Control Statements: Part 1
Let’s all move one place on.
   – Lewis Carroll

The wheel is come full circle.
   – William Shakespeare, King Lear

How many apples fell on Newton’s head before he took the hint?
   – Robert Frost

All the evolution we know of proceeds from the vague to the definite.
   – Charles Sanders Pierce
OBJECTIVES

In this chapter you will learn:

- Basic problem-solving techniques.
- To develop algorithms through the process of top-down, stepwise refinement.
- To use the If …Then and If …Then Else selection statements to choose among alternative actions.
- To use the While, Do While...Loop and Do Until...Loop repetition statements to execute statements in a program repeatedly.
OBJECTIVES

- To use the compound assignment operators to abbreviate assignment operations.
- To use counter-controlled repetition and sentinel-controlled repetition.
- To use nested control statements.
- To add Visual Basic code to a Windows Forms application.
### Outline

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>5.2</td>
<td>Algorithms</td>
</tr>
<tr>
<td>5.3</td>
<td>Pseudocode</td>
</tr>
<tr>
<td>5.4</td>
<td>Control Structures</td>
</tr>
<tr>
<td>5.5</td>
<td>If ...Then Selection Statement</td>
</tr>
<tr>
<td>5.6</td>
<td>If ...Then...Else Selection Statement</td>
</tr>
<tr>
<td>5.7</td>
<td>While Repetition Statement</td>
</tr>
<tr>
<td>5.8</td>
<td>Do While ...Loop Repetition Statement</td>
</tr>
<tr>
<td>5.9</td>
<td>Do Until ...Loop Repetition Statement</td>
</tr>
</tbody>
</table>
5.10 Compound Assignment Operators
5.11 Formulating Algorithms: Counter-Controlled Repetition
5.12 Formulating Algorithms: Sentinel-Controlled Repetition
5.13 Formulating Algorithms: Nested Control Statements
5.14 Formulating Algorithms: Nested Repetition Statements
5.15 Visual Basic Programming in a Windows Forms Application
5.16 (Optional) Software Engineering Case Study: Identifying Class Attributes in the ATM System
5.2 Algorithms

• Computers solve problems by executing a series of actions in a specific order.

• An algorithm is a **procedure** for solving a problem, in terms of:
  – the **actions** to be executed and
  – the **order** in which these actions are executed
• Consider the “rise-and-shine algorithm” followed by one junior executive for getting out of bed and going to work:

• (1) get out of bed, (2) take off pajamas, (3) take a shower, (4) get dressed, (5) eat breakfast and (6) carpool to work.
5.2 Algorithms (Cont.)

• In a slightly different order:

• (1) get out of bed, (2) take off pajamas, (3) get dressed, (4) take a shower, (5) eat breakfast, (6) carpool to work.

• In this case, our junior executive shows up for work soaking wet.

• The order in which statements execute in a program is called **program control**.
5.3 Pseudocode

- **Pseudocode** helps programmers develop algorithms.
  - It is similar to every-day English; it is convenient and user-friendly.
  - Pseudocode programs are not executed on computers.

- Pseudocode typically does not include variable declarations such as:

  ```vbnet
  Dim number As Integer
  ```

**Software Engineering Observation 5.1**

Pseudocode helps you conceptualize a program during the program-design process. The pseudocode program can be converted to Visual Basic at a later point.
5.4 Control Structures

• In **sequential execution**, statements are executed one after another.

• A **transfer of control** occurs when an executed statement does not directly follow the previously executed statement in the program.

• Early programs relied on **GoTo** statements, which were unstructured.

• Structured programming reduced development times with a clearer and easier format.
5.4 Control Structures (Cont.)

- A UML activity diagram models the workflow of a software system.

- Activity diagrams are composed of symbols:
  - the action state symbol (a rectangle with rounded sides)
  - the diamond symbol
  - the small circle symbol
  - transition arrows, representing the flow
5.4 Control Structures (Cont.)

- The sequence-structure activity diagram in Fig. 5.1 contains two **action states**.
- The **solid circle symbols** represent the activity’s **initial-state** and **final state**.
- Notes are similar to comments

*Fig. 5.1 | Sequence-structure activity diagram.*
5.4 Control Structures (Cont.)

• Visual Basic provides three types of selection statements.
  
  – The `If ..Then` single-selection statement either performs an action or skips the action depending on a condition.
  
  – The `If ..Then..Else` double-selection statement performs an action if a condition is true, and performs a different action if the condition is false.
  
  – The `Select ..Case` double-selection statement performs one of many different actions, depending on the value of an expression.
5.4 Control Structures (Cont.)

Software Engineering Observation 5.2

Any Visual Basic program can be constructed from only 11 different types of control statements (sequence, three types of selection statements and seven types of repetition statements) combined in only two ways (control-statement stacking and control-statement nesting).
5.5 If …Then Selection Statement

*If student’s grade is greater than or equal to 60 then Print “Passed”*

• The preceding pseudocode If statement may be written in Visual Basic as:

```vbnet
If studentGrade >= 60 Then
    Console.WriteLine("Passed")
End If
```

• The statement also could be written as:

```vbnet
If studentGrade >= 60 Then Console.WriteLine("Passed")
```
• **Logic errors** result from unexpected occurrences in the application or problems in the code (such as calling the wrong method).
  
  – A **fatal logic error** causes a program to fail and terminate prematurely.
  
  – A **nonfatal logic error** causes the program to produce incorrect results.
5.5 If ...Then Selection Statement (Cont.)

- Figure 5.2 illustrates the single-selection If ...Then statement.
  - At the diamond decision symbol, the workflow continues along the path determined by the guard conditions.
  - Guard conditions are specified in the square brackets.

**Fig. 5.2** If ...Then single-selection statement activity diagram.
If student’s grade is greater than or equal to 60 then
Print “Passed”
Else
Print “Failed”

• The preceding pseudocode If...Else statement may be written in Visual Basic as

If studentGrade >= 60 Then
    Console.WriteLine("Passed")
Else
    Console.WriteLine("Failed")
End If
Good Programming Practice 5.1

Visual Basic indents both body statements of an If ... Then ... Else statement to improve readability. You should also follow this convention when programming in other languages.

Good Programming Practice 5.2

A standard indentation convention should be applied consistently throughout your programs. It is difficult to read programs that do not use uniform spacing conventions.
5.6 If ...Then...Else Selection Statement (Cont.)

• An If ...Then...Else statement can also be written using a **conditional If expression**:

```csharp
Console.WriteLine(If(studentGrade >= 60, "Passed", "Failed"))
```

• A conditional If expression includes:
  – the condition
  – the value to return if the condition is true
  – the value to return if the condition is false
5.6 If ...Then...Else Selection Statement (Cont.)

- Figure 5.3 illustrates the flow of control in the If ...Then...Else statement.

Fig. 5.3 | If ...Then...Else double-selection statement activity diagram.
Nested If ... Then... Else Statements

• Nested If ... Then... Else statements test for multiple conditions:

If student’s grade is greater than or equal to 90 then
   Print “A”

Else
   If student’s grade is greater than or equal to 80 then
      Print “B”
   Else
      If student’s grade is greater than or equal to 70 then
         Print “C”
      Else
         If student’s grade is greater than or equal to 60 then
            Print “D”
         Else
            Print “F”
The preceding pseudocode may be written in Visual Basic as

```vbnet
If studentGrade >= 90 Then
    Console.WriteLine("A")
Else
    If studentGrade >= 80 Then
        Console.WriteLine("B")
    Else
        If studentGrade >= 70 Then
            Console.WriteLine("C")
        Else
            If studentGrade >= 60 Then
                Console.WriteLine("D")
            Else
                Console.WriteLine("F")
            End If
        End If
    End If
End If
End If
End If
```
Good Programming Practice 5.3

If there are several levels of indentation, each level should be indented additionally by the same amount of space; this gives programs a neatly structured appearance.
5.6 If ...Then...Else Selection Statement
(Cont.)

• Most Visual Basic programmers prefer to use the **ElseIf** keyword:

```vbnet
If grade >= 90 Then
    Console.WriteLine("A")
ElseIf grade >= 80 Then
    Console.WriteLine("B")
ElseIf grade >= 70 Then
    Console.WriteLine("C")
ElseIf grade >= 60 Then
    Console.WriteLine("D")
Else
    Console.WriteLine("F")
End If
```
A repetition statement repeats an action, depending on the loop-continuation condition.

While there are more items on my shopping list
  Put next item in cart
  Cross it off my list

When the condition becomes false, the first statement after the repetition statement executes.
Consider a program designed to find the first power of 3 larger than 100 (Fig. 5.4).

```
' Fig. 5.4: PowersOfThree.vb
' Demonstration of While statement.
Module PowersOfThree
    Sub Main()
        Dim product As Integer = 3
        ' statement multiplies and displays product
        ' while product is less than or equal to 100
        While product <= 100
            Console.Write(product & " ")
            product = product * 3 ' compute next power of 3
        End While
    End Sub
End Module
```

**Fig. 5.4** | While repetition statement used to print powers of 3. (Part 1 of 2.)
Console.WriteLine() ' write blank line
' print result
Console.WriteLine("First power of 3 " & _
  "larger than 100 is " & product)
End Sub ' Main
End Module ' PowersOfThree

3  9  27  81
First power of 3 larger than 100 is 243

Fig. 5.4 | While repetition statement used to print powers of 3. (Part 2 of 2.)
5.7 \textbf{While} Repetition Statement (Cont.)

- The UML activity diagram (Fig. 5.5) illustrates the flow of control in a \textbf{While} statement.
- The diamond \textit{merge symbol} joins two flows of activity into one.

\textbf{Fig. 5.5} | \textbf{While} repetition statement activity diagram.
Common Programming Error 5.1

Failure to provide the body of a While statement with an action that eventually causes the loop-continuation condition to become false is a logic error. Such a repetition statement never terminates, resulting in a logic error called an “infinite loop.”
The **Do While...Loop** repetition statement behaves exactly like the **While** repetition statement (Fig. 5.6).

```vbnet
' Fig. 5.6: DoWhile.vb
'Demonstration of the Do While...Loop statement.
Module DoWhile
  Sub Main()
    Dim product As Integer = 3 ' initialize product
    ' statement multiplies and displays the
    ' product while it is less than or equal to 100
    Do While product <= 100
      Console.Write(product & "  ")
      product = product * 3
    Loop
  End Sub
End Module
```

**Fig. 5.6** | **Do While...Loop** repetition statement demonstration. (Part 1 of 2.)
```vbnet
13 Console.WriteLine() ' write blank line
15 ' print result
16 Console.WriteLine("First power of 3 " & _
17     "larger than 100 is " & product)
19 End Sub ' Main
20 End Module ' DoWhile
```

First power of 3 larger than 100 is 243

**Fig. 5.6** | Do While…Loop repetition statement demonstration. (Part 2 of 2.)
Common Programming Error 5.2

Failure to provide the body of a Do While…Loop statement with an action that eventually causes the loop-continuation condition to become false creates an infinite loop.
A **Do Until** ...**Loop** is executed repeatedly as long as the **loop-termination condition** is false (Fig. 5.7).

```vbnet
Module DoUntil

    Sub Main()
        Dim product As Integer = 3

        ' find first power of 3 larger than 100
        Do Until product > 100
            Console.WriteLine(product & " ")
            product = product * 3
        Loop
    End Sub

End Module
```

**Fig. 5.7** | Do Until ...Loop repetition statement demonstration. (Part 1 of 2.)
Common Programming Error 5.3

Failure to provide the body of a `Do Until` ... `Loop` statement with an action that eventually causes the loop-termination condition to become true creates an infinite loop.
## 5.10 Compound Assignment Operators

<table>
<thead>
<tr>
<th>Compound assignment operator</th>
<th>Sample expression</th>
<th>Explanation</th>
<th>Assigns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+=</code></td>
<td><code>c += 7</code></td>
<td><code>c = c + 7</code></td>
<td><code>11 to c</code></td>
</tr>
<tr>
<td><code>-=</code></td>
<td><code>c -= 3</code></td>
<td><code>c = c - 3</code></td>
<td><code>1 to c</code></td>
</tr>
<tr>
<td><code>*=</code></td>
<td><code>c *= 4</code></td>
<td><code>c = c * 4</code></td>
<td><code>16 to c</code></td>
</tr>
<tr>
<td><code>/=</code></td>
<td><code>c /= 2</code></td>
<td><code>c = c / 2</code></td>
<td><code>2 to c</code></td>
</tr>
<tr>
<td><code>\=</code></td>
<td><code>c \= 3</code></td>
<td><code>c = c \ 3</code></td>
<td><code>1 to c</code></td>
</tr>
<tr>
<td><code>^=</code></td>
<td><code>c ^= 2</code></td>
<td><code>c = c ^ 2</code></td>
<td><code>16 to c</code></td>
</tr>
<tr>
<td><code>&amp;=</code></td>
<td><code>d &amp;= &quot;Hello&quot;</code></td>
<td><code>d = d &amp; &quot;Hello&quot;</code></td>
<td>&quot;Hello&quot; to d</td>
</tr>
</tbody>
</table>

Assume: `c = 4, d = "He"`

**Fig. 5.8 |** Compound assignment operators.
Figure 5.9 calculates a power of two using the exponentiation assignment operator.

```vbnet
' Fig. 5.9: Assignment.vb
' Using a compound assignment operator to calculate a power of 2.
Module Assignment
Sub Main()
  Dim exponent As Integer ' power input by user
  Dim result As Integer = 2 ' number to raise to a power

  Console.Write("Enter an integer exponent: ")
  exponent = Console.ReadLine() ' input exponent
```

**Fig. 5.9** | Exponentiation using a compound assignment operator. (Part 1 of 2.)
Fig. 5.9 | Exponentiation using a compound assignment operator. (Part 2 of 2.)
5.11 Formulating Algorithms: Counter-Controlled Repetition

• Consider the following problem statement:

A class of 10 students took a quiz. The grades for this quiz are made available to you. Determine the class average on the quiz.

• The algorithm for solving this problem on a computer must input each grade, keep track of the total of all grades input, perform the averaging calculation and print the result.
5.11 Formulating Algorithms: Counter-Controlled Repetition (Cont.)

Pseudocode Algorithm with Counter-Controlled Repetition

- **Counter-controlled repetition** (Fig. 5.10) inputs and processes the grades one at a time.

```plaintext
1 Set total to zero
2 Set grade counter to one
3
4 While grade counter is less than or equal to ten
5   Prompt the user to enter the next grade
6   Input the next grade
7   Add the grade into the total
8   Add one to the grade counter
9
10 Set the class average to the total divided by ten
11 Print the class average
```

**Fig. 5.10** | Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.
The pseudocode is now converted to Visual Basic (Fig. 5.11).

```vbnet
Fig. 5.11: GradeBook.vb

GradeBook.vb

' Fig. 5.11: GradeBook.vb
' GradeBook class that solves class-average problem using
' counter-controlled repetition.
Public Class GradeBook
    Private courseNameValue As String ' course name for this GradeBook

    ' constructor initializes CourseName with String supplied as argument
    Public Sub New(ByVal name As String)
        CourseName = name ' validate and store course name
    End Sub ' New

    ' property that gets and sets the course name; the Set accessor
    ' ensures that the course name has at most 25 characters
    Public Property CourseName() As String
        Get ' retrieve courseNameValue
            Return courseNameValue
        End Get
    End Property

Fig. 5.11 | Counter-controlled repetition: Class-average problem. (Part 1 of 4.)
```
Set(ByVal value As String) ' set courseNameValue
    If value.Length <= 25 Then ' if value has 25 or fewer characters
        courseNameValue = value ' store the course name in the object
    Else ' if name has more than 25 characters
        ' set courseNameValue to first 25 characters of parameter name
        ' start at 0, length of 25
        courseNameValue = value.Substring(0, 25)
    End If
End Set
End Property ' CourseName

Console.WriteLine(_
    "Course name (" & value & ") exceeds maxLength (25)."
) Console.WriteLine(_
    "Limiting course name to first 25 characters." & vbCrLf)
End If
End Property ' CourseName

Fig. 5.11 | Counter-controlled repetition: Class-average problem. (Part 2 of 4.)
Public Sub DisplayMessage()
    ' this statement uses property CourseName to get the
    ' name of the course this GradeBook represents
    Console.WriteLine("Welcome to the grade book for ",
        & vbCrLf & CourseName & "!")
End Sub ' DisplayMessage

Public Sub DetermineClassAverage()
    Dim total As Integer ' sum of grades entered by user
    Dim gradeCounter As Integer ' number of grades input
    Dim grade As Integer ' grade input by user
    Dim average As Integer ' average of grades

    ' initialization phase
    total = 0 ' set total to zero
    gradeCounter = 1 ' prepare to loop

    Counter initialized before the repetition statement.

Fig. 5.11 | Counter-controlled repetition: Class-average problem. (Part 3 of 4.)
While gradeCounter <= 10 ' loop 10 times
  ' prompt for and input grade from user
  Console.Write("Enter grade: ") ' prompt for grade
  grade = Console.ReadLine() ' input the next grade
  total += grade ' add grade to total
  gradeCounter += 1 ' add 1 to gradeCounter
End While

' termination phase
average = total \ 10 ' integer division yields integer result

' display total and average of grades
Console.WriteLine(vbCrLf & "Total of all 10 grades is " & total)
Console.WriteLine("Class average is " & average)
End Sub ' DetermineClassAverage
End Class ' GradeBook

Fig. 5.11 | Counter-controlled repetition: Class-average problem. (Part 4 of 4.)
Good Programming Practice 5.4

Although Visual Basic initializes numeric variables to 0, it is a good practice to initialize certain variables explicitly to avoid confusion and improve program readability.

Good Programming Practice 5.5

Separating declarations from other statements with a blank line improves readability.
Software Engineering Observation 5.3

The most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, the process of producing a working Visual Basic program from the algorithm is normally straightforward.
Module GradeBookTest (Fig. 5.12) creates an object of class GradeBook and demonstrates its capabilities.

```vbnet
' Fig. 5.12: GradeBookTest.vb
Create GradeBook object and invoke its DetermineClassAverage method.
Module GradeBookTest
    Sub Main()
        ' create GradeBook object gradeBook and
        ' pass course name to constructor
        Dim gradeBook As New GradeBook("CS101 Introduction to VB")

        gradeBook.DisplayMessage() ' display welcome message
        gradeBook.DetermineClassAverage() ' find average of 10 grades
    End Sub ' Main
End Module ' GradeBookTest
```

Fig. 5.12 | Module GradeBookTest creates an object of class GradeBook and invokes its DetermineClassAverage method. (Part 1 of 2.)
Welcome to the grade book for CS101 Introduction to VB!

Enter grade: 65
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 844
Class average is 84

Fig. 5.12 | Module GradeBookTest creates an object of class GradeBook and invokes its DetermineClassAverage method. (Part 2 of 2.)

Common Programming Error 5.4

Assuming that integer division rounds (rather than truncates) can lead to incorrect results. For example, 7 divided by 4, which yields 1.75 in conventional arithmetic, truncates to 1 in integer arithmetic, rather than rounding to 2.
5.12 Formulating Algorithms: Sentinel-Controlled Repetition

• Consider the following problem:

   Develop a class-averaging program that processes grades for an arbitrary number of students each time it is run.

• A sentinel value can be used to indicate when no more grades will be entered.

• Sentinel-controlled repetition is called indefinite repetition because the number of repetitions is not known before the loop begins its execution.

Common Programming Error 5.5

Choosing a sentinel value that is also a legitimate data value could result in a logic error that would cause a program to produce incorrect results.
5.12 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

• Approach the class-average program with top-down, stepwise refinement.

• Begin with a single statement that conveys the overall function of the program:

Determine the class average for the quiz

• Divide into a series of smaller tasks that are listed in the order in which they must be performed:

Initialize variables

Input, sum and count the quiz grades

Calculate and print the class average
5.12 Formulating Algorithms: Sentinel-Controlled Repetition

Software Engineering Observation 5.4

Each refinement, including the top, is a complete specification of the algorithm; only the level of detail in each refinement varies.

Software Engineering Observation 5.5

Many algorithms can be divided logically into three phases—an initialization phase that initializes the program variables, a processing phase that inputs data values and adjusts program variables accordingly, and a termination phase that calculates and prints the results.
5.12 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- In the second refinement, we commit to specific variables:
  
  Initialize total to zero  
  Initialize grade counter to zero

- The algorithm requires a repetition statement that inputs and processes each grade:
  
  Prompt the user to enter the first grade  
  Input the first grade (possibly the sentinel)  
  While the user has not yet entered the sentinel  
      Add this grade into the running total  
      Add one to the grade counter  
      Prompt the user to enter the next grade  
      Input the next grade (possibly the sentinel)
5.12 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

*Calculate and print the class average*

- The preceding pseudocode statement can be refined as follows:

  *If the grade counter is not equal to zero then*
  
  *Set the average to the total divided by the counter*
  
  *Print the class average*

*Else*

  *Print “No grades were entered”*

- An *If* statement is used to avoid division by zero.
5.12 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

• The complete second refinement is shown in Fig. 5.13.

```
1. Initialize total to zero
2. Initialize grade counter to zero

4. Prompt the user to enter the first grade
5. Input the first grade (possibly the sentinel)

7. While the user has not yet entered the sentinel
   8. Add this grade into the running total
   9. Add one to the grade counter
10. Prompt the user to enter the next grade
11. Input the next grade (possibly the sentinel)

13. If the grade counter is not equal to zero then
    14. Set the average to the total divided by the counter
    15. Print the class average
16. Else
    17. Print “No grades were entered”
```

**Fig. 5.13** | Class-average problem pseudocode algorithm with sentinel-controlled repetition.
Error-Prevention Tip 5.1

When performing division by an expression whose value could be zero, explicitly test for this case and handle it appropriately in your program. Such handling could be as simple as printing an error message.

Good Programming Practice 5.6

Include blank lines in pseudocode algorithms to improve readability. The blank lines separate pseudocode control statements and the algorithms’ phases.

Software Engineering Observation 5.6

You terminate the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for the pseudocode to be converted to a Visual Basic program.
• Class **GradeBook** (Fig. 5.14) contains the method **DetermineClassAverage**, which implements the algorithm.

```
' Fig. 5.14: GradeBook.vb
' GradeBook class that solves class-average problem using
' sentinel-controlled repetition.
Public Class GradeBook
    Private courseNameValue As String ' course name for this GradeBook

    ' constructor initializes CourseName with String supplied as argument
    Public Sub New(ByVal name As String)
        CourseName = name ' validate and store course name
    End Sub ' New

    ' property that gets and sets the course name; the Set accessor
    ' ensures that the course name has at most 25 characters
    Public Property CourseName() As String
        Get ' retrieve courseNameValue
            Return courseNameValue
        End Get

Fig. 5.14 | Sentinel-controlled repetition: Class-average problem. (Part 1 of 5.)
Set (ByVal value As String) ' set courseNameValue
    If value.Length <= 25 Then ' if value has 25 or fewer characters
        courseNameValue = value ' store the course name in the object
    Else ' if name has more than 25 characters
        ' set courseNameValue to first 25 characters of parameter name
        ' start at 0, length of 25
        courseNameValue = value.Substring(0, 25)
    Console.WriteLine( _
        "Course name (" & value & ") exceeds maximum length (25)."")
    Console.WriteLine( _
        "Limiting course name to first 25 characters." & vbCrLf)
End If
End Set
End Property ' CourseName

Fig. 5.14 | Sentinel-controlled repetition: Class-average problem. (Part 2 of 5.)
Public Sub DisplayMessage()
    ' this statement uses property CourseName to get the
    ' name of the course this GradeBook represents
    Console.WriteLine("Welcome to the grade book for " +
        _
        & vbCrLf & CourseName & "!" & vbCrLf)
End Sub ' DisplayMessage

Public Sub DetermineClassAverage()
    Dim total As Integer ' sum of grades entered by user
    Dim gradeCounter As Integer ' number of grades input
    Dim grade As Integer ' grade input by user
    Dim average As Double ' average of all grades

    ' initialization phase
    total = 0 ' clear total
    gradeCounter = 0 ' prepare to loop

Fig. 5.14 | Sentinel-controlled repetition: Class-average problem. (Part 3 of 5.)
' processing phase
' prompt for input and read grade from user
Console.Write("Enter grade or -1 to quit: ")
grade = Console.ReadLine()

' sentinel-controlled loop where -1 is the sentinel value
While grade <> -1
    total += grade ' add grade to total
    gradeCounter += 1 ' add 1 to gradeCounter
End While

Fig. 5.14 | Sentinel-controlled repetition: Class-average problem. (Part 4 of 5.)
If gradeCounter <> 0 Then ' if user entered at least one grade
    ' calculate average of all grades entered
    average = total / gradeCounter
    ' display total and average (with two digits of precision)
    Console.WriteLine(vbCrLf & "Total of the " & gradeCounter & _
                      " grades entered is " & total)
    Console.WriteLine("Class average is {0:F}", average)
Else ' no grades were entered, so output appropriate message
    Console.WriteLine("No grades were entered")
End If

' termination phase

End Sub ' DetermineClassAverage

End Class ' GradeBook

"{0: F}" indicates that the value should be displayed as a fixed-point number.

Fig. 5.14 | Sentinel-controlled repetition: Class-average problem. (Part 5 of 5.)
5.12 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

Good Programming Practice 5.7

In a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user of the sentinel value.
5.12 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

Floating-Point Number Precision and Memory Requirements

- Variables of type `Single` represent single-precision floating-point numbers and have seven significant digits.
- Variables of type `Double` represent double-precision floating-point numbers.

Floating-Point Numbers are Approximations

- Due to the imprecise nature of floating-point numbers, type `Double` is preferred over type `Single` because `Double` variables can represent floating-point numbers more accurately.
Common Programming Error 5.6

Using floating-point numbers in a manner that assumes they are represented precisely can lead to logic errors.
Implicitly Converting Between Primitive Types

• The floating-point division operator operates on Single, Double and Decimal values.
  – To ensure that the operands are one of these types, Visual Basic performs implicit conversion.
  – If both operands are of type Integer, the operands will be promoted to Double values.
• “{0: F}” indicates that the value should be displayed as a fixed-point number.

• This is an example of formatted output of a value’s data.

• The numeric value that appears before the colon indicates which argument will be formatted.

• The value after the colon is known as a format specifier.
Some common format specifiers are summarized in Fig. 5.15.

<table>
<thead>
<tr>
<th>Format Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Currency. Formats the currency based on the computer’s locale setting.</td>
</tr>
<tr>
<td>E</td>
<td>Scientific notation. For example, 956. 2 is formatted as 9. 562000E+002.</td>
</tr>
<tr>
<td>F</td>
<td>Fixed point. Sets the number of decimal places to two.</td>
</tr>
<tr>
<td>G</td>
<td>General. Visual Basic chooses either E or F for you, depending on which representation generates a shorter string.</td>
</tr>
<tr>
<td>D</td>
<td>Decimal integer. Displays an integer as a whole number in standard base-10 format.</td>
</tr>
<tr>
<td>N</td>
<td>Number. Separates every three digits with a comma and sets the number of decimal places to two. (Varies by locale.)</td>
</tr>
<tr>
<td>X</td>
<td>Hexadecimal integer. Displays the integer in hexadecimal (base-16) notation.</td>
</tr>
</tbody>
</table>

**Fig. 5.15** | Formatting codes for Strings.
5.12 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

Good Programming Practice 5.8

When formatting with two positions to the right of the decimal point, some programmers prefer to use the format specifier F2 for clarity.
• The three grades entered during the sample execution of module `GradeBookTest` (Fig. 5.16) total 257, which yields the rounded average 85.67.

```vbnet
' Fig. 5.16: GradeBookTest.vb
' Create GradeBook object and invoke its DetermineClassAverage method.
Module GradeBookTest
    Sub Main()
        ' create GradeBook object gradeBook and
        ' pass course name to constructor
        Dim gradeBook As New GradeBook("CS101 Introduction to VB")
        gradeBook.DisplayMessage() ' display welcome message
        gradeBook.DetermineClassAverage() ' find average of grades
    End Sub ' Main
End Module ' GradeBookTest
```

**Fig. 5.16 |** `GradeBookTest` module creates an object of class `GradeBook` (Fig. 5.14) and invokes its `DetermineClassAverage` method. (Part 1 of 2.)
Welcome to the grade book for CS101 Introduction to VB!

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67

Fig. 5.16 | GradeBookTest module creates an object of class GradeBook (Fig. 5.14) and invokes its DetermineClassAverage method. (Part 2 of 2.)
• Consider the following problem statement:

A college offers a course that prepares students for the state licensing exam for real estate brokers. The college wants to know how well its students did on the exam. You have been given a list of 10 students.

1. Tally each exam result ("P" or "F").
2. Display a summary of the exam results.
3. If more than eight students passed the exam, print the message "Raise tuition."
5.13 Formulating Algorithms: Nested Control Statements (Cont.)

- Proceed with top-down, stepwise refinement.

\textit{Analyze exam results and decide if tuition should be raised}

- Our first refinement is

\textit{Initialize variables}

\textit{Input the ten exam grades, and count passes and failures}

\textit{Print a summary of the exam results and decide if tuition should be raised}

- Only the counters for the number of passes, number of failures and number of students need to be initialized.

\textit{Initialize passes to zero}

\textit{Initialize failures to zero}

\textit{Initialize student to one}
Input the 10 exam results, and count passes and failures

- A loop successively inputs the result of each exam.

While student is less than or equal to 10
  Prompt the user to enter the next exam result
  Input the next exam result

If the student passed then
  Add one to passes
Else
  Add one to failures
Add one to student
Print a summary of the exam results and decide if tuition should be raised

• The preceding pseudocode statement may be refined as follows:

Print the number of passes
Print the number of failures

If more than eight students passed then
   Print “Raise tuition”
Complete Second Refinement of Pseudocode and Conversion to Class Analysis

• The complete second refinement of the pseudocode appears in Fig. 5.17.
5.13 Formulating Algorithms: Nested Control Statements (Cont.)

Fig. 5.17 | Pseudocode for examination-results problem.

```plaintext
1 Initialize passes to zero
2 Initialize failures to zero
3 Initialize student to one

While student is less than or equal to 10
   Prompt the user to enter the next exam result
   Input the next exam result

   If the student passed then
      Add one to passes
   Else
      Add one to failures
   Add one to student

Print the number of passes
Print the number of failures

If more than eight students passed then
   Print “Raise tuition”
```
The pseudocode is now converted into Visual Basic (Fig. 5.18).

Fig. 5.18 | Nested control statements: Examination-results problem. (Part 1 of 2.)
' nested control statement
If result = "P" Then
    passes += 1 ' increment number of passes
Else
    failures += 1 ' increment number of failures
End If

student += 1 ' increment student counter
End While

' display exam results
Console.WriteLine( _
    "Passed: " & passes & vbCrLf & "Failed: " & failures)

' raise tuition if more than 8 students passed
If passes > 8 Then
    Console.WriteLine("Raise tuition")
End If

End Sub ' ProcessExamResults

End Class ' Analysis

Fig. 5.18 | Nested control statements: Examination-results problem. (Part 2 of 2.)
Module `AnalysisTest` (Fig. 5.19) tests an `Analysis` object.

```vbnet
', Fig. 5.19: AnalysisTest.vb
2 ' Test program for class Analysis.
3 Module AnalysisTest
4 Sub Main()
5 Dim application As New Analysis() ' create Analysis object
6   application.ProcessExamResults() ' call method to process results
7 End Sub ' Main
8 End Module ' AnalysisTest

Enter result (P = pass, F = fail): P
Enter result (P = pass, F = fail): F
Enter result (P = pass, F = fail): P
Enter result (P = pass, F = fail): P
Enter result (P = pass, F = fail): P

(continued on next page...)

Fig. 5.19 | Test program for class `Analysis` (Part 1 of 2.)
Enter result (P = pass, F = fail): P
Enter result (P = pass, F = fail): P
Enter result (P = pass, F = fail): P
Enter result (P = pass, F = fail): P
Enter result (P = pass, F = fail): P
Passed: 9
Failed: 1
Raise tuition

Enter result (P = pass, F = fail): P
Enter result (P = pass, F = fail): P
Enter result (P = pass, F = fail): P
Enter result (P = pass, F = fail): F
Enter result (P = pass, F = fail): F
Enter result (P = pass, F = fail): P
Enter result (P = pass, F = fail): P
Enter result (P = pass, F = fail): P
Enter result (P = pass, F = fail): P
Passed: 8
Failed: 2

Fig. 5.19 | Test program for class Analysis. (Part 2 of 2.)
5.14 Formulating Algorithms: Nested Repetition Statements

• Consider the following problem statement:

Write a console application that draws a filled square consisting solely of one type of character. The side of the square and the fill character should be entered by the user. The length of the side should not exceed 20 characters.
5.14 Formulating Algorithms: Nested Repetition Statements (Cont.)

**Evolving the Pseudocode**

*Draw a square of fill characters*

- Our first refinement is

  *Prompt for the fill character*
  *Input the fill character*

  *Prompt for the side of the square*
  *Input the side of the square*

  *Draw the square if its side is less than or equal to 20; otherwise print an error message*
5.14 Formulating Algorithms: Nested Repetition Statements (Cont.)

Draw the square

• This pseudocode statement can be implemented by using one loop nested inside another.

Set row to one

While row is less than or equal to side

Set column to one

While column is less than or equal to side

Print the fill character

Increment column by one

Print a line feed/carriage return

Increment row by one
The complete second refinement appears in Fig. 5.20.

```plaintext
1  Prompt for the fill character
2  Input the fill character
3
4  Prompt for the side of the square
5  Input the side of the square
6
7  If the side of the square is less than or equal to 20 then
8       Set row to one
9
10     While row is less than or equal to side
11        Set column to one
12
13        While column is less than or equal to side
14           Print the fill character
15           Increment column by one
16
17           Print a line feed/carriage return
18           Increment row by one
19  Else
20     Print “Side is too large”
```

**Fig. 5.20** Second refinement of the pseudocode.
Many experienced programmers write programs without ever using program-development tools like pseudocode. They feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays producing final outputs. Although this might work for simple and familiar problems, it can lead to serious errors and delays in large, complex projects.
• The pseudocode is converted into Visual Basic (Fig. 5.21).

```vbnet
' Fig. 5.21: Box.vb
' Class can be used to draw a square of a specified length, using
' a specified fill character.
Public Class Box
    Private sideValue As Integer ' length of side of square
    Private fillCharacterValue As String ' character used to draw square

    ' property provides access to side length of box
    Public Property Side() As Integer
        Get
            Return sideValue ' return side length
        End Get

        Set(ByVal value As Integer)
            sideValue = value ' modify side length
        End Set
    End Property ' Side
End Class

Fig. 5.21 | Nested repetition statements used to print a square of symbols. (Part 1 of 3.)
Fig. 5.21 | Nested repetition statements used to print a square of symbols. (Part 2 of 3.)
Fig. 5.21 | Nested repetition statements used to print a square of symbols. (Part 3 of 3.)
• The **BoxTest** module (Fig. 5.22) uses the **Box** class.

```vbnet
' Fig. 5.22: BoxTest.vb
' Program draws square by creating an object of class Box.
Module BoxTest
    ' Main begins program execution
    Sub Main()
        Dim box As New Box()

        ' obtain fill character and side length from user
        Console.Write("Enter fill character: ")
        box.FillCharacter = Console.ReadLine()
        Console.Write("Enter side length (must be 20 or less): ")
        box.Side = Console.ReadLine()

        box.Display() ' display box
    End Sub ' Main
End Module ' BoxTest
```

**Fig. 5.22** | Using class Box to draw a square. (Part 1 of 2.)
Enter fill character: #
Enter side length (must be 20 or less): 8
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #

Enter fill character: *
Enter side length (must be 20 or less): 5
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *

Enter fill character: $
Enter side length (must be 20 or less): 37
Side too large

Fig. 5.22 | Using class Box to draw a square. (Part 2 of 2.)
5.15 Visual Basic Programming in a Windows Forms Application

- Open the **ASimpleProgram** project (Fig. 5.23).
- Change the name of the **Label** to **welcomeLabel** and the name of the **PictureBox** to **bugPictureBox**.

![Fig. 5.23](image) IDE showing program code for **ASimpleProgram.vb**.
5.15 Visual Basic Programming in a Windows Forms Application (Cont.)

• The IDE generates the Visual Basic code that defines how the GUI appears.
  
  – Go into the **Solution Explorer** and click the **Show All Files** button ( ).
  
  – Click the plus sign to the left of **ASimpleProgramForm.vb** to expand its node, and double click **ASimpleProgramForm.Designer.vb**.
5.15 Visual Basic Programming in a Windows Forms Application (Cont.)

- Keyword **Inherits** indicates that the class uses existing pieces from another class.
  - **Form** is the **base class** and **ASimpleProgramForm** is the **derived class**.
- **Form** provides the capabilities your application needs to appear as a window.

**Fig. 5.24** | Windows Form Designer generated code.
When a control is placed on the Form, the IDE adds code to the designer file.

This code (Fig. 5.25) specifies the property values for *WelcomeLabel*.

**Fig. 5.25** Property initializations generated by the Windows Form Designer for *WelcomeLabel*.
Modifying Properties in Design View

• The values assigned to the properties are based on the values in the Properties window.

• Select the welcomeLabel control and change its Text property to “Deitel and Associates” (Fig. 5.26).

Fig. 5.26 | Properties window used to set a property value.
5.15 Visual Basic Programming in a Windows Forms Application (Cont.)

- Return to the Designer code (Fig. 5.27).
- Note that the Label’s Text property is now assigned the text that you entered in the Properties window.

![Screen Shot of Designer code]

**Fig. 5.27** | Windows Form Designer–generated code reflecting new property value.
• Double click the Form's background in design view (Fig. 5.28).

• **Event handlers** such as this method are called in response to **events**.

  – In this case, loading the Form occurs when the application begins.

![Visual Basic Code](image)

**Fig. 5.28** | Method ASimpleProgramForm_Load created when Form is double clicked.
5.15 Visual Basic Programming in a Windows Forms Application (Cont.)

- Add the statement `welcomeLabel.Text = "Visual Basic!"` to the method (Fig. 5.29).

Fig. 5.29 | Method `ASimpleProgramForm_Load` containing program code.
5.15 Visual Basic Programming in a Windows Forms Application (Cont.)

- The property value is only changed when the method is called at runtime.
- Run the program (Fig. 5.30).

Fig. 5.30 | Changing a property value at runtime.
5.16 Software Engineering Case Study: Identifying Class Attributes in the ATM System

Identifying Attributes

- For each descriptive word in the requirements document that plays a significant role, we create an attribute (Fig. 5.31).

<table>
<thead>
<tr>
<th>Class</th>
<th>Descriptive words and phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td>user is authenticated</td>
</tr>
<tr>
<td>Balance Inquiry</td>
<td>account number</td>
</tr>
<tr>
<td>Withdraw</td>
<td>account number</td>
</tr>
<tr>
<td>account number</td>
<td>account number</td>
</tr>
<tr>
<td>Deposit</td>
<td>account number</td>
</tr>
</tbody>
</table>

Fig. 5.31 Descriptive words and phrases from the ATM requirements document. (Part 1 of 2.)
### 5.16 Software Engineering Case Study: Identifying Class Attributes in the ATM System (Cont.)

<table>
<thead>
<tr>
<th>Class</th>
<th>Descriptive words and phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>BankDatabase</td>
<td>[no descriptive words or phrases]</td>
</tr>
<tr>
<td>Account</td>
<td>Account number, PIN, balance</td>
</tr>
<tr>
<td>Screen</td>
<td>[no descriptive words or phrases]</td>
</tr>
<tr>
<td>Keypad</td>
<td>[no descriptive words or phrases]</td>
</tr>
<tr>
<td>CashDispenser</td>
<td>begins each day loaded with 500 $20 bills</td>
</tr>
<tr>
<td>DepositSlot</td>
<td>[no descriptive words or phrases]</td>
</tr>
</tbody>
</table>

**Fig. 5.31** | Descriptive words and phrases from the ATM requirements document. (Part 2 of 2.)
• Class ATM maintains information about the state of the ATM. The phrase “user is authenticated” describes a state of the ATM, so we include userAuthenticated as a Boolean attribute.

• Classes BalanceInquiry, Withdrawal and Deposit involve an “account number”.

• Balances and other monetary amounts should be represented by Decimal variables.
5.16 Software Engineering Case Study: Identifying Class Attributes in the ATM System (Cont.)

**Modeling Attributes**

![Diagram of classes with attributes]

**Fig. 5.32** | Classes with attributes.
5.16 Software Engineering Case Study: Identifying Class Attributes in the ATM System (Cont.)

userAuthentic : Boolean = False

- The attribute name is userAuthentic.
- The attribute type is Boolean.
- The initial value is False.

Software Engineering Observation 5.8

Early in the design process classes often lack attributes (and operations). Such classes should not necessarily be eliminated, however, because attributes (and operations) may become evident in the later phases of design and implementation.