FormFormatter: A Software Tool for Constructing Mouse-Optional Visual Basic Applications

Trevor N. Mansuy and Robert J. Hilderman
Department of Computer Science
University of Regina
Regina, SK, Canada S4S0A2
{mansuy1t, robert.hilderman}@uregina.ca

Abstract: Repeated use of a pointing device can cause repetitive stress injuries due to the nature of the movement required to manipulate the device. Several solutions for this problem exist; one such solution is a keyboard-based interface where use of the mouse is optional. That is, the keyboard is used as the primary input device to control all aspects of interacting with software. In order for an application to utilize this keyboard-based interface, it must be added to the source code of the application. In this paper we will look at the feasibility of a software tool that can automate this activity. Given the volume of Visual Basic source code in existence, such a tool would be extremely useful. We will discuss the implementation, capabilities, and limitations of FormFormatter, an automatic Visual Basic source code formatter designed to automatically convert any application to one where the use of the mouse is optional.

1. Introduction

Keyboard-based input strategies are not commonly found in software. Software developers tend to overlook them or discount them as a possibility for a number of reasons, like finance, time, and problem complexity. Alternative methods of input are becoming increasingly more necessary due to the increased risk in developing repetitive stress injuries in today's workplace. According to the US Bureau of Labor Statistics, repetitive stress injuries made up 67% [1] of all workplace related injuries in 2000. In 1988, there were no repetitive stress injuries reported which were associated with mouse use. By 1993, the number of reported cases soared to 325, 000 [2]. The time frame given with these numbers was a time when the computer mouse was beginning to grow in popularity. With the emergence of graphical user interfaces, a mouse became a primary input device on a desktop computer. So, as mouse use began to rise, so did mouse related injuries. Obviously, this is a problem that deserves attention. Keeping in mind the volume of source code already in existence, adding these alternative input methods is a monstrous task.

A solution to the above problems is a tool that automatically and intelligently adds a keyboard-based input strategy to an application with the click of a button. Such a tool would save time and money, and it would allow quick and efficient modification of large quantities of source code.

FormFormatter is an application written in Visual Basic that modifies Visual Basic source code to make the use of the mouse optional. FormFormatter takes a Visual Basic project file as input, and as output generates the same application with a keyboard-based interface where use of the mouse is optional.

2. Keyboard-Based Interface Overview

The keyboard-based interface generated by FormFormatter provides an alternative approach to pointing in a Visual Basic application. Fig. 1 contains an example of a Visual Basic form before the keyboard-based interface was implemented. Fig. 2 contains the same form, but is how the form looks after the keyboard-based interface is implemented. Each item on a form is associated with a label where the labels are represented as numbers. To provide a consistent interface, all menu items are also labeled. To facilitate label entry (i.e., Clicking on an item), each form in the application has a special input box where the user types a number to indicate the item to be manipulated. The user can return to the special input box to select another item by pressing the escape key.

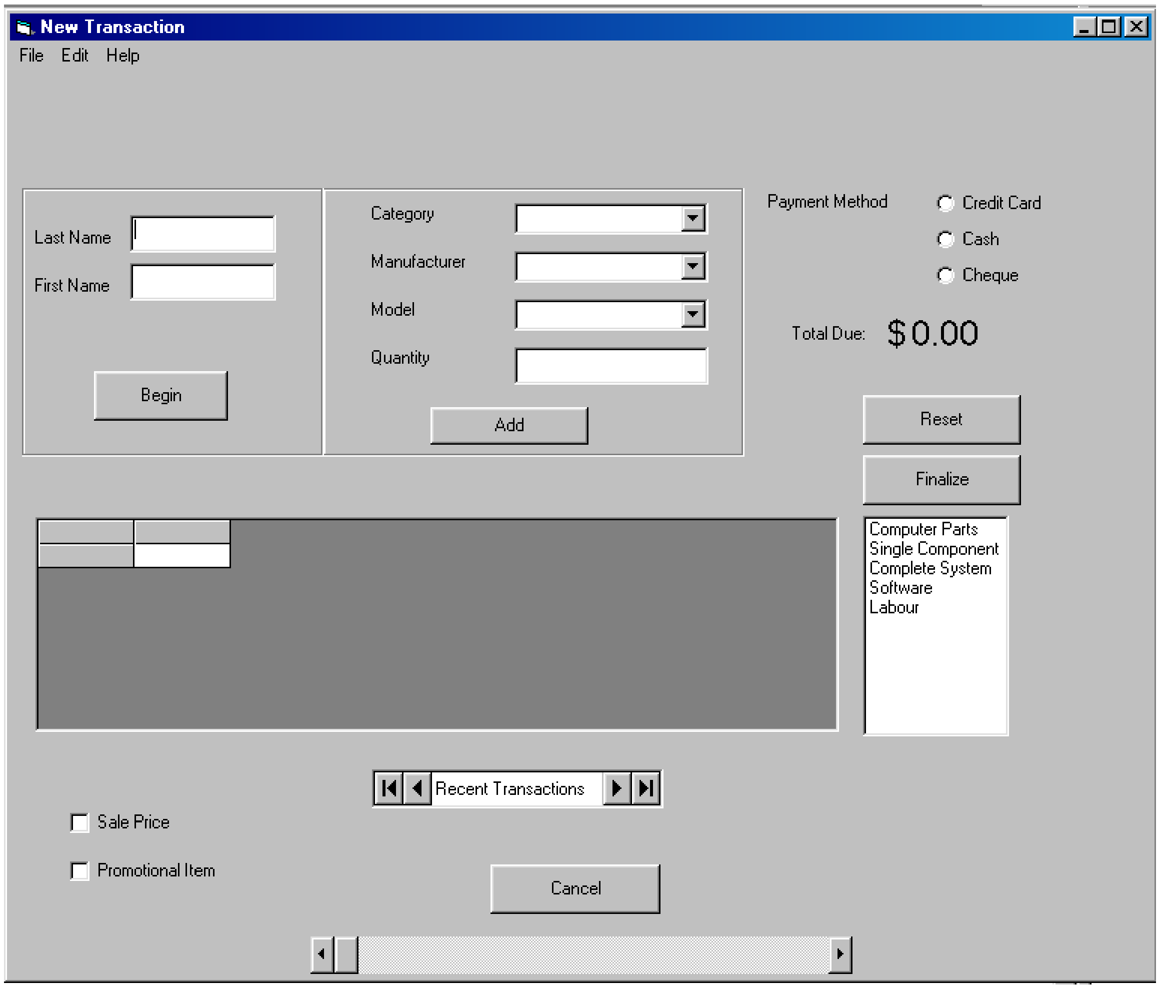
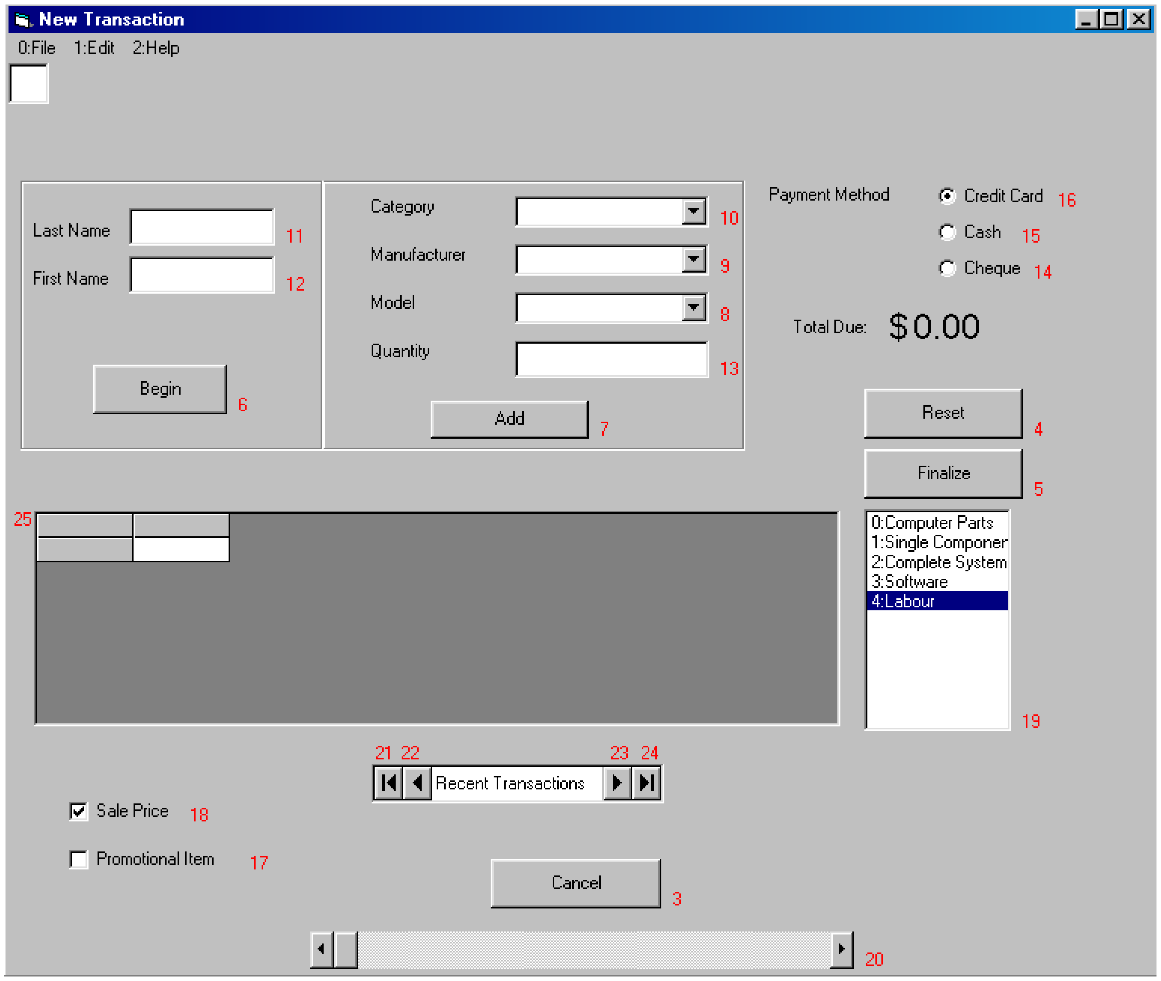


Fig. 1. A Visual Basic form before being processed by form formatter



Payment Method [✓ Credit Card] 16 Cash 15 Cheque 14

Total Due: $0.00

[✓ Sale Price] 18

Fig. 2. An example of a form processed by form formatter.

After an item has been selected, the program will perform some action associated with that item. The action performed depends on the type of item selected. When a button (labels 3, 4, 5, 6, and 7 on Fig. 2) is selected, the button is “pressed” by calling the button’s event handler. When a menu (labels 0, 1, and 2 on Fig. 2) is selected, the menu is dropped down by simulating the pressing of the F4 key. Submenu items also have a label. These items are selected by pressing the associated single digit label on the number pad. Selecting combo boxes (labels 8, 9, and 10 on Fig. 2) is similar to selecting menus. When a combo box is selected, the combo box drops down and the user can use the arrow keys to select an item from the list. When a list box (label 19 on Fig. 2) is selected, items in the list box are selected by entering the label associated with an item. The enter key must be pressed to complete the selection. When a text box (labels 11, 12, and 13 on

Fig. 2) is selected, text can be entered into the selected text box. When a scrollbar (label 20 on Fig. 2) is selected, the scrollbar position is controlled using the arrow keys. A data source control (labels 21, 22, 23, and 24 on Fig. 2) has 4 buttons which perform the functions first record, previous record, next record, and last record. Each one of these buttons has a separate label, and when selected behave in the way previously described for buttons. When an option button (labels 14, 15, and 16 on Fig. 2) is selected, that option button gets turned on and all other option buttons get turned off. When a checkbox (labels 17 and 18 on Fig. 2)is selected, it changes value based on its current value. If the checkbox is checked, then it becomes unchecked. If unchecked, it becomes checked. Finally, when a flexgrid (label 25 on Fig. 2) is selected, the user can then navigate the grid using the arrow keys. The color red was chosen for the labels because it makes them stand out quite clearly on the form.

4. Form File Format

A Visual Basic form file has a well-defined structure where the code contained within the form is divided into various sections, and each section is always in the same place with the same title in any form file. The first section, called the form section, contains the forms properties, and some other general information about the file. A sample form section is shown in Fig. 3 (the line numbers are not actually contained in the file and are shown for reader convenience). In line 1, "VERSION 5.00" tells the Visual Basic IDE the version of Visual Basic that was used to create the form. Line 2 denotes the start of a new control. The format of a line denoting the start of a new control is always "Begin <object type> <object name>". In this case, line 2 is denotes the start of a form control. A form control is the actual surface of an application that contains other controls. The "Begin" contained in line 2 is the top level begin. Every other object in the form file must follow this begin tag because everything else intuitively belongs on the form. Lines 3 to 11 all have a similar structure, consisting of a property name, followed by the equals sign followed by the property value.

```
1.      VERSION 5.00
2.      Begin VB.Form Form1
3.              Caption      =  "Form1"
4.              ClientHeight   =  6720
5.              ClientLeft     =  60
6.              ClientTop      =  345
7.              ClientWidth    =  12045
8.              LinkTopic      =  "Form1"
9.              ScaleHeight    =  6720
10.             ScaleWidth     =  12045
11.             StartUpPosition =  3 'Windows Default
```

Fig. 3. A sample form section

After the form properties are defined, controls contained on the form are defined. For example, a Command button control, called Command6, is shown in Fig. 4. The first line has the same structure as the Begin Form line in Fig. 3. Lines 2 to 7 define the properties of the object. The End line denotes the end of that control, so a property located after line 8 but before another Begin would belong to the form.

```
1.      Begin VB.CommandButton Command6
2.              Caption      =  "Command6"
3.              Height       =  495
4.              Left         =  4560
5.              TabIndex      =  10
6.              Top          =  3720
7.              Width        =  975
8.      End
```

Fig. 4. A sample command button

The rest of the control definition follows the same structure as Fig 4. However there is one control that is worth mentioning. Fig. 5 is an example of a frame control and illustrates the idea that a frame contains other controls. For example, Fig. 6 contains the code for the frame control shown in Fig. 5. In Fig. 6, lines 1 to 7 contain the properties of the frame control itself. Immediately after these lines, on line 8, we encounter a Begin line for another control without encountering an end line for the frame control. This signifies that the button control code in Fig. 6 belongs to the frame in Fig. 6. Only the control code contained between line 1 and line 16 of Fig. 6 would appear inside the frame. This idea applies to all controls in a Visual Basic file where it makes sense for a control to belong inside another control. Examples of this are Forms, Menus, Frames, and Toolbars.

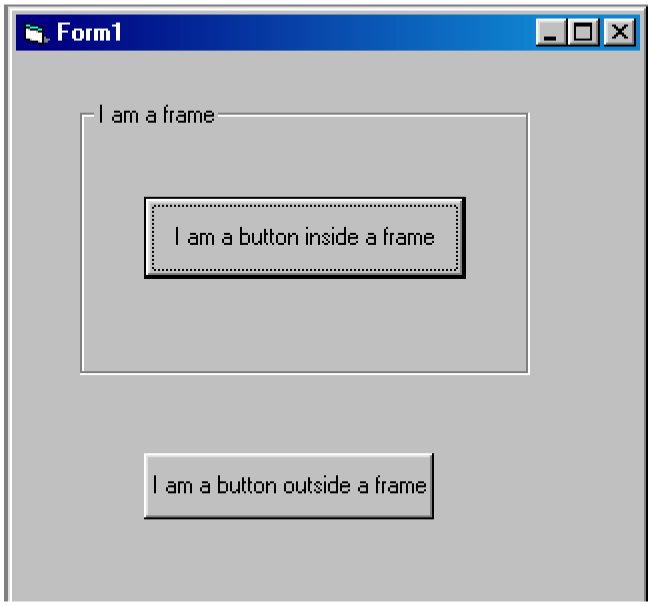


Fig. 5. An example of a frame control.

```
Begin VB.Frame Frame1
            Caption      =  "I am a frame"
            Height       =  2055
            Left         =  480
            TabIndex     =  0
            Top          =  360
            Width        =  3375
            Begin VB.CommandButton Command1
            Caption      =  "I am a button inside a frame"
            Height       =  615
            Left         =  480
            TabIndex     =  1
            Top          =  720
            Width        =  2415
End
End
```

Fig. 6. An example of frame control code

Once we are finished the control definition section, we immediately encounter the code in Fig. 7. The end tag on line 1 is used to denote the end of the form control. Following the end tag, some variables are defined that affect the way the program is compiled. For example, the Option Explicit statement ensures that all variables are declared before they are used.

```
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Option Explicit
```

Fig. 7. End of form control and Compiler Control Variables

After that, the following and final section in the file is the code section. All the functions(code blocks which return a value), sub procedures(code blocks which do not return a value), and event handlers are located there. An example of such a sub procedure can be seen in Fig. 8. The code in this section of the file appears exactly the same way it does when viewing it in the Visual Basic IDE. So for example, the code in Fig. 8 would look exactly the same whether you were viewing it from the Visual Basic form file or from the Visual Basic IDE.

```
Private Sub MCustomers_Click()
        If dbOpen = False Then
                MsgBox ("You must open the database first")
        Else
                Customer.Show 1
        End If
End Sub
```

Fig. 8. An example subprocedure

5. Implementation

During the actual execution of FormFormatter, it goes through three distinct stages. The stages are analyzing the project file, analyzing the form file, and modifying the form file. The project file need only be analyzed once. However, since there are multiple form files in a project, multiple form files need to be analyzed and modified.

After FormFormatter is started, the user must select the Visual Basic project file to format. Once they have done this, FormFormatter enters its first stage of execution, analyzing the project file. There is only one type of information in the project file that we are interested in, so we go through the file line by line looking for it. From the project file, we want to gather the filenames of the forms which belong to the project so we can record them and format them one by one. Fig. 9 contains the code used to search through the file and gather form names. The first line of Fig. 9 is the while loop which ensures that we search the entire file. The loop continues to execute until we encounter an End of Stream condition which signifies that the end of the file has been read. Through every iteration of this while loop, we read in a new line of code from the file into our working string variable line; this is performed on line 2 of Fig. 9. Line 3 is doing the actual searching for the lines we are interested in, which are lines of the form "Form=*.frm". The code in line 3 first checks for the presence of the string "Form=" in the line, and if it exists then it checks to make sure that F is the first letter in the string. If both of these conditions are satisfied, we can be fairly certain that we have encountered a form line and can safely add the form to our list. If these conditions evaluate to true for the current line of the file, Line 4 of Fig. 9 is executed and we add the form name to the list of forms. Line 4 extracts the form name by taking the characters from the 6th spot in the string until the end of the string. Starting at the 6th spot effectively drops the "Form=" from the string and gives us just the form name.

```
1.      While Not instream.AtEndOfStream
2.              line = instream.ReadLine
3.              If InStr(line, "Form=") And Mid(line, 1, 1) = "F" Then
4.                      Formlist.AddItem (Mid(line, 6, Len(line) - 5))
5.              End If
6.    Wend
```

Fig. 9. Project analyzing code example

Once we are finished this step, the user is presented with a list of all the forms belonging to the project. They can pick and choose which forms will be formatted by removing, from the list, the ones they do not want formatted. Once the user has done this, we are almost ready to begin the form analyzing stage. However, there is one small thing that needs to be completed before form analyzing can begin. FormFormatter contains two different modules. One is for analyzing normal forms, and the other is for analyzing MDI forms. Before FormFormatter can direct execution to the correct module, it must determine the type of form it is currently dealing with. The code used to perform this can be seen in Fig. 10. Much like the loop found in Fig. 9, the loop found on line 1 of Fig. 10 will continue to execute until the end of the file is encountered. However, this loop also contains two early exit condition where, if satisfied, the loop will stop immediately. The block of code in Fig. 10 increments through the file line by line looking for one of two strings. On line 3, it checks the current code line for the existence of "VB.Form". If this string is encountered, we know that the current form is a normal form, and can be processed with the normal form module. Lines 4 sets the normal form Boolean flag in this case, and line 5 exits the loop. However, if this check fails, we move to the next check on line 7, where we search the code line for the string "VB.MDIForm". If this string is encountered, we know that the current form is an MDI form, and can be processed with the MDI form module. In this case, line 8 sets the MDI form Boolean flag, and line 9 exits the loop.

```
1.      Do While Not instream.AtEndOfStream
2.              line = instream.ReadLine
3.          If InStr(line, "VB.Form") Then
4.                      normform = True
5.                      Exit Do
6.              End If
7.              If InStr(line, "VB.MDIForm") Then
8.                      isMdiForm = True
9.                      Exit Do
10.             End If
11.     Loop
```

Fig. 10. Determining the type of form

Once we have determined the type of form we currently are processing, we call the appropriate module and pass it the filename of the form to be processed. Lets assume first that a normal form was encountered. FormFormatter will call the normal form module with the name of the current form. Upon entering the normal form module, two preliminary things need to be done.

When FormFormatter finishes processing a file, it writes a tag at the beginning of the file to ensure that we are not processing files twice. On line 1 of Fig. 11, we check for this line at the beginning of the file. If it exists, we let the user know and immediately exit the module with lines 2 and 3. However, if it has not been formatted before, we want to make a backup of the file just in case FormFormatter produces undesired results. On line 5 of

Fig. 11, we copy the current form file to a file with the same name but a ".old" extension is added to the end, and it is placed in the same directory as the rest of the form files.

```
1.  If InStr(line, "'FormFormatterProcessed") Then
2.          MsgBox ("The file " & nformfile & " was already processed!")
3.          GoTo End_Sub
4.  Else
5.          filetoopen.Copy (projectpath & "\" & nformfile & ".old")
6.  End If
```

Fig. 11. Preliminary processing

Once these preliminary items have been taken care of, we can begin the form analyzing stage. In this stage, we are analyzing the file and collecting information about the controls that exist on the form. Each type of control has its own defined data type, and the information we collect on each control is stored in an array of control data types. Fig. 12 contains an example of one of the control data types. On line 1 of Fig. 12, we declare a private type called ButtonCtl, which is used for storing information about any Command Button Controls encountered in the form. There are 5 pieces of information that we are interested in storing. Line 2 contains the first piece of information, the name of the Command Button. Line 3 contains the number of the Label that FormFormatter will assign the specific command button. Line 4 stores the number of the frame that the control is stored in. If the value of the Frame variable is 0, then we know the control belongs to the main form, but if it is greater that zero it belongs to a frame on the form. Finally, lines 5 and 6 store whether or not the control is part of a control array, and if it is, the index it has been assigned in that control array.

```
1.  Private Type ButtonCtl
2.          ButtonName As String
3.          Label As Integer
4.          Frame As Integer
5.          Index As Integer
6.          IsCArray As Boolean
7.  End Type
```

Fig. 12. A sample control data type

To begin the analyzing stage, we analyze and collect information on the menus contained on the form. Menus are analyzed separately from the rest of the controls because the menu analyzing stage requires exclusive use of the stack, and cannot share it with the rest of the program. Fig. 13 contains the code used to analyze menus. The entire block, lines 1 to 29, are executed until the entire form file has been processed. On line 3, each line of code is checked for the presence of the string “VB.Menu”. If we encounter this string in a line, we know we have encountered a menu and must process it. On line 4, we push the menu line onto the stack so we know we are inside a menu control. The reason this is necessary is that main menus are the only menus we want to assign labels from the global label pool. If we encounter a menu, and the stack tells us that we are already inside a menu control, we know that the menu encountered is a submenu and we do not record any information on it. On line 5 we increase the size of the menu control array by 1. Lines 6-16 then extract the name of the menu from the line of code taken from the file. Lines 17-24 are then used to skip over any submenus inside the current main menu. These lines of code are executed until the stack is empty, which means the main menu that was pushed has been popped because we have encountered its End tag. Finally, lines 25-27 assign the recently discovered menu a label from the global label pool and increase the menu count. Once the code in Fig. 13 has completed, FormFormatter has collected the necessary information on every main menu in the form file, and we are ready to move on to the rest of the controls.

```
While Not instream.AtEndOfStream
    line = instream.ReadLine
If InStr(line, "VB.Menu") Then
        Stack.Push (line)
       ReDim Preserve menus(0 To menucount)
        Call Tokenizer.Tokenize(line, wordarray)
        menus(menucount).MenuName = wordarray(2)
    line = instream.ReadLine
        Call Tokenizer.Tokenize(line, wordarray)
        Dim menustr As String
        menustr = ""
       Dim x As Integer
        For x = 2 To UBound(wordarray)
            menustr = menustr & wordarray(x) & " "
        Next
        menus(menucount).Caption = menustr
        While Stack.IsEmpty = False
        line = instream.ReadLine
        If InStr(line, "Begin") Then
            Stack.Push (line)
        ElseIf InStr(line, "End") Then
                Stack.Pop
        End If
        Wend
    menus(menucount).Label = GlobalLabelCount
        GlobalLabelCount = GlobalLabelCount + 1
        menucount = menucount + 1
    End If
Wend
```

Fig. 13. The menu analyzing code

The rest of the analyzing step consists of a large block of code that steps through the file line by line. Fig. 14 contains a pseudocode example of this code block. For each line, a series of string tests are performed which search for controls. The other function performed inside this block of code is the tracking of frames. Much like menu analysis, the stack is used to determine whether a control is inside a frame or not, and if it is, which frame it is inside.

```
1.  repeat while not at end of file
2.      get a line of code from the file
3.      if the line denotes the start of a frame control
4.          push the frame onto the stack
5.      end if
6.      control analyzing done here
7.      if the line denotes the end of a frame control
8.          pop the frame from the stack
9.      end if
10. end repeat
```

Fig. 14. Pseudocode example of analyze code block

Line 6 of Fig. 14 represents a collection of conditional blocks. Each conditional block is designed to check for one type of control. Code is contained inside each block which collects information on a certain type of control. An example of one of these code blocks, which can be found in Fig. 15, is the block designed to handle Command Buttons. When a line containing the string "VB.CommandButton" is encountered in the file, all the other conditionals in the analyze loop will fail until the code in Fig. 15 is reached. Once that happens, the condition found on line 1 of Fig. 15 will evaluate to true, and the Command Button information collection will begin. On line 2, we increment the size of the button control array so we have room to store another button. Lines 3-5 extract the name of the button from the current line and store it in the array. Line 6 initializes the control array Boolean to false. If the control is a member or a control array this Boolean will be set to true later on in the code block. Lines 7-11 handle the case where the control is inside a frame. If the current frame number is not zero, then we are inside a frame so we store the frame the button is inside in the array. Otherwise, we set it to 0 denoting that the button belongs to the form itself. Lines 13-23 are contained inside a smaller loop. This loop increments through the button control properties until it finds the index property or until it reaches the end of the button control. The index property is not always present inside a controls code block. The presence of index means that the control is part of a control array. If this is the case, we want to record this index in the array so we can write code that is able to reference this control later on. We also set the control Boolean to true at this point. Once this loop has finished we increase the button count and exit the conditional block. Execution will continue through the remaining conditional blocks even though none of them will evaluate to true. Once this analyze loop has reached the end of the file, we are finished the code analyzing section of FormFormatter for the current form.

```
1.       If InStr(line, "VB.CommandButton") Then
2.               ReDim Preserve combuttons(0 To buttoncount)
3.               Call Tokenizer.Tokenize(line, wordarray)
4.               ctlname = wordarray(2)
5.               combuttons(buttoncount).ButtonName = ctlname
6.               combuttons(buttoncount).IsCArray = False
7.               If currentframe > 0 Then
8.                       combuttons(buttoncount).Frame = framecount
9.               Else
10.              combuttons(buttoncount).Frame = 0
11.              End If
12.              line = instream.ReadLine
13.              While(InStr(line, "End") = 0) And InStr(line,"EndProperty")=0
14.                      If InStr(line, "Index") Then
15.              Call Tokenizer.Tokenize(line, wordarray)
16.                      If wordarray(0) = "Index" Then
17.                                      ctlint = Int(wordarray(2))
18.                              combuttons(buttoncount).Index = ctlint
19.                                      combuttons(buttoncount).IsCArray = True
20.                      End If
21.              End If
22.              line = instream.ReadLine
23.      Wend
24.      buttoncount = buttoncount + 1
25.    End If
```

Fig. 15. The button control code block

Before we being the actual code writing stage, there is one preliminary step we must do first. This is due to the way FormFormatter determines when to begin writing the code. Like the analyze stage, the writing stage steps through the code line by line until it encounters one of the lines it is looking for. In the case of this stage, those lines are either Sub procedure headers or Menu Control declarations. However, some of the sub procedure headers might not have been declared in the file. In this case, we need to write them in so that the code writing stage knows where to begin inserting code. The preliminary step consists of two parts. The first part is scanning the file for the existence of these headers. The code used to do this can be seen in Fig. 16. This code scans goes through the file line by line until it reaches the end. While it is incrementing through the file, it checks for the existence of the strings in line 3 and line 6. If either of these strings are found, their corresponding Boolean flag is set to true. The writing step, which can be viewed in Fig. 17, is even more simple than this. The file is simply opened in append mode. If either of the flags were set to true in the search, their corresponding sub procedure header is appended to the file.

Once this preliminary step is done, we are ready to begin the code writing stage.

```
While Not instream.AtEndOfStream
        codeline = instream.ReadLine
        If Not InStr(codeline, "Form_Load") = 0 Then
                formload = True
End If
        If Not InStr(codeline, "Form_KeyDown") = 0 Then
                keydown = True
End If
Wend
```

Fig. 16. Code used to search for sub procedure headers

```
If formload = False Then
        outstream.WriteLine ("Private Sub Form_Load()")
outstream.WriteLine ("End Sub")
End If
If keydown = False Then
outstream.WriteLine ("Private Sub Form_KeyDown(KeyCode
                As Integer, Shift as Integer)")
        outstream.WriteLine ("End Sub")
End If
```

Fig. 17. Code used to write the subprocedure headers

The code writing stage can be broken up into three distinct blocks. Each block is contained within the same loop that goes through the file, and every time a string is encountered with which one of the blocks is interested, it enters that block of code. Lines 2-4 of Fig. 18 represent the block of code which works with the Form_Load procedure. Inside the Form_Load procedure, we are inserting code which adds labels to all the controls we identified in the analyze stage. Lines 5-7 represent the code block which works with the KeyInput_KeyPress procedure. This procedure is the event handler which is called when a label is entered into the special input box. Finally, lines 8-10 are used for handling menu controls. The reason menu controls need special treatment is that we have to actually add their labels into the text of the menu item. Once this writing loop has incremented through the entire file, the writing stage is finished, and FormFormatter has completed one form.

```
1.  Repeat until end of file
2.      If Form_Load sub procedure is encountered
3.          Write label generation code
4.      End if
5.      If KeyInput_Keypress is encountered
6.          Write case statement for control handling
7.      End if
8.      If menu is encountered
9.          Add label to the menu text
10.     End if
11. End repeat
```

Fig. 18. Pseudocode overview of writing stage

The Form_Load sub procedure is the first block we encounter in the code. Once the Form_Load header has been detected, we enter its block. The block of code simply checks every supported control count to see if the analyze stage found any controls. For each control that FormFormatter found, we write a label for it. Each different type of control has its own block of code for writing its labels. Fig. 19 contains the code for writing these labels for command buttons. Lines 2-10 write the first line of code for the label. This line is used to create the label, and tell visual basic which object the label belongs to. This is where it becomes important that we know if the current control is inside a frame or not. On line 4, we check to see if the control is in fact inside a frame. If it is not, we simply create the label inside "Me" which is visual basics term for the form itself. If it is however, we get the frame number that the current control is inside, and use that number to retrieve that frame's name from the frame control array. We then use its name to create the label inside that frame. Lines 11-28 are used for writing the second line. The purpose of the second line is the actual position of the label on the form. We want the label to beside it's corresponding control, so we write code to do this. However, we must first check if the control is part of a control array. The reason we do this is due to the way we calculate the offset of the controls. In order to position the controls, the code line we write makes use of the .Top, .Width, .Height, and .Left properties of a control. If we wanted to reference these properties on a normal control, we would simply use a line like "CommandButton1.Height". This statement would give us the height of the control. However, if this control is part of a control array it does not work this way. We must tell the control array the index of the particular control that we are interested in. This is why we needed to record this index in the analyze stage. The code would then have to be written as "CommandButton1(14).Height". This code would give us the height of the 14th control in that particular control array. Lines 29-30 simply write the line that sets the color of the current label. The color is set in hexadecimal, where there are six digits. Each pair corresponds to a primary color component. In our case, we have used the color 0000FFH which is red. Line 31-32 write the code that assigns the value to the label. We take a label from the global label pool, and then increment the global label pool to prepare for the next label. In this way, we ensure that each controls label is unique. That is all that needs to be done to write a single label. Once FormFormatter has done this for every control discovered in the analyze stage, it exits this block and continues the code writing stage.

```
1.      For i = 1 To buttoncount
2.              codeline = "set ctlabel = Controls.Add(""VB.Label"", "
3.              codeline = codeline & """" & "blabel" & i & """"
4.           If combuttons(i - 1).Frame = 0 Then
5.                      codeline = codeline & ", Me)"
6.                      outstream.WriteLine (codeline)
7.           Else
8.                codeline=codeline &", "&frames(combuttons(i -
                        1).Frame).framename")"
9.                outstream.WriteLine (codeline)
10.          End If
11.        If combuttons(i - 1).IsCArray = True Then
12.                     leftindex = combuttons(i - 1).ButtonName & "(" & _
13.               combuttons(i - 1).Index & ")" & ".left + " _
14.               & combuttons(i - 1).ButtonName & "(" & _
15.               combuttons(i - 1).Index & ")" & ".width + 100"
16.               topindex = combuttons(i - 1).ButtonName & "(" & _
17.               combuttons(i - 1).Index & ")" & ".top + " _
18.               & combuttons(i - 1).ButtonName & "(" & _
19.               combuttons(i - 1).Index & ")" & ".height - 200"
20.             Else
21.                     leftindex = combuttons(i - 1).ButtonName & ".left + " & 22.
        combuttons(i - 1).ButtonName & ".width + 100"
23.               topindex = combuttons(i - 1).ButtonName & ".top + " & 24.
        combuttons(i - 1).ButtonName & ".height - 200"
25.        End If
26.        codeline = "ctlabel.Move " & leftindex & ", "
27.        codeline = codeline & topindex & ", 200, 200"
28.        outstream.WriteLine (codeline)
29.        codeline = "ctLabel.ForeColor = &HFF"
30.        outstream.WriteLine (codeline)
31.        codeline = "ctlabel.Caption = """ & GlobalLabelCount & """"
32.        outstream.WriteLine (codeline)
33.        codeline = "ctlabel.Visible = True"
34.        outstream.WriteLine (codeline)
35.        combuttons(i - 1).Label = GlobalLabelCount
36.        GlobalLabelCount = GlobalLabelCount + 1
```

Fig. 19. Label generation code for command buttons

The second block encountered in this stage is the Keyinput_Keypress writing block. Once the Keyinput_Keypress header has been detected, we enter its block. For every label that we have written, this block of code writes a case for handling when that label is entered in the special input box. This block is similar to the label writing block in that it has code for handling each type of control. Fig. 20 contains the code for handling command buttons. For each command button discovered in the analyze stage, we need to write a case to handle it here. Lines 3-4 write the actual case header, with the current controls label as the case (ie "Case 13" where 13 is the label of the current control). Lines 5-12 write the line of code where we actually handle the case when this control was selected. Again we must check if the control is part of a control array, and act accordingly. If the command button is not part of a control array, then we simply call the command buttons click event handler. The line of code would look something like "CommandButton1_Click". However, if it is part of a control array, then the code needs to reference the index of the current control in the control array. The code would then look like "CommandButton1_Click(14)" where 14 is the index of the desired control. Once FormFormatter has written a case for each control discovered, it exits this block of code.

```
1.      If Not buttoncount = 0 Then
2.          For i = 0 To buttoncount - 1
3.              codeline = "Case " & combuttons(i).Label
4.              outstream.WriteLine (codeline)
5.             If combuttons(i).IsCArray = True Then
6.              codeline = combuttons(i).ButtonName & "_Click"
7.              codeline=codeline&"("&combuttons(i).Index&")"
8.                  outstream.WriteLine (codeline)
9.              Else
10.                 codeline = combuttons(i).ButtonName & "_Click"
11.                 outstream.WriteLine (codeline)
12.           End If
13.         Next
14.     End If
```

Fig. 20. Command button case code

The final block of code encountered in the code writing stage is the Menu handling block. This block is necessary because labeling the menus means changing the text of the actual menu caption. Fig. 21 contains the menu labeling code. When we encounter a menu, we immediately enter this block. The first thing done, on lines 1-16, is we check to see if the menu encountered is a main menu. To do this, we loop through all the discovered main menus in the menu array. If it is, we place its label in the beginning of the label caption. For example, a menu name “File” would become “2:File” where “2” is the label assigned to that menu. On lines 13-14 we set flags which indicate that the menu is a main menu, and that it hasn’t been processed yet. We do this because sub menus are not assigned labels from the global label pool. They instead get assigned a single digit from a counter, which is reset for each new menu. On lines 17-32 we handle the case where the menu is a sub menu. On line 18-20, we check to see if this a new menu. If it is, we reset the sub menu label count back to 0. Once FormFormatter has handled the currently encountered menu, it exits the menu block and continues execution until it encounters the next menu.

```
1.    For i = 0 To menucount - 1
2.            If InStr(line, Trim(menus(i).Caption)) Then
3.                    Call Tokenizer.Tokenize(line, wordarray)
4.                    Dim menustr As String
5.                menustr = ""
6.                Dim x As Integer
7.                For x = 2 To UBound(wordarray)
8.                        menustr = menustr & wordarray(x) & " "
9.                    Next
10.                   line = "Caption = """ & menus(i).Label & ":"
11.                   line = line & Mid(menustr, 2, Len(menustr))
12.                   outstream.WriteLine (line)
13.                   ismainmenu = True
14.           newmenu = True
15.           End If
16.   Next
17.   If ismainmenu = False Then
18.           If newmenu = True Then
19.                   newmenu = False
20.                   SubMenuLabelCount = 0
21.       End If
22.       Call Tokenizer.Tokenize(line, wordarray)
23.       Dim tempstr As String
24.       tempstr = ""
25.       Dim z As Integer
26.       For z = 2 To UBound(wordarray)
27.                   tempstr = tempstr & wordarray(z) & " "
28.           Next
29.           line = "Caption = """ & SubMenuLabelCount & ":"
30.           line = line & Mid(tempstr, 2, Len(tempstr))
31.           outstream.WriteLine (line)
32.           SubMenuLabelCount = SubMenuLabelCount + 1
33.   End if
```

Fig. 21. Menu labeling code

All of the above example were given assuming that FormFormatter was dealing with a normal form. If it was dealing with an MDI form, much of the processing would be quite similar. The MDI module processes menus in the same was as the normal module. However, the MDI form does not analyze other controls or label them. Only menus are of interest in an MDI form. The only other difference is that instead of inserting a textbox control for the special input box, the MDI module inserts a toolbar with a textbox on it.

6. Problems and Solutions

During the coding of the form formatter, a number of problems were recognized and overcome. Some of these problems were quite trivial, while others were perplexing and their solutions are worth noting.

The first of these problems encountered was that of extracting the data properly from each line in the source code. The first approach taken was to use visual basic string parsing functions like Right(), Left() and Mid(). The reason this approach failed was that it was based on the assumption that the same amount of spaces and indentations will exist on every line of the file. This unfortunately need not be true and isn't true in some cases. The solution was to use a string tokenizer for this task. The string tokenizer breaks up the line into words. In this case, words are defined as strings of alphanumeric characters separated by spaces. At the same time, the tokenizer also removes any whitespace from the beginning and end of the line. The tokenizer solved the problem of the varying indentation.

The next problem was encountered while trying to label controls which were contained inside a frame. Visual basic considers a frame to be a control container. Whenever an attempt was made to place a label beside an object inside that frame, the label would not appear on the form at all. The cause and solution to this problem came from two different areas. The first area that was causing the problem was that in order for a label to appear on a frame, it has to belong to that frame. When you instantiate a label in the code, you have to specify that it is being created inside that frame in order for it to appear on that frame. This idea leads us to the next problem area. When you are looking at the location variables of an object(top and left), they are given in units relative to the top left corner of the form. However, when you are looking at or changing the location of an object inside a frame, these values are relative to the top left corner of the frame and not the form.

Another interesting problem encountered dealt with intercepting keystrokes. Originally, the home key on the keyboard was used to return focus to the key input box. This functionality was supposed to work no matter what control had focus at the time. The forms key press handler would intercept the home key keystroke and send the focus to the input box. However, some types of controls, like a Microsoft flexgrid, do not allow the form to intercept a home key keystroke when they have focus. The solution to this problem was simply defining another key as the focus return key. The escape key was chosen.

7. Capabilities

Form Formatter is capable of handling a number of different situations encountered in visual basic source code. It is capable of handling many different types of controls. It is also capable of handling normal and mdi forms.

When speaking about controls, form formatter can handle every default visual basic control that is available in the IDE. It can also handle a few others. The control types that it supports include command buttons, text boxes, list boxes, combo boxes, scroll bars, option buttons, check boxes, menus, basic data sources, and the Microsoft Flexgrid control.

Form formatter is also capable of processing both normal and MDI forms. When processing a normal form, the labels and key input box are simply added to the form, and it simply enumerates all the controls on the form. Fig. 22 is an example of a normal form.

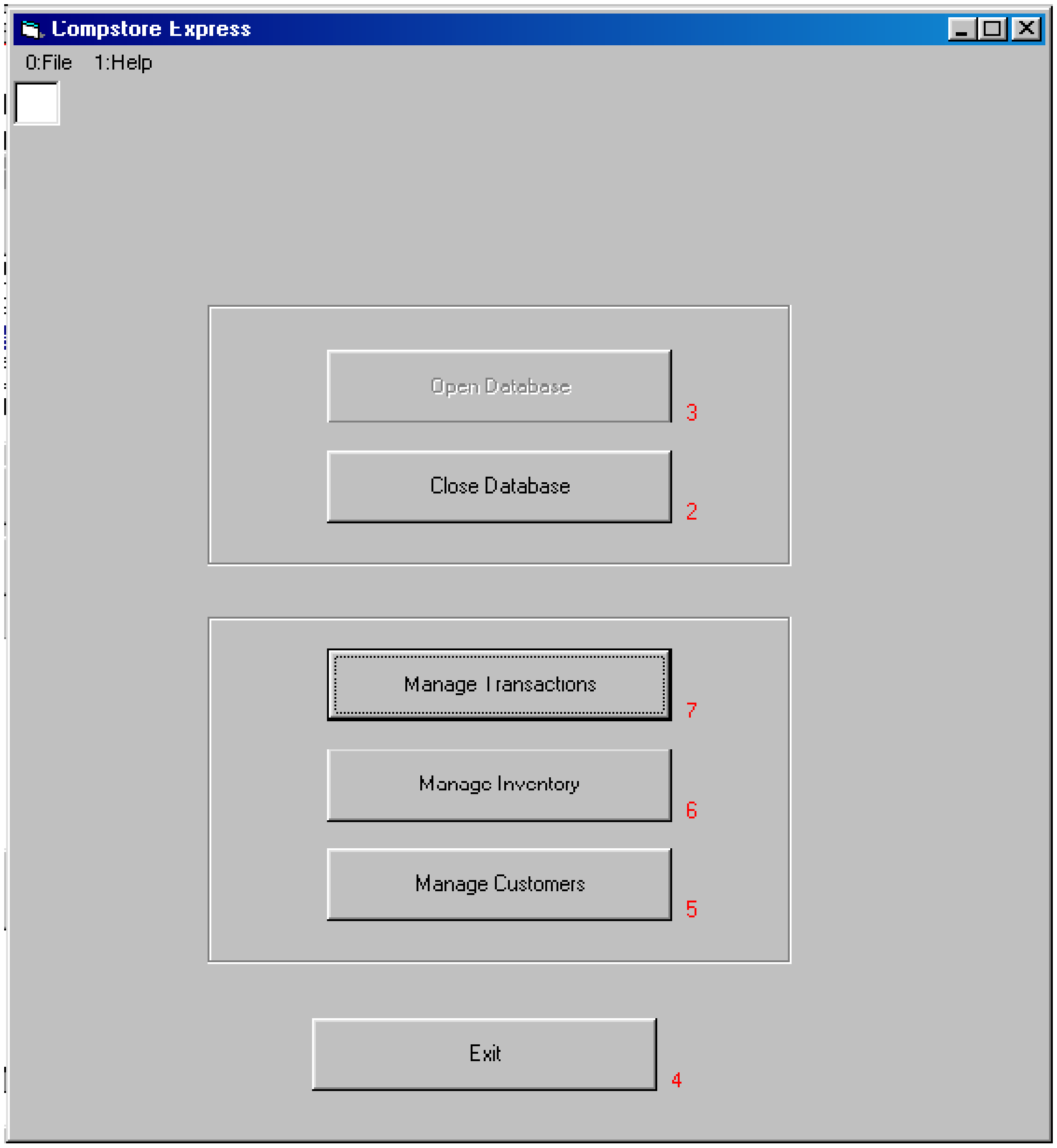


Fig. 22. A normal form

However, MDI forms need to be processed a little differently. An MDI form is simply a container for other forms. No controls exist on this type of form, and no controls can be added to this type of form. Fig. 23 is an example of an MDI form. We handle this situation by placing a toolbar on the mdi form, and placing the special input box on the toolbar. In this way, the form has a toolbar which contains the special input text box which accepts commands for manipulating the menus. Menu items are the only things that this special input box needs to handle because no other controls can exist on an MDI form.

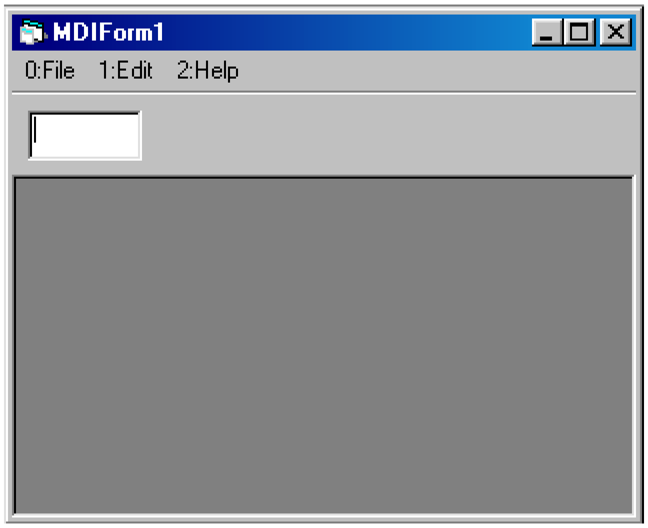


Fig. 23. An MDI form

8. Limitations

Many problems were encountered during the development of form formatter. Some of these problems had solutions. However, many of them turned out to be unsolvable, and are worth discussing.

Control support is one area where problems arise. Although FormFormatter supports a number of different types of controls, there are many that it does not. For each type of control that it needs to support, a significant amount of code is required. Given the number of different controls, this is not a practical situation. The problem becomes exponentially worse when you consider user defined active x controls. This problem could be managed through the use of signature files. Each signature file would contain directions for form formatter to recognize a certain type of control. For example, the signature could contain a string for formatter to use to recognize the control, a list of the properties that the application should collect on that control, and an example of the type of code necessary to handle that control when it is selected in the application. The signature files could then be loaded during runtime, and FormFormatter would load all of the control recognition strings into an array. It could increment through the array every time a control is encountered to determine if the current control is supported. If the control is supported, it could then go back to that controls signature file to determine how to handle the control. Fig. 24 is a possible example of the format of a signature file. The first line would be loaded into the array for control recognition. Line 2 would be treated as a series of binary switches, where each switch signifies a type of data to look for on the control. Line 3 would be used when the FormFormatter is writing the code to handle the current button. It could recognize the <> tags and insert the current controls name in its place. Adding support for a new type of control would be as simple as adding another signature file to the group. The idea of a signature file would only reduce the workload however. It is not a solution because it does not address the large volume of control type possibilities.

```
1.      Control Recognition String = Begin VB.CommandButton
2.      Required Data = 1 0 0 1 1 0 1
3.      Default Action = <ButtonName>_Click
```

Fig. 24. Possible format for signature file

In order for FormFormatter to intelligently modify source code, it needs to make certain assumptions about the structure of the source code. This need to make assumptions turns out to be a big drawback. An example of this situation is the section of code that adds the special input textbox to the form. After viewing many visual basic source code files, you will notice that the line “StartupPosition 3” is consistently the last line before the start of the control section. This line is used to notify the Visual Basic compiler where the user would like the application to appear on the screen when it starts. Since it seems like this line is always in the same spot after looking at many different Visual Basic form files, it would make sense to add the textbox immediately after this line. This method works quite well until the moment when you encounter a source file that does not contain this line. When that happens, the key input textbox does not get added and FormFormatter fails to work correctly. On one occasion when testing FormFormatter, this exact situation was encountered. It is not known why the line was missing from the file. This is one example from the limitless number of situations where assumptions cannot be made. Source code is an extremely volatile thing. When assumptions like this are made, there is no guarantee that the program will consistently work correctly.

Form formatter relies heavily on the ability to recognize strings inside of larger strings. This works correctly most of the time, but does have the potential to fail. If the source code contains strings which are in the wrong context but also match the strings we are looking for, we will get incorrect results. For example, form formatter maintains a stack which tracks the depth of the begin/end tag hierarchy. When a line contains the string "Begin" the line is pushed onto the stack. However, if a control name is also "Begin", then that line will incorrectly get pushed to the stack. This situation has the potential of occurring many times, and can cause seriously undesired results in the code. To combat this problem, string recognition rules could be made much more strict. An example could be a rule specifying that "Begin" must be the first word in the line in order for it to pass the recognition. Taking this idea, many specific conditions could be worked out and implemented into the code in the form of complex conditionals. This does not form a complete solution to the problem, but it would reduce the occurrence of false recognition significantly.

Form formatter places labels on the edge of the controls. This works quite well on a form where all of the controls are well spaced and organized. If form formatter processes a form that has crowded controls, the labels have a chance of being partially covered or not appearing at all. A solution to this problem would be to draw the labels directly on the controls. It doesn't seem like there is an easy way of doing this. A possible but not necessarily easy solution to this problem would involve the use of the Windows API. Using Windows Messages, Window Handles, and other API specific things, a programmer could alter the way Windows actually draws the forms so that labels could be drawn directly on the control.

In order to achieve the desired results, form formatter needs to define special key handlers for some controls, and for the form itself. This poses a problem when the original author of the code has already defined key handlers for these controls. After being processed by form formatter, this code will contain redefinitions of key handlers and will fail to run. A solution to this problem could be a scan for the presence of key handlers. If key handlers already exist for certain controls, the user of FormFormatter would be notified and required to change the way their code works before FormFormatter proceeds.

9. Form Formatter Tutorial

Form formatter is a tool which allows a programmer to quickly and easily add a keyboard-based input strategy automatically. Form formatter analyzes the code written by the programmer and intelligently modifies it to support keyboard-based interaction. FormFormatter is simple and very easy to use. This short tutorial is all you need to get you started on your way to mouse free living. Once you start FormFormatter, you will be presented with the screen shown in Fig. 25.

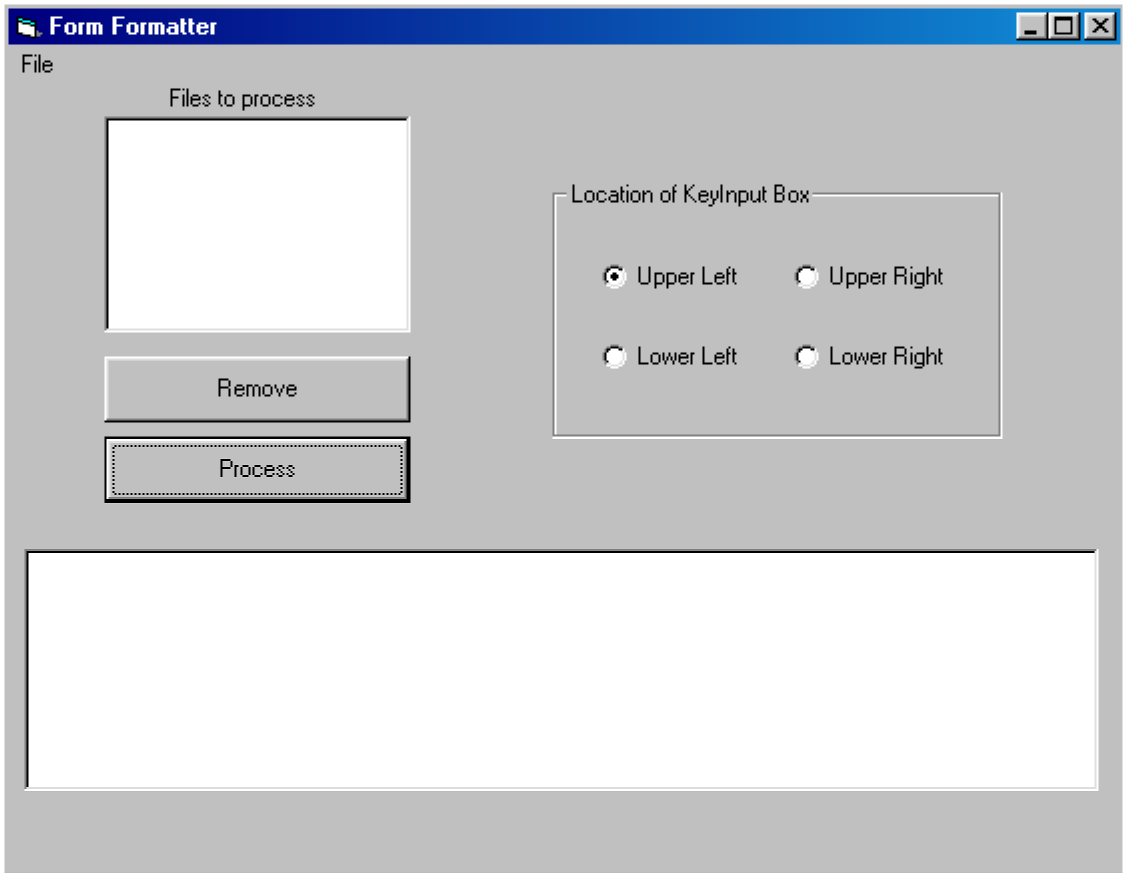


Fig. 25. The form formatter user interface.

After starting FormFormatter, the first thing you will want to do is open the visual basic project file of the program you want to format. To do this, you must click file, then open, and navigate to the directory containing this project file. Once there, highlight the file, and click open. This screen can be seen in Fig. 26.

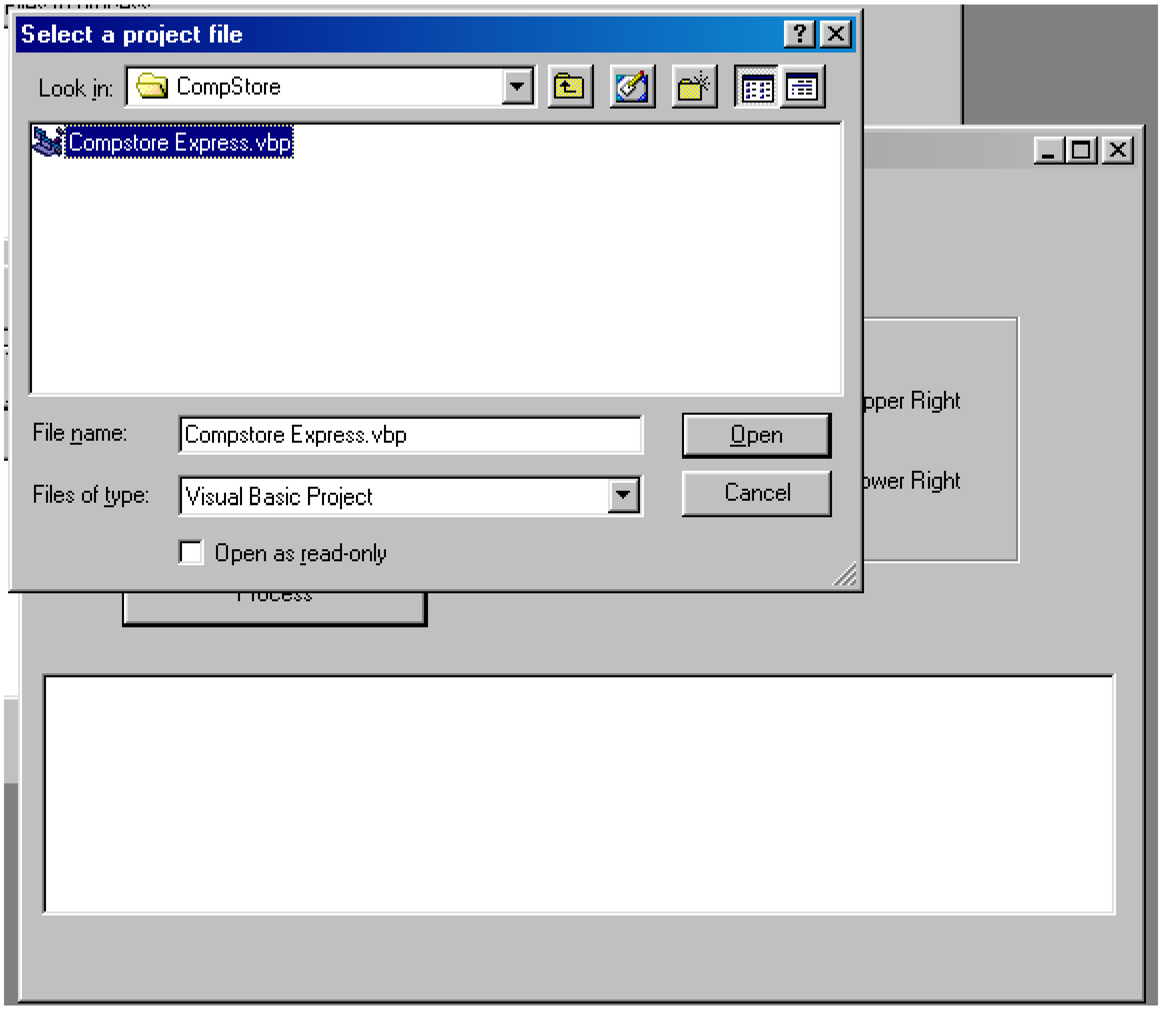


Fig. 26. Form formatter only displays files with the vbp extension.

After clicking open, the file dialog will close and you will be back at the main form again. On the main form there is a list box with the title Files To Process directly above it. After the project file is opened, this list box will contain the name of every form that belongs to that project. Look over the list and make sure that you want every file to be processed by form formatter. If the list contains any form file which you do not want processed, you can simply highlight the form by clicking it with the mouse, and click remove to remove it from the list. In Fig. 27 you can see the file DeleteCustomer.frm highlighted and ready to be deleted.

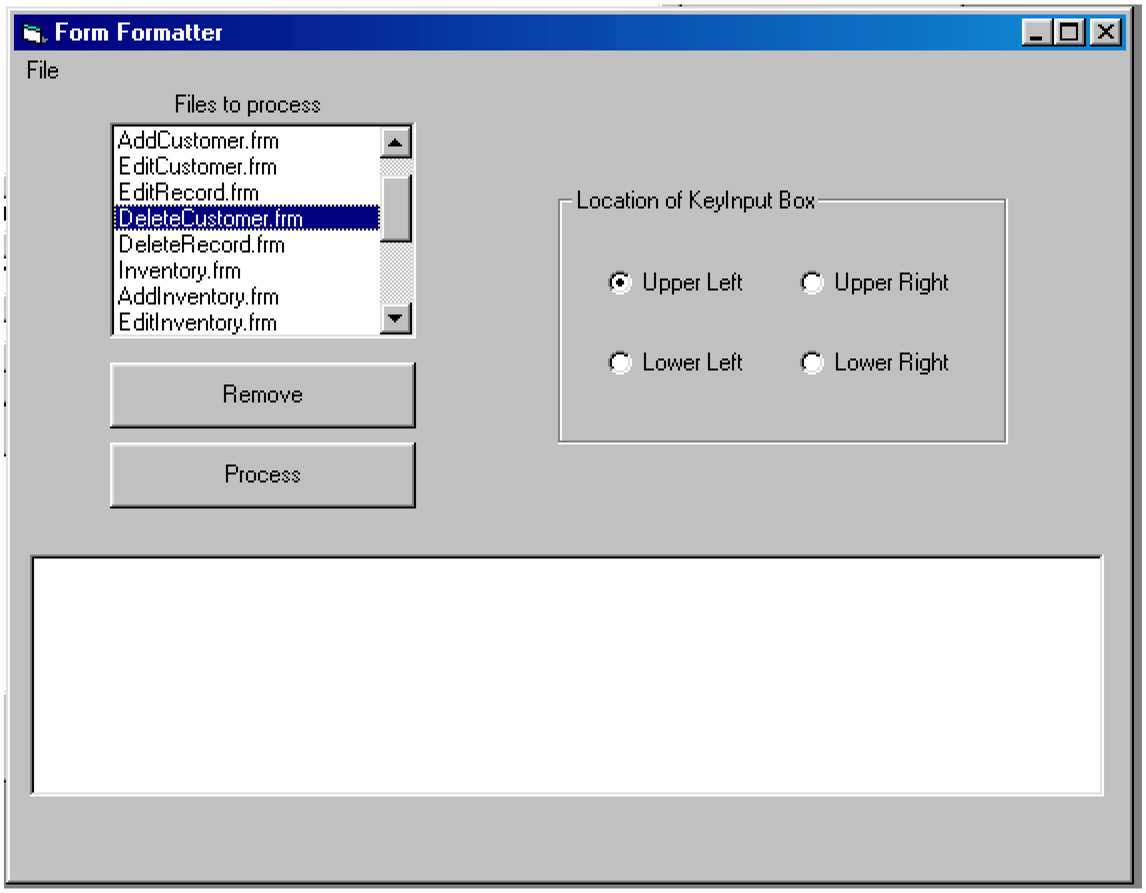
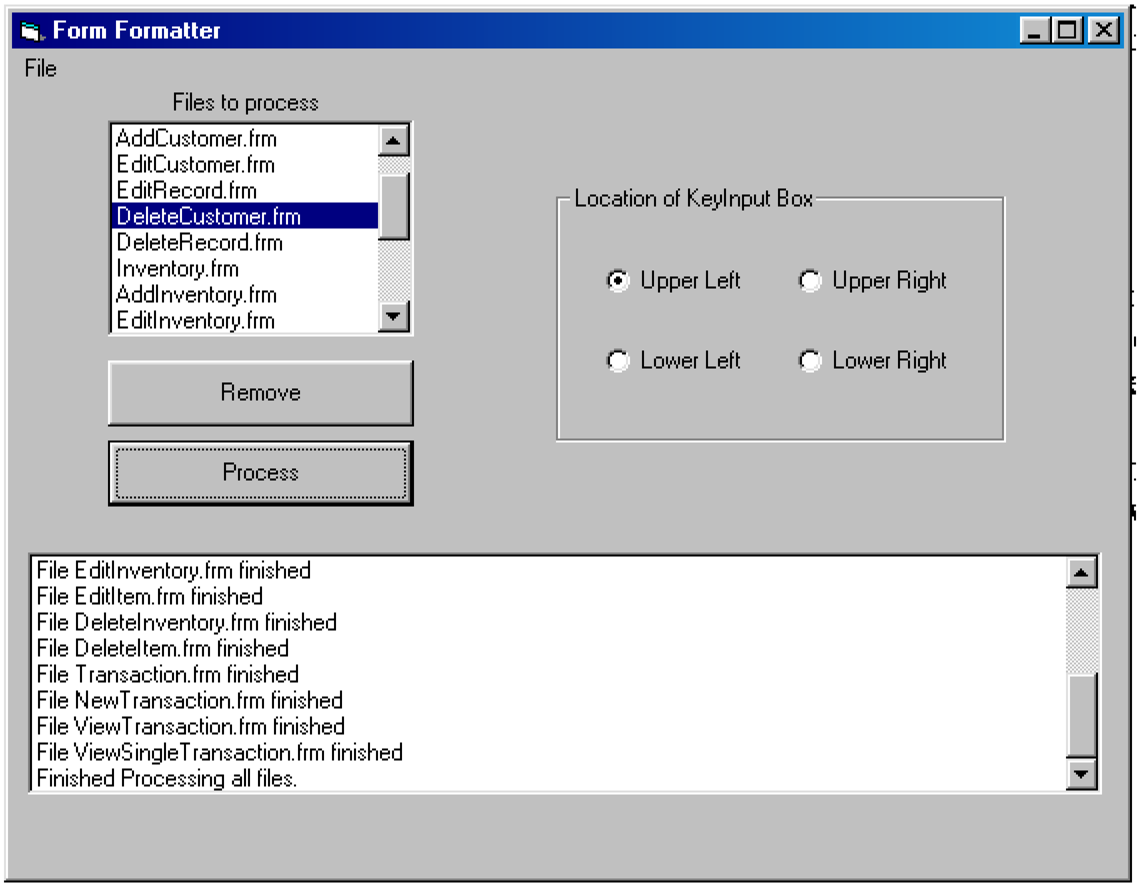


Fig. 27. We don't want to process deletecustomer.frm, so we highlight it and click remove.

The next thing you want to do is decide on the location of the keyinput box on the forms. There are four locations you can choose from, and the desired location can be selected by clicking on its corresponding option button. Once you have done this, you are ready to process the forms. To do this, simply click on the process button. The status window under the process button will notify you when each form is finished, and then finally when all the files are finished. Once this happens, you are finished with the form formatter code formatting experience. You can see a populated status window in Fig. 28.



AddCustomer.frm
EditCustomer.frm
EditRecord.frm
DeleteCustomer.frm
DeleteRecord.frm
Inventory.frm
AddInventory.frm
EditInventory.frm

[✓ Upper Left]

```
File EditInventory.frm finished
File EditItem.frm finished
File DeleteInventory.frm finished
File DeleteItem.frm finished
File Transaction.frm finished
File NewTransaction.frm finished
File ViewTransaction.frm finished
File ViewSingleTransaction.frm finished
Finished Processing all files.
```

Fig. 28. The status box tells us every file that was processed, and when it is finished completely.

There are a few error messages that might appear in the status box. If the status box says “There is no project open to analyze” it means there is currently no open project. In this case, an error might have occurred while opening the project file or the project file might be corrupt. If you see the error message "Error, please select a file!", it means you have not opened a project file before clicking the process button. If you see "An error has occured. FormFormatter cannot determine the file type of file <file here>. Please ensure it is a valid form file" it means that one of the form files could not be recognized properly. The form file might be from an incorrect version of visual basic, or it might not be a form file at all. Finally, if the status box says "Please select an item to remove" it means you have clicked the remove button without selecting a form to remove from the list.

After form formatter has finished, there are some places in the source code that might need attention. If some of the labels on the form do not appear properly, then the Form_Load procedure is where you want to look. Each of the labels gets their location from the control that they belong to. If you want to find out which label needs fixing, simply look for the lines of code that contain the name of the control in question. A sample of a piece of label code can be seen in Fig. 29. In this piece of code, we are adding the label “1” to a control called AddButton. The location of the label is set with line 2. The .Move function takes 4 arguments. The first argument is the horizontal location on the form. The second argument is the vertical location. The last two are the height and width of the control. The color of the label is set with line 3. Finally, the actual text of the label is set with the 4th line.

```
1.          set ctlabel = Controls.Add("VB.Label", "blabel2", Me)
2.         ctlabel.Move AddButton.left + AddButton.width + 100, AddButton.top
             + AddButton.height - 200, 200, 200
3.         ctLabel.ForeColor = &HFF
4.         ctlabel.Caption = "1"
```

Fig. 29. A sample of label code

If you are unsatisfied with the default action performed on a certain control, then you need to look at the Keyinput_Keypress procedure. In this procedure, there is a large case statement which handles the input in the textbox. Simply make note of the label on the control which needs attention, and look for the corresponding case in the case statement. A sample of some code from this case statement can be seen in Fig. 30. If 1 was entered into the Keyinput textbox, then case one would be executed. This would mean that on line 2, AddButton_Click procedure would be called which would emulate that button being clicked. If any of the default operations is unsatisfactory to you, you can simply change the behaviour by changing the lines of code following the case declaration.

```
1.         Case 1
2.                    AddButton_Click
3.         Case 2
4.                    Category.SetFocus
5.                    SendKeys ("{F4}")
6.         Case 3
7.                    KeyInput.SetFocus
```

Fig. 30. A sample case statement

Finally, if you need or want to designate a key other than the ESC key as the focus return, then you will want to look in the Form_Keydown function. The code currently waits for keycode 27 (the ESC Key) but a different key can be designated by simply changing the number on line 2 of Fig. 31. The code in Fig. 31 is fairly straightforward. The only thing that might seem strange is the code on line 3. This is necessary because we don't want any default key handlers handling the ESC key keystroke.

```
1.      Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
2.              if Keycode = 27 then
3.                      keycode = 0
4.                      KeyInput.SetFocus
5.              End If
6.              End Sub
```

Fig. 31. An example of the focus return key handler

10. Conclusion

FormFormatter has the capability of saving a programmer a lot of time when adding a keyboard input strategy to an application. It automates many mundane and tedious tasks, and can do these tasks quickly. However, it is not a complete and flawless solution. Source code is something that is very personal, variable, and unique. Given this and the problems we encountered, form formatter is not a 100% accurate solution. There is still much work to do after running form formatter, and in some cases it can actually make more work for the programmer if it seriously fails.

11. References

[1] U.S. Department of Labor – Bureau of Labor Statistics / Nonfatal occupational illnesses by category of illness, private industry, 1996-2000 http://www.bls.gov/iif/oshwc/osh/os/ostb0995.txt

[2] A. Hedge, T. Muss, and M. Barrero. *Comparitive Study of Two Computer Mouse Designs* - Department of Design and Environmental Analysis, Cornell University. Ithaca, NY. 1999