CHAPTER 10

Automation Using Visual Basic Applications (VBA)

A powerful capability of all Microsoft Office applications is the ability to program actions using the Visual Basic Applications (VBA) programming language. For Microsoft Excel, VBA is especially useful for frequently used commands that require multiple procedures, repetitive actions, and in more advanced situations for calculations that exceed the spreadsheet's processing ability. Examples relevant to cash flow modeling include creating buttons that control print commands and goal seek functions, building a quick system to run multiple scenarios, or constructing an amortization engine that can generate and aggregate the cash flow for thousands of loans. Implementing such functionality requires a basic understanding of the VBA language and how the language interacts with Excel.

Most users have unknowingly used VBA by recording a macro to complete simple repetitive tasks. However, few take the step to learn how to write and edit VBA code by hand. The problem most users have with unlocking the full potential of VBA is learning how an object-oriented programming (OOP) language works. While entire books can and have been written on using VBA, this chapter intends to introduce the model operator to the basics of using VBA through additions to Project Model Builder and other relevant examples. Beginners may find additional texts helpful for further explanation, while intermediate to advanced users may want to skip to the specific code examples.

CONVENTIONS OF THIS CHAPTER

One of the most useful means of explaining VBA is through the use of code examples shown in this chapter. Recall that this is also the convention for named cells or ranges. Distinguishing between the two should not be difficult since the VBA code is typically written in blocks of text, while the named cells or ranges are written into the normal text.

THE VISUAL BASIC EDITOR

Coding in VBA is simplified by an interface called the Visual Basic Editor (VBE). VBA code can be written, stored, run, and debugged from the VBE. To access the VBE go to Tools, Macro, Visual Basic Editor or use the ALT-F11 keyboard shortcut. The VBE will open in a separate window and should appear as in Figure 10.1.

The Menu Bar

The general menu bar features recognizable commands such as File and Edit; however, most of the options within each command will seem strange to a new user. For now look at the Standard Toolbar that should appear as a default setting.

The Standard Toolbar has a few buttons that will be useful for the basic operation of the VBE. Keep in mind the following buttons, which can be seen in Figure 10.2:

- View Microsoft Excel: Jumps back to the Excel workbook
- Run Sub/UserForm: Runs the code currently selected
- Break: Breaks the code currently being run
- Reset: Resets the code after a break has occurred
- Object Browser: opens the library of VBA objects

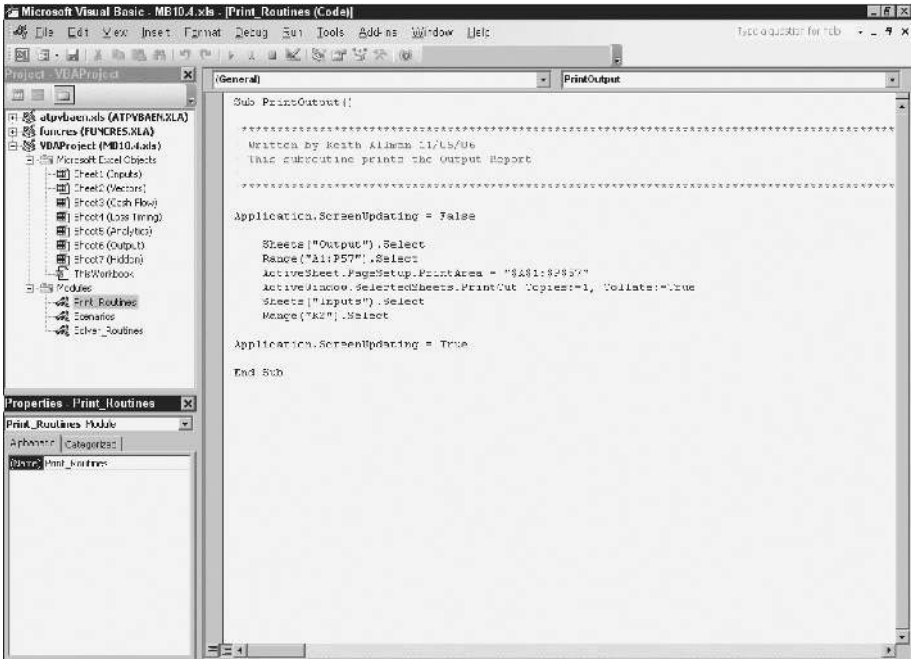

FIGURE 10.1 The Visual Basic Editor (VBE).

FIGURE 10.2 The Standard Toolbar looks similar to other Office Toolbars, but has unique buttons specifically for creating, editing, and running code.

The Project Explorer and the Properties Window

To the left side of the VBE there are two important windows: the Project Explorer and the Properties Window. The Project Explorer looks a little like Windows Explorer in the way it organizes information. It is set up as a directory tree where more detailed information within a general concept can be expanded or compressed by clicking on "+" and "−" symbols.

The most general category in VBA is a Project, which is essentially the Excel workbook and any associated additions created in the VBE. The first subfolder contains the Excel objects, which are the individual sheets in the workbook. Code can be stored under a sheet or for the workbook in general, but for this book's examples code will be created and stored in a module.

A module is a separate area to enter code. The code is often organized by purpose and functionality into individual sections called subroutines. Basic macros use one subroutine to accomplish a task, while more advanced macros often use multiple subroutines. Related subroutines are stored in the same module. For instance, a module might be named Print_Routines and contain three subroutines that format and print different sections of the Excel workbook. See Figure 10.3 for a detailed look at a module.

VBA CODE

Writing VBA code is like typing out a set of instructions using words and values that are specific to Excel. Trying to run code that Excel does not understand will generate an error and stop the subroutine from running. For a crash course in VBA, the most basic elements that a new programmer should know are objects, methods, and variables. While there are certainly more components to VBA, learning about

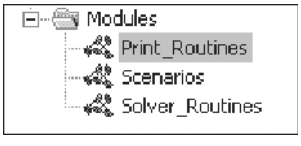

FIGURE 10.3 Modules organize code for subroutines and functions.

the three elements mentioned above will greatly aid a reader in understanding the example code in this chapter.

The first basic element of VBA is an object. In object-oriented programming objects are the building blocks of code. They are items that code performs tasks on and have properties that can be manipulated. Workbooks, worksheets, ranges, cells, and so on are all objects in VBA for Excel.

Objects are worked with primarily through the use of methods. The properties of an object are changed by using different methods or combinations of methods. For example, if a cell is an object and one wanted to change the background of a cell to yellow they would use the .Interior.ColorIndex method. There are hundreds of methods to learn, which are best picked up through examples such as those in this chapter.

Variables are the third basic element of VBA. They are particularly useful for understanding the examples in this chapter because most of the examples make heavy use of variables. A variable is a character or string of characters that are assigned a value. The designers of VBA created specific types of variables in order to save memory and allow a user to create a more efficient macro. For instance, a variable can be declared as a Boolean, meaning that the only acceptable value is "true" or "false." It is important to understand the different types of variables in VBA because if a programmer attempts to assign an inconsistent value to a variable type an error will be generated and the macro will stop running.

SIMPLE AUTOMATION FOR PRINTING AND GOAL SEEK

Print and Goal Seek are the two most commonly used menu tools while operating a cash flow model. Both take time and involve repetitive tasks. For printing there is always the concern that the print area has changed or that the page set up is different. Goal seek requires clicking through a number of fields and entering references and values. Both tasks can be quickly transformed into a single macro that can be run with the click of a button.

MODEL BUILDER 10.1: AUTOMATING PRINT PROCEDURES

1. Press **Alt + F11** to open the VBE. In the Project Explorer, find the name of the Excel workbook. The name of each workbook should be prefaced by VBAProject. Right-click the name, and then on the menu bar click **Insert**, and then click **Module**. This should create a file folder named Modules, with one module named Module1. Using the Properties window rename Module1 to **Print_Routines**. See Figure 10.4 for detail.
2. Double-click **Module1** in the Project Explorer. Select the main code window so there is a blinking cursor. Type the following code:

```
Sub PrintOutput()
```

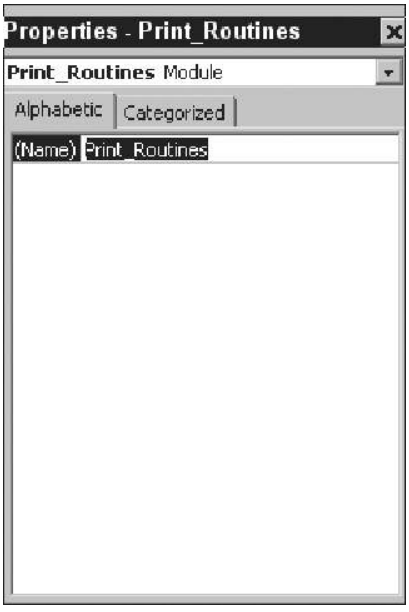

FIGURE 10.4 The Properties window changes the characteristics of items in the VBE.

Starting a module with Sub indicates the beginning of a subroutine, the name PrintOutput is a user created description of the subroutine, followed by an open and close parenthesis. After entering this code and pressing **Enter**, the VBE will automatically enter End Sub, indicating an end to the subroutine. Press a few hard returns after the Sub PrintOutput() so that there is space to enter the main body of code between that beginning and End Sub. The End Sub code should always be at the end.

3. The next step is to turn off screen updating, which displays the results of the macro as it is running. Screen updating slows down a macro considerably, so in most cases it should be turned off. Under the previous code enter:

   Application.ScreenUpdating = False

   This line of code is a perfect example of object/method interaction. The application is the object, which is affected by the screen updating method. Writing the object followed by a period and then the method is a typical object-oriented programming convention.

4. The main part of a print macro is designating the range to be printed, the correct page set up, and ordering the print. Under the previous code enter:

   ```
   Sheets("Output").Select
   Range("A1:P57").Select
   ```

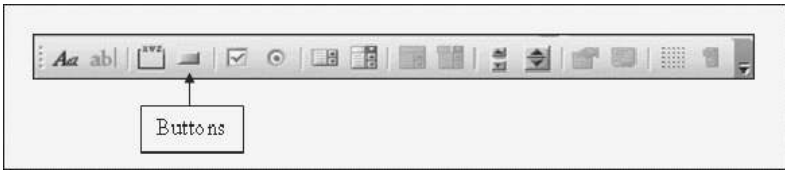

FIGURE 10.5 The Button function on the Form Toolbar is often used to control macros.

```
ActiveSheet.PageSetup.PrintArea = "$A$1:$P$57"
ActiveWindow.SelectedSheets.PrintOut Copies: = 1, Collate: = True
Sheets("Inputs").Select
Range("K2").Select
```

This section of the code selects the Output sheet, selects the entire Output range that needs to be printed, designates that range as the print area, and orders the area to be printed with a few page set up characteristics. Since this macro will be initiated from the Inputs sheet, the last bit of code instructs a cell on the Inputs sheet to be selected when the macro is done. This will prevent the screen from jumping to the Output sheet every time the macro is run.

5. Finally, make sure to activate screen updating before ending the macro by inserting the following under the previous code:

```
Application.ScreenUpdating = True
```

The End Sub that was automatically created should be at the very end.

6. The final step is to create a button on the Inputs page to run this macro quickly. The easiest method for creating macro buttons is using a Form object. Still on the Inputs sheet, make the Forms tool bar visible by selecting View, Toolbars, Forms. This will bring up a series of buttons that looks like Figure 10.5. Click the Button button labeled in Figure 10.5.

   After clicking the Button button, the cursor changes to a crosshair and allows the user to draw a rectangular button by left-clicking (while holding down the click) and dragging until the desired size is created. Make such a button near the G4 area of the Inputs sheet. Immediately upon finishing the click a dialog box will appear that instructs the user to assign a macro. Select the PrintOutputs macro and click **OK**. Finally double click on the name of the button (Button 1) and rename it **Print Output Sheet**.

Model Builder 10.1 Final Code:

```
Sub PrintOutput()
Application.ScreenUpdating = False
  Sheets("Output").Select
  Range("A1:P57").Select
  ActiveSheet.PageSetup.PrintArea = "$A$1 : $P$57"
```

```
    ActiveWindow.SelectedSheets.PrintOut Copies: = 1, Collate: = True
    Sheets("Inputs").Select
    Range("K2").Select

Application.ScreenUpdating = True

End Sub
```

MODEL BUILDER 10.2: AUTOMATING GOAL SEEK TO OPTIMIZE ADVANCE RATES

1. Similar to printing, using goal seek can be automated to work by pushing one button. Goal seek is a bit more complicated to automate because there are a few inputs that need to be entered. Also, certain steps need to be taken to ensure that the goal seek can find an acceptable solution each time. Go to the Project Explorer in VBE and insert another module. Rename the module **Solver_Routines**.
2. Start a new subroutine named **SolveAdvance**. This macro optimizes the senior debt advance rate. It is identical to the goal seek procedure done earlier, where the final senior debt balance is iterated to zero by changing the advance rate.
3. For macros that take a few seconds to run, a useful line of code to insert is one that provides progress information in the Status Bar. The Status Bar in Excel is on the bottom left and typically reads Ready as shown in Figure 10.6.
   The Status Bar can be changed when a macro is running to provide useful information to the model operator. After this line of code and until the Status Bar is changed in the code, the Status Bar will read "Optimize Advance Rate . . .". To do this, enter the following code under the subroutine name:

   ```
   Application.StatusBar = "Optimize Advance Rate . . . "
   ```

   Similar to changing screen updating, the object is the entire application and the method is the StatusBar. Here the message is a constant that is customized. On the line after this, enter the code to turn off screen updating.
4. The next step is to create string constants (constants are objects that are assigned values that do not change throughout the code) that are assigned range names from the Excel sheet. This step does not actually assign a numerical value or reference to the constants, it is assigning the literal text. The purpose of this

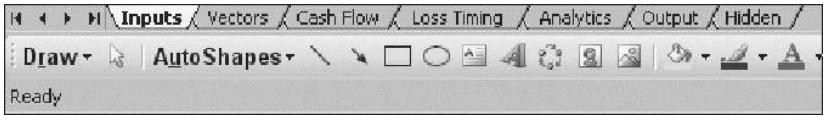

FIGURE 10.6 The Status Bar displays the status of the workbook.

becomes clear when the range names on the Excel sheet are changed. By using constants that are assigned the range name once in the beginning, any changes to the Excel sheet range names only have to be update once in the code. Otherwise any time the Excel sheet range is directly referenced in the code it would have to be changed, which is tedious and prone to errors. Under the previous code enter the following:

```
Const DebtBal As String = "FinalLoanBal"
Const AdvRate1 As String = "LiabAdvRate1"
```

The first line declares DebtBal as a constant. The variable type is a string, which means that any value assigned to DebtBal will be treated as text. Now anytime DebtBal is used in the code it actually has a value of "FinalLoanBal".

5. The next lines of code declare all of the variables:

```
Dim UnpaidLoan As Range
Dim AdvanceRate As Range
```

Variables are declared or dimensioned using the Dim command. A name is created for the variable, followed by the type of variable. In this case, two range variables are declared. It is important to declare variables, otherwise the variable will be declared as a memory intensive variant, which any type of data can be passed through.

6. For any goal seek, it is important to set a realistic starting point for the cell that will be changing during each iteration. Occasionally goal seek is able to find a solution if the changing cell begins with an illogical value. In the case of optimizing the advance rate, the cell that is changing is the advance rate. Since 100 percent is the maximum that the advance rate can be, it makes sense to optimize down from that value each time. To make sure that the advance rate is always 100 percent at the beginning of each optimization, enter the following code after the declared variables:

```
Range(AdvRate1) = 1
```

Range(AdvRate1) is an object from the Excel sheet. Remember that AdvRate1 is a constant that is the equivalent to ''LiabAdvRate1''. The VBE reads Range(AdvRate1) as Range(''LiabAdvRate1''), which is how Excel ranges are referenced in VBA. Recall that LiabAdvRate1 is the named range for the advance rate in Project Model Builder.

Values can be assigned to Excel ranges in this way by stating the range name, an equal sign, and then the value to be assigned. Similarly, a variable can be assigned an Excel value by doing the opposite. See Figure 10.7.

7. Prior to setting up the actual goal seek, code can be used to check to see if an optimal solution already exists, which would save calculation time. If the transaction can be run with a 100 percent advance rate, then that is the optimal solution. Since 100 percent has already been entered in the prior step, all that needs to be done is a check to see if the final senior debt balance is paid. Values

```
'   This assigns the Excel sheet range "Example" a value of 1.
Range("Example") = 1

'   This assigns the variable i a value determined by the Excel sheet range "Example"
i = Range("Example")
```

FIGURE 10.7 Assigning values is an important part of coding for financial applications.

can be checked using an If statement, much like on the Excel sheet. After the previous line of code enter the following:

If Range(DebtBal) > 0.01 Then

The If-Then construct in VBA works very similar to Excel. First the If statement is declared, followed by the test, and then a Then statement. The major difference is that an End If must be inserted at the end of the code that takes place when the statement is true. (See Model Builder 10.2 Final Code at the end of the exercise for placement.)

8. The next step is assigning values to the ranges that were declared earlier. These ranges will be used for the required range inputs for the goal seek. Below the previous line of code enter:

   Set UnpaidLoan = Range(DebtBal)
   Set AdvanceRate = Range(AdvRate1)

   Using the methodology just touched upon in step six, this code assigns values to the VBA ranges from Excel sheet ranges.

9. The actual goal seek command is a single line of code in VBA. Below the previous line of code enter:

   UnpaidLoan.GoalSeek Goal: = 0.01, ChangingCell: = AdvanceRate

   The object UnpaidLoan, which was earlier referenced and valued as the Excel range FinalLoanBal, is the goal range. The goal is set to .01 since the purpose of the exercise is to find the advance rate that completely pays off the loan. Zero is not used because occasionally goal seek will have trouble iterating to such a solution. Finally, the last part of the code designates the changing cell, which in this case is the advance rate.

10. Some final lines of code are necessary to set the Excel environment back to normal. After the previous code enter the following:

    Application.ScreenUpdating = True
    Application.StatusBar = False
    Calculate
    Application.Calculation = xlCalculationAutomatic
    End Sub

    The screen updating should be turned back on, the Status Bar should be set to false (this will set it back to Ready), a calculate should be performed to update

all of the calculations in the model, and finally the model should be set to automatic calculation in case it was turned to manual during the goal seek. Also create a button near D4 on the Excel Inputs sheet named **Optimize Advance Rate** and assign the macro.

Model Builder 10.2 Final Code:

```
Sub SolveAdvance()
Application.StatusBar = "Optimize Advance Rate . . . "
Application.ScreenUpdating = False
Const DebtBal As String = "FinalLoanBal"
Const AdvRate1 As String = "LiabAdvRate1"
Dim UnpaidLoan As Range
Dim AdvanceRate As Range

Range(AdvRate1) = 1

If Range(DebtBal) > 0.01 Then

   Set UnpaidLoan = Range(DebtBal)
   Set AdvanceRate = Range(AdvRate1)
   UnpaidLoan.GoalSeek Goal: = 0.01, ChangingCell: = AdvanceRate
End If

   Application.ScreenUpdating = True
   Application.StatusBar = False
   Calculate
   Application.Calculation = xlCalculationAutomatic
End Sub
```

UNDERSTANDING LOOPING TO AUTOMATE THE ANALYTICS SHEET

Looping is one of the most robust processes that VBA allows. A loop allows code to be repeated for multiple iterations, while giving the option for unique changes during each iteration. Such functionality allows for sensitivity scenarios, loan level amortization, and ultimately stochastic modeling. In the following Model Builder example, a simple goal seek loop is created to perform analytics on the assets and each tranche of debt.

MODEL BUILDER 10.3: AUTOMATING GOAL SEEK TO PERFORM TRANSACTION ANALYTICS

1. Remember that on the Analytics sheet the yield was dependent on making the present value of the cash flows equal to the current values of the different assets

and liabilities. This was done using a goal seek on row 6 of the Analytics sheet (labeled PV Diff). Instead of having to do this by hand for both the assets and liabilities, a macro can be set up that completes all three automatically. The first step is to create a new subroutine in the Solver_Routines module called **SolveYield**.

2. Since this subroutine is essentially a goal seek very similar to the one in Model Builder 10.2, these instructions are condensed. The main point of this exercise is to demonstrate how a basic loop command can be very useful. The beginning of the macro should be familiar and start as follows:

```
Sub SolveYield()
Application.StatusBar = "Solving Analytics . . . "
Application.ScreenUpdating = False
  Const YieldChange As String = "rngYieldChange"
  Const Target As String = "rngYieldTarget"

  Dim YieldRange As Range
  Dim TargetRange As Range
```

3. The first unique part of the code is an additional variable that needs to be declared to assist in looping. This is a declaration of the loop counting variable. Insert this code directly below where the previous code left off:

```
Dim i as Integer
i = 1
```

   The variable i is used in the loop construct to track the number of iterations. To ensure that the variable is cleared to its starting value, i is assigned a value of 1.

4. The next step is to activate the Analytics sheet since it contains information necessary to the macro:

```
Worksheets("Analytics").Select
```

5. The core of this macro is the next few lines of code that create a loop. The most common method to create a loop is using For–Next statements. This construct works by opening with the parameters of the loop using the For statement and looping through each parameter with the Next statement. For instance, by writing the code For i = 1 to 10, the parameters of the loop have been set so there can be 10 possible loops. The code below the For statement will continue to run until the variable i equals 10. Since a programmer only wants certain code below the For to be run during each loop, there needs to be a method for moving on to the next iteration. In the above example the code that moves back to the beginning of the For statement is Next i. This instructs the program to jump back to the beginning of the For statement, but with the next value for the variable (2 in this case).

   The example in Project Model Builder is made slightly more complicated because the number of loops depends on the number of cash flows where the present value needs to be optimized. In the current model, there is a set of cash

flows for the assets and two tranches of debt, so three loops are necessary. However, if there was an additional tranche of debt, then four loops would be necessary; two more tranches of debt and five loops would be necessary, and so on.

This can be overcome by using the range that was created for the PV differences. Recall that the range rngYieldTarget was defined earlier as E6:G6 on the Analytics sheet. This range should always encompass all streams of cash flow that are to be analyzed. If an additional tranche of debt were in column H, then this range should be extended to column H. Since the range will always contain a number of cells equal to the number of cash flow streams to be optimized, a count method can be integrated into the For statement. Enter the following below the previous line of code:

```
For i = 1 To Range(Target).Cells.Count
```

The code begins like a standard For statement with For i = 1 To, but, instead of providing a numerical value, the reference Range(Target).Cells.Count is used. The range ''rngYieldTarget'' is being referenced here using the Target variable. Cells is referring to the individual cells within the range and the Count method counts those cells. Since there are three cells in the range, the For statement is effectively For i = 1 to 3.

6. The next few lines of code are mostly familiar, with a few new concepts. Enter this code below the previous line:

```
Set TargetRange = Range(Target).Cells(1, i)
Set YieldRange = Range(YieldChange).Cells(1, i)
TargetRange.GoalSeek Goal: = 0, ChangingCell: = YieldRange
Next i
```

Notice the addition of the Cells property that comes after Range(Target) and Range(YieldChange). Earlier Cells was used to count the number of cells; but it can also be used to reference individual cells within a range. By entering an open parenthesis right after the Cells property, VBA is instructed to look for a reference within the cells of the respective range. VBA follows an RC (rows, columns) convention, so a (1, 1) would be the first row and first column. In the above code the reference is (1, i), which means row 1 and column i. The variable i will take on a numerical variable depending on the loop iteration. This numerical value will also correspond to the column order in the ranges. Specifically during the first loop i will equal 1, making the reference (1, 1). For each range the first row and first column will be referenced. During the next iteration i will equal 2 starting at the beginning of the For statement, making the reference (1, 2), which will refer to the first row and second column. The entire process will carry on until i is equal to the parameter set in the For statement, which is three (the total range count). This is the crux of how looping works in ranges. Finally, add a button labeled **Calculate Analytics** near D7 on the Inputs sheet and assign the macro.

Model Builder 10.3 Final Code:

```
Sub SolveYield()
Application.StatusBar = "Solving Analytics . . . "
Application.ScreenUpdating = False
   Const YieldChange As String = "rngYieldChange"
   Const Target As String = "rngYieldTarget"
   Dim YieldRange As Range
   Dim TargetRange As Range
   Dim i As Integer
   i = 1
   Worksheets("Analytics").Select

For i = 1 To Range(Target).Cells.Count
   Set TargetRange = Range(Target).Cells(1, i)
   Set YieldRange = Range(YieldChange).Cells(1, i)
   TargetRange.GoalSeek Goal: = 0, ChangingCell: = YieldRange
Next i

Application.ScreenUpdating = True
Application.StatusBar = False
Calculate
Application.Calculation = xlCalculationAutomatic

End Sub
```

AUTOMATED SCENARIO GENERATION

A model operator often needs to run multiple scenarios by changing a number of variables for each run. The manual process would be to change each variable on the Inputs sheet by hand and then save or print out the Output Report for each run. This process can become very inconvenient and repetitive. A VBA solution is ideal for such a problem.

Creating a modelwide scenario generator demonstrates intermediate methods and techniques in VBA that can be transferred to other projects. Reading arrays, looping through arrays, and writing out the outputs are practices that allow a model builder tremendous flexibility. Also, additional concepts such as automating the creation and naming of new workbooks saves a model operator from repetitive time-consuming tasks.

MODEL BUILDER 10.4: CREATING A TRANSACTION SCENARIO GENERATOR

1. The first steps take place on the Inputs sheet of the Excel workbook. In cell B35 on the Inputs sheet, type the label **Scenario Generator** and label cell B36

Scenario. Cells C36 through E36 are the labels for assumptions that are to be varied. This is up to the user, but in this example enter the following:
C36: Gross Loss
D36: Loss Timing
E36: Recovery
For now enter scenario number labels starting from 1 for B37:B42. The assumptions to vary can be copied from the Model Builder 10.4 file on the CD-ROM or made up at the user's discretion. It is recommended to copy the data from the CD-ROM so that the screenshots and examples can be tied to the user's model under construction. Also, a few ranges need to be named:
C37:C41: rng_ScenGen1
D37:D41: rng_ScenGen2
E37:E41: rng_ScenGen3
The new area should look like Figure 10.8.

2. Next launch the VBE and create a new module named **Scenarios**. In that module, start a new subroutine called **Scenario_Generator**().
3. The first part of the code is declaring constants and variables. Enter the following below the subroutine name:

```
Const MaxScens = 5
Dim i As Integer, k As Integer
Dim Scen1Array(1 To MaxScens), Scen2Array(1 To MaxScens), Scen3Array
(1 To MaxScens)
Dim Scen1Option As String, Scen2Option As String, Scen3Option As String
```

In this exercise, a constant is set up for the maximum number of scenarios, counting variables i and k are created, and array and string variables are declared. Arrays are a new concept for this section. An array is a range of data that can be multidimensional (containing both columns, rows, or multiple combinations). Arrays can be visualized as a range of cells on the worksheet. In this example the arrays will be one-dimensional with a single column of data.

It is important to understand that an array can be visualized as a range of cells in VBA, but it does not entirely act like one. To get data into an array in VBA it must be read in through looping; copying and pasting will not work.

	A	B	C	D	E	F
34						
35		SCENARIO GENERATOR				
36		Scenario	Gross Loss	Loss Timing	Recovery	
37		1	1.00%	Timing Curve 1	50.00%	
38		2	3.00%	Timing Curve 1	50.00%	
39		3	5.00%	Timing Curve 1	50.00%	
40		4	7.00%	Timing Curve 1	30.00%	
41		5	10.00%	Timing Curve 3	30.00%	
42						

FIGURE 10.8 The Scenario Generator is controlled on the Inputs sheet, but run using VBA.

Similar loops must be created to get parts of an array out of VBA and onto an Excel sheet, unless the entire array is to be written to the sheet.

4. As in the previous macros, assign string values to certain variables so the code does not have to be changed too drastically if the sheet names change. In this case there are three sheet references that will be varied per scenario. Insert the follow code after the variable declarations (also notice that screen updating is turned off here):

```
Scen1Option = "pdrCumLoss1"
Scen2Option = "pdrLossTime1"
Scen3Option = "pdrRecovRate1"
Application.ScreenUpdating = False
```

5. The first loop that needs to be set up reads in the possible values for each scenario and stores those values in VBA arrays. This is done by creating a loop for each possible scenario (5 in this case) and storing a value in a specific, ordinal section of an array. Below the previous code enter the following:

```
For i = 1 To MaxScens
   Scen1Array(i) = Range("rng_ScenGen1").Cells(i, 1)
   Scen2Array(i) = Range("rng_ScenGen2").Cells(i, 1)
   Scen3Array(i) = Range("rng_ScenGen3").Cells(i, 1)
Next i
```

   In this code, the array Scen1Array(i) will store the value in the first cell of the sheet range rng_ScenGen1. The macro understands to do this because the first loop changes the variable i to a value of 1, which means that Scen1Array(1) will be equal to the 1 row, 1 column in rng_ScenGen1 (through the use of the Cells method). Each array will be filled until the maximum number of scenarios is reached (5 in the example model).

6. This section of code is another loop. This time each iteration of the loop will complete all of the tasks to calculate and export the scenario. The first part of this code is transferring the values from the arrays to the sheet for each iteration. Enter the following after the previous code:

```
For k = 1 To MaxScens
   Range(Scen1Option) = Scen1Array(k)
   Range(Scen2Option) = Scen2Array(k)
   Range(Scen3Option) = Scen3Array(k)
   Calculate
```

   For each k loop, the respective array value is written back to the Excel sheet in the range determined by the string variables. The command Calculate is included to make sure that all of the formulas in the Excel sheet calculate since changes will have occurred due to changing the three ranges.

7. After the workbook is calculated, the user may want to optimize the advance rate. If this is the case insert the following code:

```
Call SolveAdvance
```

The Call command runs the subroutine that follows it.

8. After each scenario is calculated and the debt optimized, the results should be recorded and stored. The Output sheet contains all of the results that are wanted and is in a constant form. This makes it easy to copy and paste. First the Output sheet needs to be copied at the end of each k loop iteration:

```
Sheets("Output").Select
Range("A1:P38").Copy
```

This code selects the Output sheet and copies the data ranges. Note that the graphs are not copied since they would retain the original links and not have scenario specific results.

9. The copied data should be stored in a separate worksheet. To do this, a new worksheet needs to be inserted, the data pasted, and formatted correctly. Below the previous code enter the following:

```
Sheets.Add
Range("A1").Select
Selection.PasteSpecial Paste: = xlValues
Selection.PasteSpecial Paste: = xlFormats
```

This code adds a sheet to the workbook, selects cell A1, pastes the values of the copied range, and then pastes the formats of the copied range.

10. To save time the sheet should be automatically assigned a name and placed in a consistent place in the workbook. To accomplish this enter the following below the previous code:

```
ActiveSheet.Name = Format("Scen Output" & k)
ActiveSheet.Move After: = Sheets(6 + k)
```

Both the Name method and the Move method affect the ActiveSheet. The Name method changes the worksheet name to begin with Scen Output, followed by whatever number scenario the loop is iterating through. The Move method moves the sheet after a designated number of sheets. In this case the first k loop will move the sheet at the end of the seventh workbook sheet (6 + k on the first loop is seven).

11. Change the Status Bar so that the user knows the progression of the macro based on the loop iteration. Since this is the last line of code for the iteration, end the For Loop with a Next command. Enter the following below the previous code:

```
Application.StatusBar = "Running Scenario: " & Str(k) & " of " & Str(MaxScens)
Next k
```

12. A few lines of code are needed to clean up the macro. Below the last code enter the following:

```
Sheets("Inputs").Select
```

```
Application.ScreenUpdating = True
Application.StatusBar = False
End Sub
```

Instead of the macro ending on a scenario output and the user having to select the Input sheet again, the first line of the previous code automatically selects the Input sheet. Screen updating is turned back on and the Status Bar is reset. Finally, create a button on the Inputs sheet near cell G7 and assign this macro to run when it is pushed. The button should be labeled appropriately, such as **Generate Scenarios**.

13. An additional subroutine that saves time is one that deletes old scenario sheets. This can be added at the user's discretion; but it can be incredibly useful if deleting old scenarios is becoming a repetitive task. It also shows model builders how to identify specific sheets in the workbook automatically and perform operations on them. First, create a new subroutine and declare a worksheet variable:

```
Sub DeleteSheets()
Dim VBAwksht As Worksheet
```

14. Next, a loop should be inserted to delete the old scenarios that are stored in the workbook. This loop will loop through each sheet, check to see if the sheet is a scenario output (remember they all start with the same naming convention of "Scen Output"), and delete the sheet if it is. Enter the following code below the code that turns off screen updating:

```
For Each VBAwksht In ActiveWorkbook.Worksheets
  If Left(VBAwksht.Name, 11) = "Scen Output" Then
    VBAwksht.Delete

End If
Next
```

The For-Next loop is slightly different since it essentially reads "For each object of this type in this object". The If statement uses the Left command, which looks at each worksheet's name and checks to see if the first 11 characters starting from the left read Scen Output. If the statement is true then the worksheet is deleted, otherwise it is skipped and the loop continues until there are no more worksheets in the workbook. When this macro is actually run there will be a prompt to delete each individual sheet as a protection against deleting important data. This macro is contained in the scenarios module in file *MB10-4.xls* in the Ch10 folder on the CD-ROM, but no button has been created in the model.

Model Builder 10.4 Final Code:

```
Sub Scenario_Generator()
Const MaxScens = 5
Dim i As Integer, k As Integer
Dim Scen1Array(1 To MaxScens), Scen2Array(1 To MaxScens), Scen3Array
(1 To MaxScens)
```

```
Dim Scen1Option As String, Scen2Option As String, Scen3Option As String
Dim VBAwksht As Worksheet
Scen1Option = "pdrCumLoss1"
Scen2Option = "pdrLossTime1"
Scen3Option = "pdrRecovRate1"
Application.ScreenUpdating = False
For Each VBAwksht In ActiveWorkbook.Worksheets
   If Left(VBAwksht.Name, 11) = "Scen Output" Then
      VBAwksht.Delete
   End If
Next
For i = 1 To MaxScens
   Scen1Array(i) = Range("rng_ScenGen1").Cells(i, 1)
   Scen2Array(i) = Range("rng_ScenGen2").Cells(i, 1)
   Scen3Array(i) = Range("rng_ScenGen3").Cells(i, 1)
Next i
For k = 1 To MaxScens
   Range(Scen1Option) = Scen1Array(k)
   Range(Scen2Option) = Scen2Array(k)
   Range(Scen3Option) = Scen3Array(k)
   Calculate

Call SolveAdvance

   Sheets("Output").Select
   Range("A1:P38").Copy
   Sheets.Add
   Range("A1").Select
   Selection.PasteSpecial Paste: = xlValues
   Selection.PasteSpecial Paste: = xlFormats
   ActiveSheet.Name = Format("Scen Output" & k)
   ActiveSheet.Move After: = Sheets(6 + k)
   Application.StatusBar = "Running Scenario: " & Str(k) & " of " &
Str(MaxScens)

Next k
Sheets("Inputs").Select
Application.ScreenUpdating = True
Application.StatusBar = False
End Sub
```

Optional Macro

```
Sub DeleteSheets()
Dim VBAwksht As Worksheet
   For Each VBAwksht In ActiveWorkbook.Worksheets
```

```
        If Left(VBAwksht.Name, 11) = "Scen Output" Then
          VBAwksht.Delete
        End If
      Next
    End Sub
```

WORKING WITH MACROS IN EXCEL

Now that a number of example macros have been created, it should be stated outright that macros can do quirky things. Personal experience has witnessed a handful of times that an error was generated when everything seemed completely correct. Code was rechecked for the slightest misspelling or typo. Macros were debugged using the watch window and break commands. Hours spent poring over each line of code only to realize that after saving the model, shutting it down, and reopening it, the macro worked perfectly!

Glitches like this are rare; but they can occur. Much of the time an error is due to a misspelled variable or reference. Other typical errors include a data mismatch where the variable was declared as a certain type, but the code is directing a different type of value to be passed through, a value exceeds the constraints of a dimensioned array, or a looping variable has exceeded the parameters of the loop. Learning how to use code breaks and the watch window are invaluable for debugging these and all other errors. Even though the errors can be frustrating and the learning curve high, the benefit of understanding VBA is well worth it.