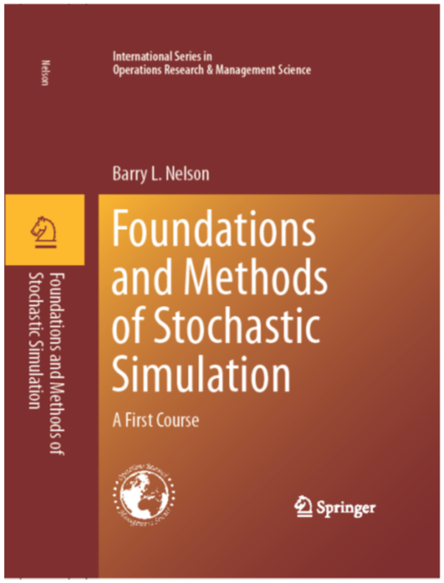

International Series in Operations Research & Management Science
Barry L. Nelson
Foundations and Methods of Stochastic Simulation
A First Course
Springer

Chapter 2.3: VBA Primer

©Barry L. Nelson
Northwestern University
December 2012

Visual Basic for Applications

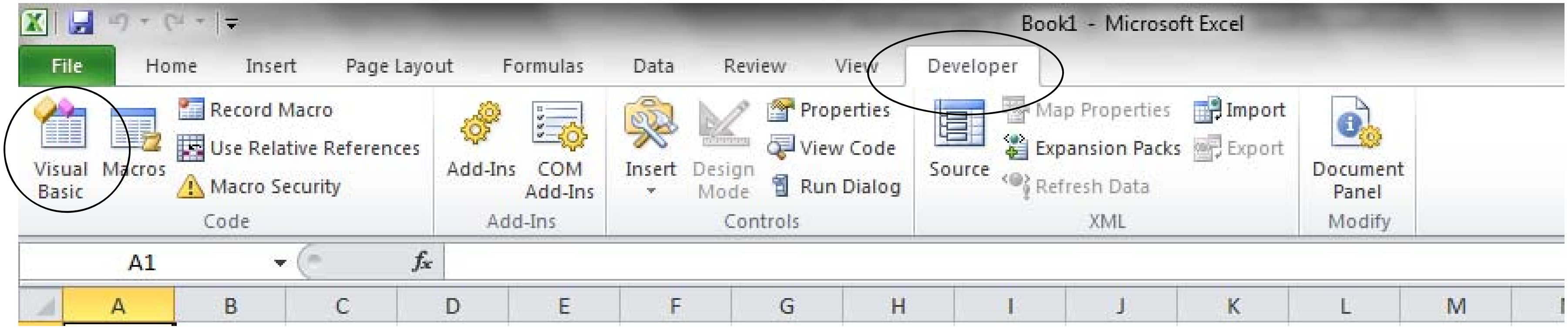
- VBA a significant subset of the stand-alone Visual Basic programming language
- It is integrated into Microsoft Office applications (and others)
- It is the macro language of Excel
- You can add
  - Forms for dialog boxes with user input
  - **Modules containing procedures ← this lecture**
  - **Class Modules for object definitions ← later**

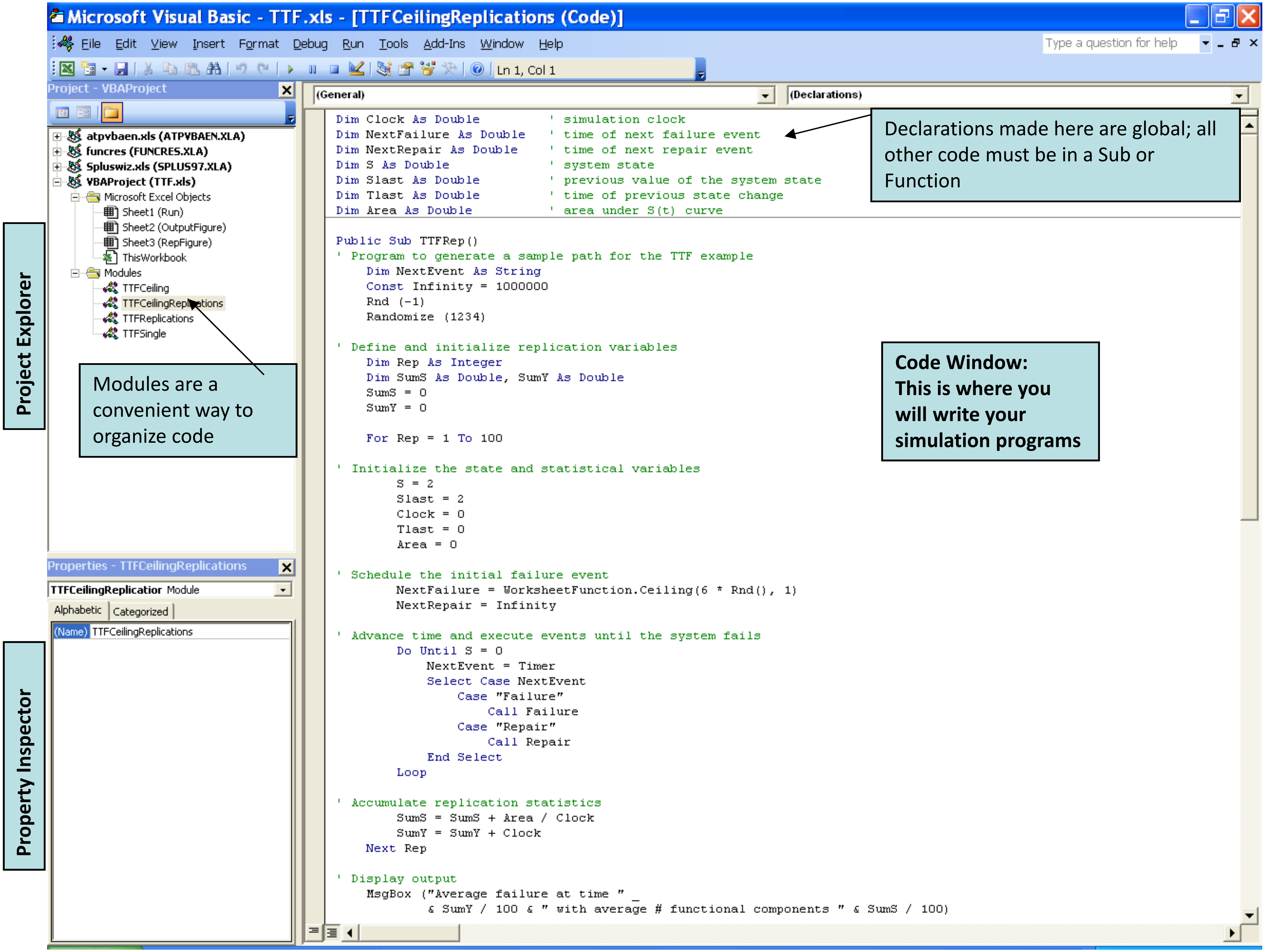
VBA & Excel for discrete-event simulation

- Advantages
  - VBA is easy to learn and available almost everywhere
  - VBA is a full-featured programming language so what you learn translates directly to C++, Java, etc.
  - You have access to Excel functions for computation and Excel itself for storing and analyzing outputs
- Disadvantages
  - VBA is interpreted, not compiled, so execution is slow
  - Excel functions can be buggy

Accessing VBA in Excel 2010+

- You launch the Visual Basic Editor from the **Developer Tab**.
- If you don’t have a Developer Tab in the Ribbon, then go to the **File, Options**, and add the “Developer” tab to the ribbon.





```
Dim Clock As Double          ' simulation clock
Dim NextFailure As Double    ' time of next failure event
Dim NextRepair As Double     ' time of next repair event
Dim S As Double              ' system state
Dim Slast As Double          ' previous value of the system state
Dim Tlast As Double          ' time of previous state change
Dim Area As Double           ' area under S(t) curve

Public Sub TTFRep()
' Program to generate a sample path for the TTF example
    Dim NextEvent As String
    Const Infinity = 1000000
    Rnd (-1)
    Randomize (1234)

' Define and initialize replication variables
    Dim Rep As Integer
    Dim SumS As Double, SumY As Double
    SumS = 0
    SumY = 0

    For Rep = 1 To 100

' Initialize the state and statistical variables
        S = 2
        Slast = 2
        Clock = 0
        Tlast = 0
        Area = 0

' Schedule the initial failure event
        NextFailure = WorksheetFunction.Ceiling(6 * Rnd(), 1)
        NextRepair = Infinity

' Advance time and execute events until the system fails
        Do Until S = 0
            NextEvent = Timer
            Select Case NextEvent
                Case "Failure"
                    Call Failure
                Case "Repair"
                    Call Repair
            End Select
        Loop

' Accumulate replication statistics
        SumS = SumS + Area / Clock
        SumY = SumY + Clock
    Next Rep

' Display output
    MsgBox ("Average failure at time " _
        & SumY / 100 & " with average # functional components " & SumS / 100)
```

Declarations made here are global; all other code must be in a Sub or Function

Project Explorer

Modules are a convenient way to organize code

Code Window: This is where you will write your simulation programs

Property Inspector

Structure of a VBA project

- **Modules** are collections of VBA code
  - From menu: Insert → Module
  - Module can be named in the Property Inspector
  - Includes:
    - Global Declarations that occur before any Subs or Functions
    - Procedures (Subs) and Functions of executable code
- **Class Modules** are covered later….
- **UserForms** are graphic objects for user input and output; we will not work with UserForms

Variables

- It is good programming practice to declare the type of all variables
- Standard types in VBA
  - **Single, Double** (single and double-precision reals)
  - **Integer, Long** (small 32k and large 2 billion integers)
  - **String** (character variables)
  - **Boolean** (True or False)

Variable scope

- “Scope” determines to what part of your VBA code a variable is visible
- Project level: Entire workbook
  - **Public** X As Double
  - Must be declared at the top of a Module
  - You will rarely want to do this
- Module level: Entire module (“Global”)
  - {**Dim** or **Private**} Z As Long
  - Must be declared at the top of a Module
- Procedure level: Sub or Function (“Local”)
  - {**Dim** or **Private**} Y As String
  - Declared inside a Sub or Function

Constants & Statics

- **Const** constantName [**As** type] = expression
  - Value cannot be changed

  Const PI = 3.1, NumPLANETS = 9

- **Static** staticName **As** type
  - **Static** causes variables in Subs and Functions to retain their values (normally lost when you exit Sub or Function)

  Static yourName **As String**

Examples

The values of these variables in the initial declarations are available to all Subs or Functions in this Module, but not to other Modules

```
Dim Clock As Double            ' simulation clock
Dim NextFailure As Double      ' time of next failure event
Dim NextRepair As Double       ' time of next repair event
Dim S As Double                ' system state
Dim Slast As Double            ' previous value of the system state
Dim Tlast As Double            ' time of previous state change
Dim Area As Double             ' area under S(t) curve
```

```
Public Sub TTFRep()
' Program to generate a sample path for the TTF example
    Dim NextEvent As String
    Const Infinity = 1000000
    Rnd (-1)
    Randomize (1234)

' Define and initialize replication variables
    Dim Rep As Integer
    Dim SumS As Double, SumY As Double
```

Declaration of a constant

These variables' values are only known to Sub TTFRep

Arrays

- Arrays can have any number of dimensions
- Where the indices start is up to you

```
Dim X(1 to 100) as Integer
Dim Elf(0 to 5, 0 to 20) as String
```

- You can also dynamically allocate and reallocate an array

```
Dim Calendar() as Integer
ReDim Calendar (1 to 31) as Integer
```

Control Structures

- VBA contains the usual control structures for branching, looping, etc.
- We will present a few of the most useful ones.
- A consistent feature of VBA control structures is that there is an explicit "end" statement

If-Then-Else-Endif Structure

```
If Index = 0 Then
      X = X + 1
      Y = VBA.Sqr(X)
Else If Index = 1 Then
      Y = VBA.Sqr(X)
Else If Index = 2 Then
      Y = X
Else
      X = 0
End If
```

Note: All control structures in VBA have an explicit ending statement

Select Case Structure

```
Select Case IndexVariable
    Case 0
            statements…
    Case 1 to 10
            statements…
    Case Is < 0
            statements…
    Case NumSteps
            statements…
    Case Else
            statements…
End Select
```

The case will be selected based on the value of this variable

Notice that the “cases” can be constants, ranges, conditions and variables; this is a powerful control structure that we will use to select events to execute

Loops

```
For counter = start To end [Step increment]
        statements
Next counter
```

```
Do
   statements…
Loop {While|Until} condition
```

```
Do {While|Until} condition
   statements…
Loop
```

```
For Each element In group
        statements
Next element
```

```
For Rep = 1 To 100

' Initialize the state and statistical variables
        S = 2
        Slast = 2
        Clock = 0
        Tlast = 0
        Area = 0

' Schedule the initial failure event
        NextFailure = WorksheetFunction.Ceiling(6 * Rnd(), 1)
        NextRepair = Infinity

' Advance time and execute events until the system fails
        Do Until S = 0
            NextEvent = Timer
            Select Case NextEvent
                Case "Failure"
                    Call Failure
                Case "Repair"
                    Call Repair
            End Select
        Loop

' Accumulate replication statistics
        SumS = SumS + Area / Clock
        SumY = SumY + Clock
    Next Rep
```

Timer is a function that returns the name of the next event; more on that later…

Because the “Until” condition appears at the top it is tested before the loop is executed for the first time

Notice that NextEvent is a String variable so the cases are in “ ”

Exiting control structures

```
For J = 1 To 10 Step 2
    [statement block]
    Exit For
    [statement block]
Next J

Do
    [statement block]
    Exit Do
    [statement block]
Loop Until Check = False
```

Optional statements to allow early, graceful exit from the loop before the termination condition

Subs and Functions: Where the action occurs

- **Private Sub** mySub (arguments)
  - no value returned except through arguments
  - Called when needed
    Call mySub(param1, param2)
- **Private Function** myFunction (arguments) **As** type
  - value returned
  - assign return value to function name
    X = myFunction(2, 7, Z)
- By default Subs and Functions have module-level scope; can have project-level scope by declaring them **Public**

Subs

- Basic syntax:

```
{Public|Private} Sub name(arguments)
    statements…
    Exit Sub
    statements…
End Sub
```

Optional way to leave the Sub
before reaching the End statement

Functions

- Basic syntax:

```
{Public|Private} Function name(arguments) AS type
        statements…
        name = return value
        Exit Function
        statements…
End Function
```

Value returned as the name of the function

Optional way to leave the Function before reaching the End statement

Arguments for procedures

- Pass by Reference (default) means that changes to the value of the variable will be returned

```
Sub stuff(item As String, price As Integer)
```

- Pass by Value means only the value is passed so the original variable is unchanged

```
Sub stuff(ByVal item As String, ByVal price As Integer)
```

```
Private Function Timer() As String
    Const Infinity = 1000000

' Determine the next event and advance time
    If NextFailure < NextRepair Then
        Timer = "Failure"
        Clock = NextFailure
        NextFailure = Infinity
    Else
        Timer = "Repair"
        Clock = NextRepair
        NextRepair = Infinity
    End If
End Function
```

Notice that a Function must have a type since it returns a value

Value is returned as the name of the Function

```
Private Sub Failure()
' Failure event
' Update state and schedule future events
    S = S - 1
    If S = 1 Then
        NextFailure = Clock + WorksheetFunction.Ceiling(6 * Rnd(), 1)
        NextRepair = Clock + 2.5
    End If

' Update area under the S(t) curve
    Area = Area + Slast * (Clock - Tlast)
    Tlast = Clock
    Slast = S
End Sub
```

No arguments are passed here, so how does the Function or Sub know the values of these variable?

Another example from VBASim

“Variant” allows any variable type

The underscore character means “continued on the next line”

```
Public Sub Report(Output As Variant, WhichSheet As String, Row As Integer,_
        Column As Integer)

' basic report writing sub to put an output on worksheet WhichSheet(Row, Column)

Worksheets(WhichSheet).Cells(Row, Column) = Output

End Sub
```

This is one way to reference a particular cell in a worksheet

Interacting with Excel

- We will frequently interact with Excel in two ways:
  1. Reading from and writing to cells in a worksheet
  2. Using Excel intrinsic functions within VBA code

Writing to a worksheet

- Put the absolute value of the variable Fudge in row I=2, column J=20 of the Worksheet named Sheet1.

```
Worksheets(“Sheet1”).Cells(2,20) = VBA.Abs(Fudge)

Worksheets(“Sheet1”).Cells(I,J) = VBA.Abs(Fudge)

Worksheets(“Sheet1”).Range(“T2”)=VBA.Abs(Fudge)
```

This is how you address
VBA intrinsic functions

Reading from a worksheet

- Here we read the value from row 4, column 7 of the worksheet "myData"

```
X = Worksheets(“myData”).Cells(4, 7)
```

Using an Excel function

- VBA has a limited number of built-in functions which you access as **VBA.**function

  X = VBA.Exp(7)

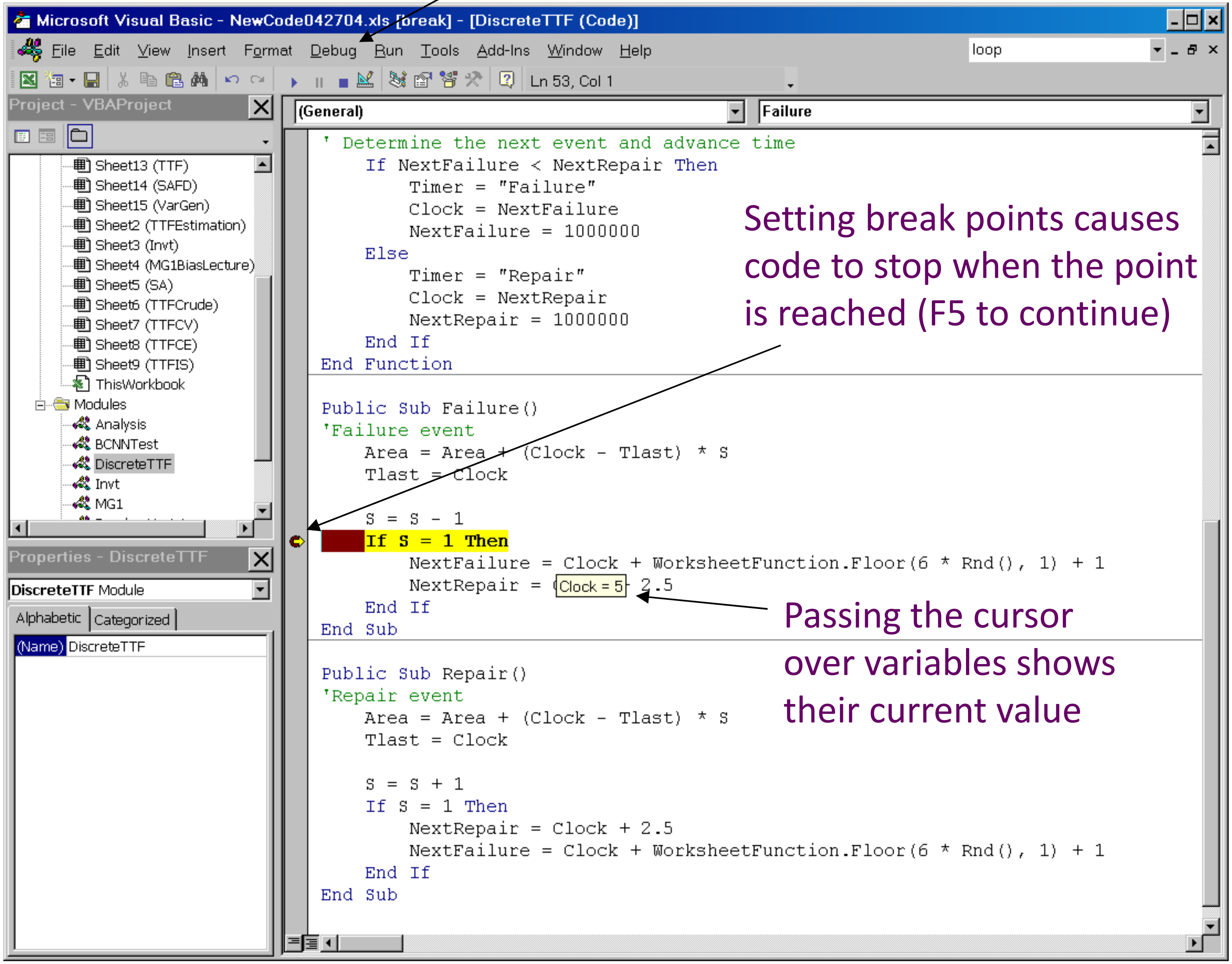
- You can use any Excel worksheet function in the following way: **WorksheetFunction.**functionname
  - W = WorksheetFunction.Max(0, W + S - a)
  - NextFailure = WorksheetFunction.Ceiling(6 * Rnd(), 1)

Running the Code

- Perhaps the easiest way to run the code is to place your cursor in the module you want to run and press the **Run** ▶ button (which is also function key F5).
- Your modules will also appear as Macros that can be run from Excel

Debugging

Useful tools in the Debug menu, especially setting a Watch to track how a variable or expression changes



```
' Determine the next event and advance time
    If NextFailure < NextRepair Then
        Timer = "Failure"
        Clock = NextFailure
        NextFailure = 1000000
    Else
        Timer = "Repair"
        Clock = NextRepair
        NextRepair = 1000000
    End If
End Function

Public Sub Failure()
'Failure event
    Area = Area + (Clock - Tlast) * S
    Tlast = Clock

    S = S - 1
    If S = 1 Then
        NextFailure = Clock + WorksheetFunction.Floor(6 * Rnd(), 1) + 1
        NextRepair = Clock = 5 2.5
    End If
End Sub

Public Sub Repair()
'Repair event
    Area = Area + (Clock - Tlast) * S
    Tlast = Clock

    S = S + 1
    If S = 1 Then
        NextRepair = Clock + 2.5
        NextFailure = Clock + WorksheetFunction.Floor(6 * Rnd(), 1) + 1
    End If
End Sub
```

Setting break points causes code to stop when the point is reached (F5 to continue)

Passing the cursor over variables shows their current value

Finishing up

- Exercise:
  - Insert a new Module and name it “Test”
  - Write a Function that evaluates the standard normal density function
    $f(x) = exp(-x^2/2)/sqr(2\pi)$
  - Write a Sub that uses a loop to call your function and evaluate the standard normal density at *x* = -2.5, -1.5, -0.5, 0.5, 1.5, 2.5 then write the results in column B of an Excel worksheet