To create a tool that extracts VB6 functions and subs from source files into individual `.bas` files organized by the original file name, follow this step-by-step implementation using **C#** (ideal for file I/O and regex parsing).

Tool Overview

- **Input**: A directory containing VB6 `.bas` files.
- **Output**: A structured directory where each original file becomes a subdirectory containing extracted functions/subs as individual `.bas` files.
- **Features**: Parses standard `Sub` / `Function` procedures, preserves full code (including parameters/comments), and handles nested directories.

Implementation Code

1. Project Setup

Create a new C# Console App (.NET Framework/.NET Core) and replace the default code with the following:

```
using System;
using System.IO;
using System.Linq;

namespace Vb6ProcedureExtractor
{
    class Program
    {
        static void Main(string[] args)
        {
            // Parse command-line arguments
            string inputDir = null, outputDir = null;
            for (int i = 0; i < args.Length; i++)
            {
                switch (args[i].ToLower())
                {
                    case "--input":
                        inputDir = args[++i];
                        break;
                    case "--output":
                        outputDir = args[++i];
                        break;
                }
            }

            // Validate arguments
            if (string.IsNullOrEmpty(inputDir) || string.IsNullOrEmpty(outputDir))
            {
                Console.Error.WriteLine("Usage: Vb6ProcedureExtractor --input
<INPUT_DIR> --output <OUTPUT_DIR>");
                return;
```

```
            }
            if (!Directory.Exists(inputDir))
            {
                Console.Error.WriteLine($"Error: Input directory '{inputDir}' not
found.");
                return;
            }

            // Create output directory (if missing)
            Directory.CreateDirectory(outputDir);

            // Process all .bas files recursively
            var basFiles = Directory.EnumerateFiles(inputDir, "*.bas",
SearchOption.AllDirectories);
            foreach (var file in basFiles)
            {
                ExtractProcedures(file, outputDir);
            }

            Console.WriteLine("Extraction complete!");
        }

        /// <summary>
        /// Extracts all Subs/Functions from a VB6 file and saves them to the
output directory.
        /// </summary>
        private static void ExtractProcedures(string sourceFile, string
outputRoot)
        {
            // Get the base name of the source file (e.g., "MathUtils.bas" →
"MathUtils")
            string sourceFileName = Path.GetFileNameWithoutExtension(sourceFile);

            // Create a subdirectory for this file's procedures
            string targetSubdir = Path.Combine(outputRoot, sourceFileName);
            Directory.CreateDirectory(targetSubdir);

            // Read all lines of the VB6 file
            var lines = File.ReadAllLines(sourceFile).ToList();
            int lineCount = lines.Count;
            int currentLine = 0;

            while (currentLine < lineCount)
            {
                string line = lines[currentLine].Trim();

                // Skip empty lines or comments (lines starting with ')
                if (string.IsNullOrWhiteSpace(line) || line.StartsWith("'"))
                {
                    currentLine++;
                    continue;
                }

                // Check for Sub/Function declaration using regex patterns
                const string subPattern =
@"^\s*(Public|Private|Friend)\s+Sub\s+(\w+)\s*$.*?$\s*$";
                const string funcPattern =
@"^\s*(Public|Private|Friend)\s+Function\s+(\w+)\s*$.*?$\s*As\s+\w+\s*$";
```

```
                if (System.Text.RegularExpressions.Regex.IsMatch(line,
subPattern))
                {
                    // Capture Sub name (second group in regex)
                    string procName =
System.Text.RegularExpressions.Regex.Match(line, subPattern).Groups[2].Value;
                    SaveProcedure(lines, currentLine, "Sub", procName,
targetSubdir);
                }
                else if (System.Text.RegularExpressions.Regex.IsMatch(line,
funcPattern))
                {
                    // Capture Function name (second group in regex)
                    string procName =
System.Text.RegularExpressions.Regex.Match(line, funcPattern).Groups[2].Value;
                    SaveProcedure(lines, currentLine, "Function", procName,
targetSubdir);
                }
                else
                {
                    // Not a procedure start—move to next line
                    currentLine++;
                }
            }
        }

        /// <summary>
        /// Saves a complete procedure (from start to matching End Sub/Function)
to the target directory.
        /// </summary>
        private static void SaveProcedure(List<string> lines, int startLine,
string procType, string procName, string targetDir)
        {
            int lineCount = lines.Count;
            var procedureLines = new System.Collections.Generic.List<string>();
            int indentLevel = 0; // Track nested blocks (e.g., If/Else) to find
the correct End statement
            bool endFound = false;

            for (int i = startLine; i < lineCount; i++)
            {
                string originalLine = lines[i];
                procedureLines.Add(originalLine);

                // Update indent level for nested blocks (adjust based on VB6
indentation conventions)
                string trimmedLine = originalLine.Trim();

                // Increase indent for block starts (e.g., If, For, Function)
                if (trimmedLine.StartsWith("If") || trimmedLine.StartsWith("For")
||
                    trimmedLine.StartsWith("While") || trimmedLine.Equals("Then:",
StringComparison.OrdinalIgnoreCase))
                {
                    indentLevel++;
                }

                // Decrease indent for block ends (e.g., End If, Next, Wend)
```

```
                if (trimmedLine.StartsWith("End If") ||
trimmedLine.StartsWith("Next") ||
                    trimmedLine.StartsWith("Wend") || trimmedLine.Equals("Else:",
StringComparison.OrdinalIgnoreCase))
                {
                    indentLevel--;
                }

                // Check for the matching End statement (account for indentation)
                if (procType == "Sub" && originalLine.Trim().StartsWith($"End
Sub"))
                {
                    endFound = true;
                    break;
                }
                else if (procType == "Function" &&
originalLine.Trim().StartsWith($"End Function"))
                {
                    endFound = true;
                    break;
                }

                // Handle nested procedures (ignore inner Subs/Functions)
                const string innerProcPattern =
@"^\s*(Public|Private|Friend)\s+(Sub|Function)\s+";
                if (System.Text.RegularExpressions.Regex.IsMatch(originalLine,
innerProcPattern))
                {
                    indentLevel++; // Treat inner procedures as part of the
current block
                }
            }

            if (!endFound)
            {
                Console.Error.WriteLine($"Warning: No matching End {procType}
found for '{procName}' in file.");
                return;
            }

            // Save the procedure to a new .bas file
            string targetFilePath = Path.Combine(targetDir, $"{procName}.bas");
            File.WriteAllLines(targetFilePath, procedureLines);
        }
    }
}
```

Key Features Explained

1. Regex Pattern Matching

- **Sub Detection**: `^\s*(Public|Private|Friend)\s+Sub\s+(\w+)\s*$.*?$\s*$`
  Matches `Sub` declarations with optional access modifiers (Public/Private/Friend) and captures the procedure name.

- **Function Detection**: `^\s*(Public|Private|Friend)\s+Function\s+(\w+)\s*$.*?$\s*As\s+\w+\s*$` Matches `Function` declarations, including the return type, and captures the procedure name.

2. Indentation Tracking

The tool tracks indent levels to correctly identify the matching `End Sub`/`End Function`, even with nested blocks (e.g., `If...Else`, loops). This avoids stopping at inner procedures or misaligned end statements.

3. Directory Structure

- For each input file (e.g., `MathUtils.bas`), a subdirectory named `MathUtils` is created in the output folder.
- Each extracted procedure is saved as `<ProcedureName>.bas` inside this subdirectory.

How to Use

1. Build the Tool

Compile the code into an executable (e.g., `Vb6Extractor.exe`) using Visual Studio or the .NET CLI:

```
dotnet build -c Release
```

2. Run the Tool

Open a command prompt and execute the tool with the input and output directories:

```
Vb6Extractor.exe --input "C:\Path\To\Vb6\Source" --output "C:\Path\To\Output"
```

Example Workflow

Input Directory Structure

```
VB6Source/
├── MathUtils.bas
└── FormHandlers/
    └── LoginForm.bas
```

Content of `MathUtils.bas`

```
Public Function AddNumbers(a As Integer, b As Integer) As Integer
    AddNumbers = a + b
```

```
End Function

Private Sub LogResult(result As Integer)
    Debug.Print "Result: " & result
End Sub
```

Output Directory Structure

After running the tool:

```
Output/
├── MathUtils/
│   ├── AddNumbers.bas
│   └── LogResult.bas
└── LoginForm/
    └── [Extracted procedures from LoginForm.bas]
```

Content of `AddNumbers.bas`

```
Public Function AddNumbers(a As Integer, b As Integer) As Integer
    AddNumbers = a + b
End Function
```

Limitations & Notes

- **Nested Procedures**: Inner `Sub`/`Function` procedures are treated as part of the parent block (not extracted separately). Adjust the regex if you need to extract nested procedures.
- **Comment Handling**: Lines starting with `'` are skipped during procedure detection but preserved in the output files.
- **Case Sensitivity**: The tool assumes standard VB6 case insensitivity (e.g., `END SUB` is treated the same as `End Sub`).

This tool provides a robust way to organize and extract VB6 procedures, making it easier to maintain or migrate legacy codebases.