is known as "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

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.

Analysis Deliverable

Problem	Data Name	Input or Output	Sample Problem
Compute a monthly	`amount`	Input	`12500.00`
loan payment	`rate`	Input	`0.08`
	`months`	Input	`48`
	`payment`	Output	`303.14`

Analysis Deliverable

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

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

1-1 Given the problem of converting British pounds to U.S. dollars, provide a meaningful name for the value that must be input by the user. Give a meaningful name for a value that must be output.

1-2 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.

Item	**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. The name `courseGrade` represents the requested information to be output to the user.

This problem becomes more generalized when the user enters values to produce the result. If the program asks the user for data, the program can be used later to compute course grades for many students with any set of grades. So let's decide on and create names for the values that must be input. To determine `courseGrade`, three values are required: `test1`, `test2`, and `finalExam`. The first three analysis activities are now complete:

- Problem understood.
- Information to be output: `courseGrade`.
- Data to be input: `test1`, `test2`, and `finalExam`.

However, a sample problem is still missing. Consider these three values

- `test1` ***74.0***
- `test2` ***79.0***
- `finalExam` ***84.0***
- `courseGrade` ***?***

Sample inputs along with the expected output provide an important benefit−we have an expected result for one set of inputs. In this problem, to create this `courseGrade` problem, we must understand the difference between a simple average and a weighted average. Because the three input items comprise different portions of the final grade (either 25% or 50%), the problem involves computing a weighted average. The simple average of the set 74.0, 79.0, and 84.0 is 79.0; each test is measured equally. However, the weighted average computes differently. Recall that `test1` and `test2` are each worth 25%, and `finalExam` weighs in at 50% of the final grade. When `test1` is 74.0, `test2` is 79.0, and `finalExam` is 84.0, the weighted average computes to 80.25.

```
(0.25 x test1) + (0.25 x test2) + (0.50 x finalExam)
 (0.25 x 74.0) +  (0.25 x 79.0) +   (0.50 x 84.0)
     18.50     +      19.75     +       42.00
                      80.25
```

With the same exact grades, the weighted average of 80.25 is different from the simple average (79.0).

Failure to follow the problem specification could result in students who receive grades lower, or higher, than they actually deserve.

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:

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

1-3 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 future value = present value * (1 + rate)$^{\text{periods}}$.

1.3 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 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

1-4 Cake recipes typically omit a very important activity. Describe an activity that is missing from the algorithm above.

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.

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

Computer programs 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: Input Process Output (IPO)

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 and Indeterminate Loop. 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.

Consider this quote from Christopher Alexander's book A Pattern Language:

> 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.

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.

Three-Step Pattern	**Pattern Applied to a Specific Algorithm**
1. Input	1. Obtain `test1`, `test2`, and `finalExam`
2. Process	2. Compute `courseGrade`
3. Output	3. Display `courseGrade`

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

1-5 Read the three activities of the algorithm above. Do you detect a missing activity?

1-6 Read the three activities of the algorithm above. Do you detect any activity out of order?

1-7 Would this previous algorithm work if the first two activities were switched?

1-8 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 analysis, there is a document with a list of data (variables) and a sample problem.
2. From the design phase there is an algorithm.

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?

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 simulated computer activities.

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

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:

Display the value of `courseGrade`

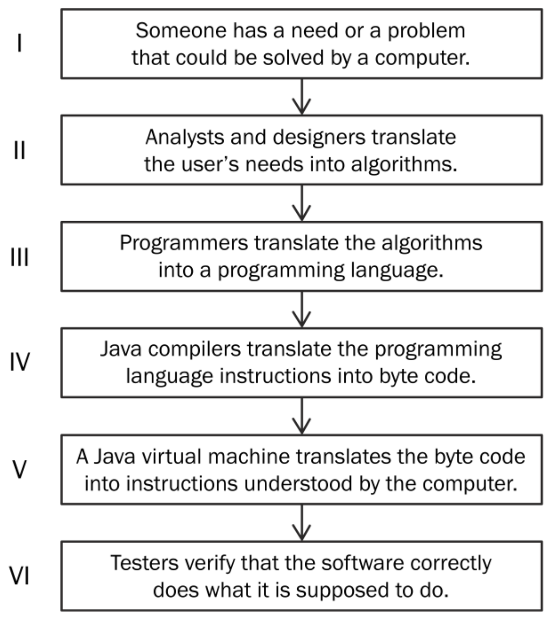
could be translated into Java source code that might look like this:

```
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 your 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 (JVM). The Java virtual machine then converts the byte code into the machine language of that particular machine. In this way, the same Java program can run on a variety of platforms such as Unix, Mac OS, Linux, and 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 as shown to the right (at least this is one opinion/summary).


I
Someone has a need or a problem that could be solved by a computer.
II
Analysts and designers translate the user's needs into algorithms.
III
Programmers translate the algorithms into a programming language.
IV
Java compilers translate the programming language instructions into byte code.
V
A Java virtual machine translates the byte code into instructions understood by the computer.
VI
Testers verify that the software correctly does what it is supposed to do.

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

1-9 Review the above figure 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.

A Preview of a Java 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 in Chapter

2. 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 a `Scanner` (discussed in Chapter 2).

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

public class TestCourseGrade {

  public static void main(String[] args) {
    System.out.println("This program computes a course grade when");
    System.out.println("you have entered three requested values.");

    // I)nput test1, test2, and finalExam.
    Scanner keyboard = new Scanner(System.in);

    System.out.print("Enter first test: ");
    double test1 = keyboard.nextDouble();
    System.out.print("Enter second test: ");
    double test2 = keyboard.nextDouble();
    System.out.print("Enter final exam: ");
    double finalExam = keyboard.nextDouble();

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

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

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%
```

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 experienced the problems incurred by *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?
- In a short time you will see how a newer form of unit testing will help you develop software.

You should have a sample problem before the program is coded—not after. Determine the input values and what you expect for output.

When the Java implementation finally does generate output, the predicted results can then be 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. Testing reduces errors and increases confidence that the program works correctly.

In Chapter 3, you will be introduced to an industry level testing tool that does not require user input. You will be able to build reusable automated tests. In Chapter 2, the program examples will have user input and output that must be compared manually (not automatically).

Self-Check

1-10 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?

1-11 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?

1-12 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?

Answers to Self-Check Questions

1-1 Input: `pounds` and perhaps `todaysConversionRate`, Output: `USDollars`

1-2 `CDCollection`, `currentSelection`

1-3

Problem	Data Name	Input or Output	Sample Problem
Compute the	presentValue	Input	1000.00
future value of	periods	Input	360 (30 years)
an investment	monthlyInterestRate	Input	0.0075 (9%/year)
	futureValue	Output	14730.58

1-4 Turn the oven off (or you might recognize some other activity or detail that was omitted).

1-5 No (at least the author thinks it's okay)

1-6 No (at least the author thinks it's okay)

1-7 No. The `courseGrade` would be computed using undefined values for `test1`, `test2`, and `finalExam`.

1-8 No. The details of the process step are not present. The formula is missing.

1-9 -a I, II, III, and VI
-b IV and V

1-10 The program is wrong.

1-11 The prediction is wrong. The problem asked for a weighted average, not a simple average.

1-12 The program is wrong.