Woodstocks’ Stock Control

Editor

Developer Manual

Joshua Grimmett

# Table Of Contents

[**Introduction**](#_uk0inaxo72t6) **3**

[**Code Breakdown**](#_19iw6blg0dvi) **4**

[Program.cs (Program entry point)](#_ky8icfuangj2) 4

[MainWindow.cs](#_bkhv4tl4zfuy) 5

[StockItems.cs](#_v7lstemvbxep) 8

[StockItem.cs](#_zfekzup3cblx) 10

[**Unit Testing**](#_1upua14y4vkt) **12**

# 

# Introduction

The Woodstocks’ Stock Editor program is a Windows Forms application, programmed in Visual Studio 2019. The program was written with an object-oriented design philosophy.

# Code Breakdown

## Program.cs (Program entry point)

The Program class is responsible for spawning the main application window and for my case, also containing the stock list. The stock list is held in a custom collection class: StockItems.

The program class also holds a static stocks’ path attribute: defaultStocksCSVPath, which is imported from the configuration file. The class also holds static method named ReloadStocks which resets the stocks to a new temporary StockItems instance loaded from the csv file.

|  |
| --- |
| namespace StockControl {  static class Program  {  public static StockItems stockItems;  public static readonly string defaultStocksCSVPath = ConfigurationManager.AppSettings["stocklist"];   */// <summary>*  */// The main entry point for the application.*  */// </summary>*  [STAThread]  static void Main()  {  *// Initial stocks load*  ReloadStocks();   Application.EnableVisualStyles();  Application.SetCompatibleTextRenderingDefault(false);  Application.Run(new MainWindow());  }   public static void ReloadStocks()  {  *// Load stocks from CSV file*   stockItems = StockItems.LoadFromCSVFile(defaultStocksCSVPath);  }  } } |

## MainWindow.cs

The MainWindow.cs file contains a partial class MainWindow that extends Windows.Forms.Form, holding the functionality and events for the MainWindow class.

The **MainWindow\_Load** method is responsible for loading the grid view when the window loads.

|  |
| --- |
| private void MainWindow\_Load(object sender, EventArgs e) {  *// Disable Auto Columns*  gridStocks.AutoGenerateColumns = false;   *// Initial Grid View Load*  LoadStocksView(); } |

The **LoadStocksView** method is responsible for creating a new data table to be inserted into the data grid view. The method reloads the stocks then created a new DataTable with the columns: Item Code, Description, Current Count and On Order. The method then loops through the global stock items, item by item, converting into DataRows and inserting into the table. The grid’s data source is then updated to the new DataTable.

|  |
| --- |
| private void LoadStocksView() {  *// Create new table to assign stocks to*  DataTable stockDataTable = new DataTable();  *// Initialise DataTable rows*  DataColumn[] columns =  {  new DataColumn("ItemCode", typeof(string)),  new DataColumn("Description", typeof(string)),  new DataColumn("CurrentCount", typeof(Int32)),  new DataColumn("OnOrder", typeof(Boolean))  };   *// Add columns to table*  stockDataTable.Columns.AddRange(columns);   *// Loop through items*  foreach (StockItem item in Program.stockItems)  {  *// Add item to table as row*  stockDataTable.Rows.Add(item.GetDataRow(stockDataTable));  }   *// Set grid view to table*  gridStocks.DataSource = stockDataTable; } |

The **UpdateSideControls** method is responsible for reloading the labels and text boxes on the right hand side of the interface.

|  |
| --- |
| private void UpdateSideControls() {  *// Update current row*  currentRow = gridStocks.CurrentRow;  *// Set all fields to current row*  txt\_ItemCode.Text = currentRow.Cells[0].Value.ToString();  txt\_Description.Text = currentRow.Cells[1].Value.ToString();  txt\_CurrentCount.Text = currentRow.Cells[2].Value.ToString();  box\_OnOrder.Checked = Boolean.Parse(currentRow.Cells[3].Value.ToString()); } |

The **GridStocks\_CellEnter** method is an event handler responsible for updating the side control panel when a new cell is selected.

|  |
| --- |
| private void GridStocks\_CellEnter(object sender, DataGridViewCellEventArgs e) {  *// Update the side control panel*  UpdateSideControls(); } |

The **Btn\_Load\_Click** method is an event handler responsible for (re)loading the new stock list and updating the interface when the “Load Stocks” button is pressed.

|  |
| --- |
| private void Btn\_Load\_Click(object sender, EventArgs e) {  *// Reload stock view*  LoadStocksView();   MessageBox.Show("Stocks have been reloaded."); } |

The **Btn\_Save\_Click** method is responsible for saving the updated table to the csv file when the “Save Stocks” button is pressed.

|  |
| --- |
| private void Btn\_Save\_Click(object sender, EventArgs e) {  bool saved;    *// Save all stocks to the default path*  saved = StockItems.SaveToCSV((DataTable) gridStocks.DataSource, Program.defaultStocksCSVPath);   if (saved)  {  MessageBox.Show("Stocks have been saved.");  } } |

The **Txt\_CurrentCount\_TextChanged** method is responsible for updating the selected cell’s current count field to the value of the side panel text box field when the text box is updated.

|  |
| --- |
| private void Txt\_CurrentCount\_TextChanged(object sender, EventArgs e) {  *// Update current row*  currentRow = gridStocks.CurrentRow;  *// Set current count to the current count text box value*  currentRow.Cells[2].Value = txt\_CurrentCount.Text; } |

## StockItems.cs

The StockItems class is a collection class, holding a list of stocks with functionality of adding stocks, loading stocks from a csv file and saving stocks to a csv file. The StockItems class has a private readonly List attribute, named \_stocks, which holds a list of all StockItem items.

The **LoadFromCSVFile** method is a static method responsible for loading all stocks from the csv file of the given path. It has one string argument, “path” which represents the path to which the file should be saved. The method confirms that the file exists before continuing to create a new temporary instance of the StockItems class. The file is then read and outputted to a new string array variable, lines. The method loops through each line, creating a new StockItem instance from the CSV line. The StockItem instance is then added with the AddStockItem method. The temporary StockItems instance is then returned.

|  |
| --- |
| public static StockItems LoadFromCSVFile(string path) {  *// Confirm file exists*  if (!File.Exists(path))  {   throw new FileNotFoundException(path);  }   *// Instantiate stock items*  StockItems stockItems = new StockItems();   *// Load lines from file*  string[] lines = File.ReadAllLines(path);    *// Loop lines to create stock*  foreach (string line in lines)  {  *// First line of CSV is column names*  if (lines[0] == line) continue;   *// Create new stock item from line string*  StockItem item = new StockItem(line);  *// Add new item to stock list*  stockItems.AddStockItem(item);  }   *// Return object*  return stockItems; } |

The **AddStockItem** method is a public method responsible for adding a new stock item to the private stock item list.

|  |
| --- |
| public void AddStockItem(StockItem item) {  *// Add item to stocks*  this.\_stocks.Add(item); } |

The **SaveToCSV** method is a static method responsible for saving a DataTable to the given path. The method has two arguments: table, a variable that holds the DataTable to be written, and path, a string variable representing the path that the file will be written to. The method begins by creating a new string variable, newFile to be written to the file. The newFile variable begins with a string of CSV containing the columns’ names. The method then loops through each row in the table converting them to CSV format and adding to the newFile variable. The method finally attempts to write the newFile string to the file.

|  |
| --- |
| public static bool SaveToCSV(DataTable table, string path) {  *// Inisialise new file string*  string newFile = $"{table.Columns[0].ColumnName},{table.Columns[1].ColumnName}," +  $"{table.Columns[2].ColumnName},{table.Columns[3].ColumnName}\n";   *// Loop through existing rows*  foreach(DataRow row in table.Rows)  {  *// Parse true/false to Yes/No*  string onOrder;  if(Boolean.Parse(row["OnOrder"].ToString()) == true)  {  onOrder = "Yes";  }  else  {  onOrder = "No";  }  *// Add CSV line to file*  newFile += $"{row["ItemCode"]},{row["Description"]},{row["CurrentCount"]},{onOrder}\n";  }   try  {  *// Write new text to the file*  File.WriteAllText(path, newFile);  return true;  }  catch  {  MessageBox.Show("Unauthorized access. Could not write to: " + path);  return false;  } } |

## StockItem.cs

The StockItem class represents a stock item with an item code, description, count and on order attributes. The StockItem class can load a stock item from CSV and convert into a DataRow.

The StockItem constructor takes a string of CSV and converts it into 4 variables: ItemCode, Description, CurrentCount and OnOrder.

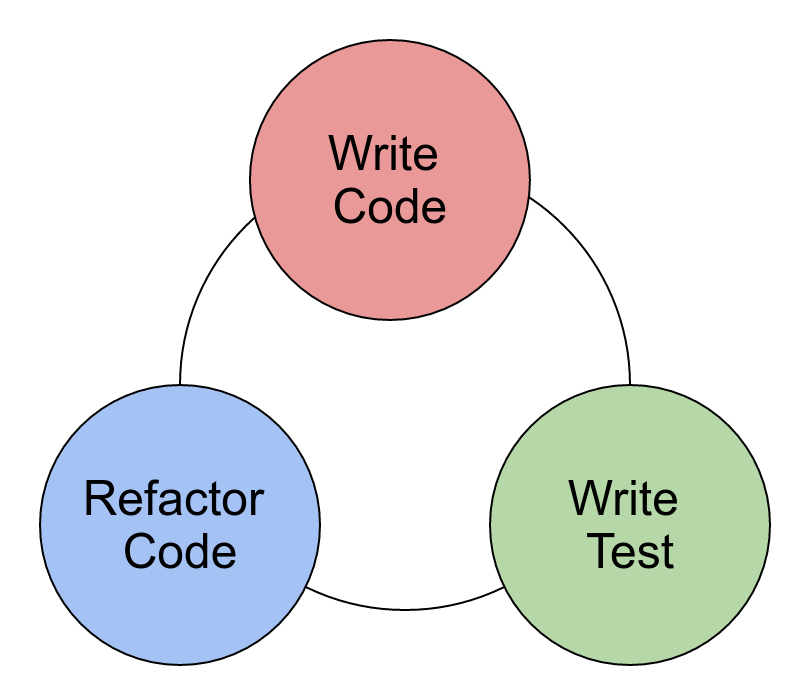
|  |
| --- |
| public StockItem(string csvLine) {  *// Parse CSV to array*  string[] line = csvLine.Split(',');  *// Columns count should not exceed 4*  if (line.Length > 4)  {  throw new Exception("Invalid CSV line");  }   *// Parse line to columns*  ItemCode = line[0];  Description = line[1];  CurrentCount = Int32.Parse(line[2]);   *// Parse Yes/No*  string onOrder = line[3].ToLower();  if (onOrder == "true" || onOrder == "yes")  {  OnOrder = true;  }  else if (onOrder == "false" || onOrder == "no")  {  OnOrder = false;  }  else  {  throw new InvalidCastException();  } } |

The **GetDataRow** method is responsible for converting a stock item into a DataRow. The method takes a single argument, table, an instance of DataTable which the DataRow will be added to. The method creates a new row from the given table, and assigns the attributes of the StockItem.

|  |
| --- |
| public DataRow GetDataRow(DataTable table) {  *// Initialise new row variable*  DataRow row;  *// Create row from given table*  row = table.NewRow();   *// Assign column values*  row["ItemCode"] = this.ItemCode;  row["Description"] = this.Description;  row["CurrentCount"] = this.CurrentCount;  row["OnOrder"] = this.OnOrder;   *// Return the new row*  return row; } |

# Unit Testing

During the development of this application Microsoft Visual Studio’s Unit Testing was utilized. The unit testing project allows you to create test methods and run and debug tests over an over with refactoring.



In this unit test, **StockItemFromCSVTest**, we make sure that when you initialise a StockItem instance with CSV, it returns a correct output. After refactoring code, this test passed.

|  |
| --- |
| public class StockItemTests {  public static string productCsv;  public StockItemTests()  {  *// Testing csv product*  productCsv = "T001,A test product,7,No";  }   [TestMethod]  public void StockItemFromCSVTest()  {  *// Expected item code: T001*   *// Create stock item from StockItem class*  StockItem stockItem = new StockItem(productCsv);   *// Assert that item's code is T001*  Assert.AreEqual("T001", stockItem.ItemCode);  } } |