MillingConverter

Maintenance Manual

Simon Walker
simon@stwalkerster.co.uk

3 November 2011

MillingConverter is a tool to take the output from a CNC postprocessor and perform a second set of post-processing in the form of whitespacing and adapting the output to make it readable by the CNC machine. MillingConverter is implemented as a CLI application, targeting the .NET Framework 4 Client Profile on Windows XP Service Pack 2.

Contents

1 Source Control

The source code for the project is held in a Git repository hosted on the Helpmebot cluster, currently stored at `/home/stwalkerster/git/millingconverter.git`. This is a private repository—no anonymous access is allowed.

2 General Information

The code is split into two parts:

1. Graphical user interface and file-handling
2. Parse and processing tree

Each part handles a very specific part of the system, and while there is some crossover, this is as loosely-coupled as possible within the code—it is possible to simply call the parse and processing side with a single function call with a single parameter, and get the corrected version back.

2.1 GUI and File Handling

The GUI is designed to be as simplistic as possible, there are a total of three user interaction points on the interface, all of which are buttons. The user is forced to browse for a file path for the input file (`OpenFileDialog`), browse a path for the output file (`SaveFileDialog`), and click a Go button; when processing is complete a message appears stating this.

The GUI is structured with a `TableLayoutPanel`, set up as close to fully-automatic as possible. An extra zero-height row is included at the bottom for the auto-height rows to be of use. Column 2 is set to auto-size (for the buttons), whereas column 1 is set to 100%, as to force the buttons to be as small as possible.

Event handlers on the browse buttons is as simplistic as possible - show the dialog, and if the dialog was closed properly, set the adjacent `TextBox` to be the selected file.

The event handler on the Go button is more complex. Firstly, it uses the file dialogs to open the file streams for reading and writing the input and output files. Then, it sets up a `StreamReader` and `StreamWriter` to handle the reading and writing of these streams. Once the input and output is set up, we create a list of the lines data in the file with a simple loop. When this has been completed, the input file is closed to free up resources.

To actually convert the text, we use a LINQ-based lambda expression with extension methods to automatically select every item from the input file, apply the conversion to it, and put it into a new list we call `outputData`, if the output line actually contains some data. If not, we discard that line entirely.

```
List<string> outputData = inputData.Select(convert).Where(done =>
done != string.Empty).ToList();
```

We then iterate through the `outputData` list, write every line to the output stream, then finally flush the stream and close the file.

2.2 Parse and Processing Tree

The majority of the application is based around the processing tree, of which the main interface is in the `Form1` code, specifically the `string convert(string);` method.

This method splits the line into a combination of a sequence number, command, and arguments to the command—after checking the line is not just full of whitespace (in which case the method simply returns exactly what it was passed).

This method also rebuilds the string once it has passed through the main processing tree.

The "tree" referred to by this document is the class inheritance tree which forms the main processing engine. All commands have their own specific class which defines the spacing behaviour for that command. All command classes inherit either directly or indirectly from `GenericCommand`. The `GenericCommand` class parses the passed command name, searches for a class (using reflection) which is capable of handling that command (based on the class name: for example the class to handle a `TOOL` command would be called `TOOLCommand` in the `MillingConverter.Commands` namespace).

```
public static GenericCommand create(string command, string arguments)
{
    Type t = Type.GetType("MillingConverter.Commands." + command + "Command");
    if(t == null)
    {
        return new NullCommand(arguments);
    }
    return (GenericCommand)Activator.CreateInstance(t, arguments);
}
```

The `GenericCommand` class hides the `ToString()` method from the `System.Object` class, by declaring it again as `public new abstract string ToString();`. This means we can simply create the command objects, and then convert them back into a string and they will be in the correct format. This was re-defined in order to make it abstract—to force the developer to implement that method.

There are parts where the data has to be manipulated in order to get the correct format.

3 Test Framework

There is a test framework project included within the source code of the system. This test suite is built on the MSTest framework, and consists of approximately four hundred tests which test many aspects of the parse framework, mainly by providing example inputs and required outputs for those inputs.

As of writing, the application passes all unit tests sucessfully.