First Steps for your Spreadsheet Application
Cpt S 322 Homework Assignment
by Evan Olds

Submission Instructions: (see the syllabus)

Important note: This is the framework for a spreadsheet application that you will build over the course of the semester. Almost ALL remaining homework assignments will build on top of this.

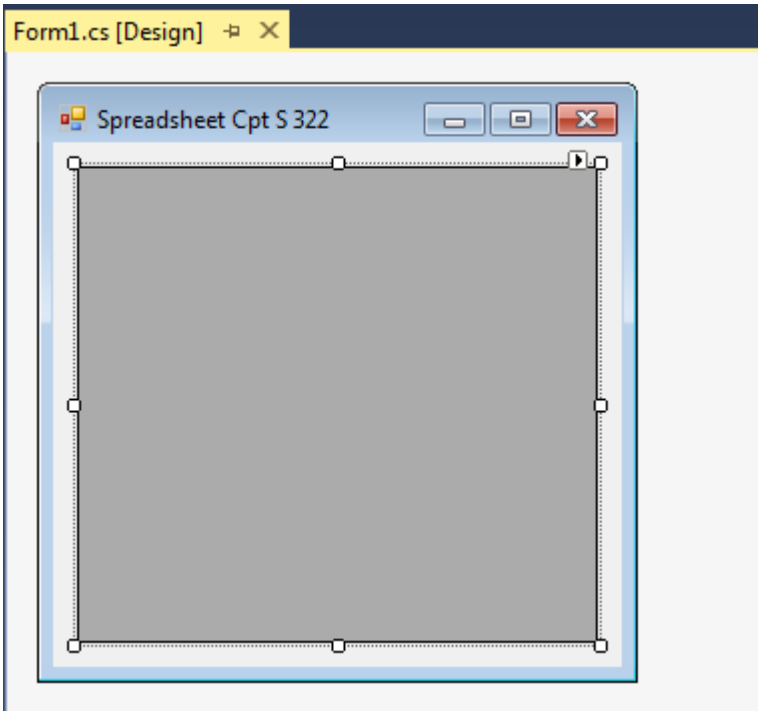
Assignment Instructions:

Read each step's instructions *carefully* before you write any code.

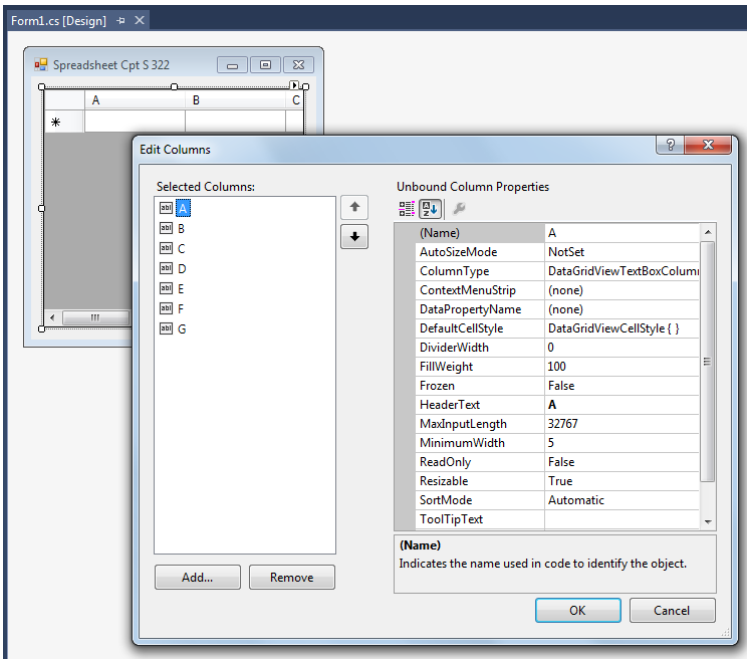
In this assignment you'll create the basic parts of a spreadsheet application. Most of the formula and computation-oriented functionality will be implemented in later assignments so you won't be implementation the entire application in one week. But this assignment lays the foundation and it important because you will build on top of it in future assignments.

Using Visual Studio or SharpDevelop, create a WinForms application and name it "Spreadsheet_" (without quotes) followed by the first initial of your first name and then your entire last name. So for example if your name is Alex Smith then the project name would be **Spreadsheet_ASmith**. Put your name in comments at the top of every code file.

After creating the application, drag a DataGridView (another link here with info about properties, methods and events) control into the window. This control gives you a spreadsheet-like array of cells, but requires some setup. It initially looks like this (image below) when you first add it to the form:



In the properties tool window there is a **Columns** property for the grid view. Click the "…" button next to it to open up the column editor. Here you can add columns. Experiment with adding a few columns, say A, B, C, and D:



Note that the **HeaderText** property is what actually appears as the column header in the control. You can also add rows in the designer but we want to do this programmatically to save time.

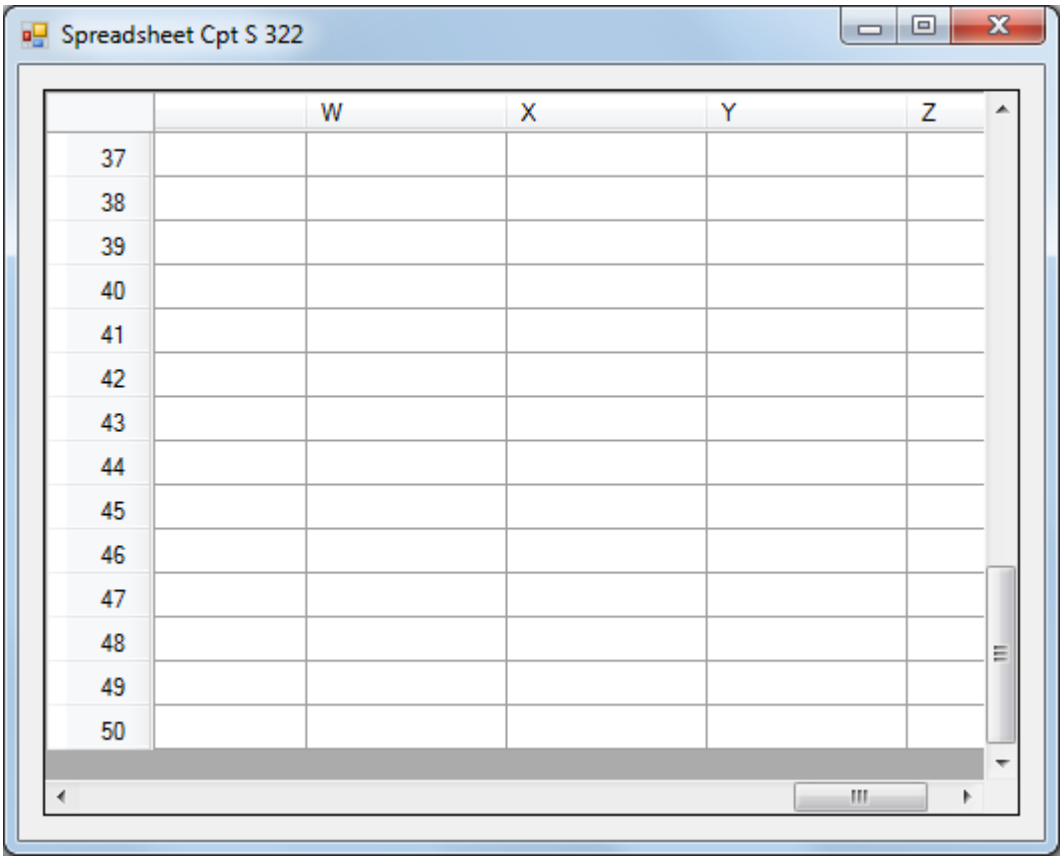
Step 1 – Create Columns A to Z with code (1 point):

- Set the AllowUserToAddRows and AllowUserToDeleteRows properties to false. We will create a fixed number of rows and columns for our spreadsheet.
- In the form's constructor, or in the OnLoad event, write code to programmatically create columns A to Z. If you've added some columns in the designer you'll want to clear those first (see the Clear method of the Columns property). The Columns property also has an Add method so investigate this.

Step 2 – Create Rows 1 to 50 (1 point):

- Again, in code, create 50 rows programmatically. Do this *after* you've created the columns.
- Much like the columns, the DataGridView control also has a Rows property that will allow you to do this.
- Make sure the columns are numbered 1 to 50, not 0 to 49.

At this point you should see something like this when you run your app:



Step 3 – Create the logic engine class library and reference it (1 point)

1. Right click the solution (not the project) in the "Solution Explorer" tool window and go to "Add" then click "New Project…".
2. Select "Class Library" as the type.
3. Choose a reasonable name for it. I have chosen "SpreadsheetEngine" for mine. Click the "OK" button to create it.
4. In the WinForms app add the class library to the references. In the window that pops up when you add a new DLL reference, there's an option to reference one that's within the solution. Choose that.

You will have a solution in Visual Studio (or SharpDevelop) that has two projects in it at this point. One is a WinForms application and the other is a class library. Remember the idea behind having the logic in a class library is that you're decoupling it from the WinForms stuff. So don't reference System.Windows.Forms in that DLL project.

Step 4 – Create and implement the "Cell" class (1 point)

In the class library code, create and implement the "Cell" or "SpreadsheetCell", (or some other similar name of your choosing) class. This class represents one cell in the worksheet.

- It must be an abstract base class
- It (the class itself) must be declared publicly so the world outside of the class library can see it
- Add a RowIndex property that is read only (set in constructor and returned through the get)
- Add a ColumnIndex property that is read only (set in constructor and returned through the get)
- It must have a string "Text" property that represents the actual text that's typed into the cell. Recall that a property is essentially a getter/setter. So make this property a getter and setter for a member variable that's marked as protected.
    - The getter can just return the protected property
    - The setter must do the following:
        1. If the text is being set to the exact same text then just ignore it. Do not invoke the property change event (discussed below) if the property isn't actually being changed.
        2. If the text is actually being changed then update the private member variable and fire the PropertyChanged event, which is discussed in the next bullet point.
- Make the spreadsheet cell class implement the INotifyPropertyChanged interface (declared in the System.ComponentModel namespace).
    - We need a way for the logic engine in this class library to notify other the UI layer when changes have been made to the spreadsheet.
    - Implementing this interface allows the cell to notify anything that subscribes to this event that the "Text" property has changed.

    - Going back to the point mentioned above, call the PropertyChanged event (just like calling a function) when the text changes. You'll have to read up on events and the INotifyPropertyChanged interface to figure out how to do this.
    - This page talks about properly implementing this interface.
- It must have a **Value** property that's also a string. Like the **Text** property this needs to be a protected member variable that is exposed through a property.
    - This represents the "evaluated" value of the cell. It will just be the **Text** property if the text doesn't start with the '=' character. Otherwise it will represent the evaluation of the formula that's type in the cell.
    - Since many formulas in spreadsheet cells reference other cells we need for the actual spreadsheet class to set this value.
    - However, we don't want the "outside world" (outside of the spreadsheet class) to set this value so you must design it so that only the spreadsheet class can set it, but anything can get it.
    - The big hint for this: It's a protected property which means inheriting classes can see it. Inheriting classes should NOT be publically exposed to code outside the class library.
    - So to summarize, this **Value** property is a getter only and you'll have to implement a way to allow the spreadsheet class to set the value, but no other class can.

Step 5 – Create and implement the "Spreadsheet" class (2 points)

- In the class library code implement the "Spreadsheet" (or some similar name of your choosing) class.
- The spreadsheet object will serve as a container for a 2D array of cells. It will also serve as a factory for cells, meaning it is the entity that actually creates all the cells in the spreadsheet. Remember that the cell class is abstract, so the outside world can't create an instance of a cell. It can only get a reference to a cell from the spreadsheet object.
- Have a constructor for the spreadsheet class that takes a number of rows and columns. For the WinForms stuff in this assignment we made a fixed number of rows and columns, but we don't want the actually spreadsheet object to have the "dimensions" to be hard coded. So the constructor should allocate a 2D array (or 1D if you want to do the indexing math for all the cell retrieval functions) of cells.
    - Initialize the array of cells and make sure to give them proper RowIndex and ColumnIndex values
- Again, you need to come up with a design here that actually allows the spreadsheet to create cells and there were hints before about how to do this. You cannot make the publically declared cell class non-abstract.
- Make a **CellPropertyChanged** event in the spreadsheet class. This will serve as a way for the outside world (like UI stuff) to subscribe to a single event that lets them know when any property for any cell in the worksheet has changed.

    - The spreadsheet class has to subscribe to all the PropertyChanged events for every cell in order to allow this to happen.
    - This is where the spreadsheet will set the value for a particular cell if its text has just changed. The implementation of this is discussed more in step 6.
    - When a cell triggers the event the spreadsheet will "route" it by calling its **CellPropertyChanged** event.
- Make a **GetCell** function that takes a row and column index and returns the cell at that location or null if there is no such cell. The return type for this method should be the abstract cell class declared in step 4.
- Add properties **ColumnCount** and **RowCount** that return the number of columns and rows in the spreadsheet, respectively.

Step 6 – Complete the implementation of the CellPropertyChanged event in the spreadsheet (2 points)

- The rules are if the text of the cell has just changed then the spreadsheet is responsible for updating the **Value** of the cell.
- If the **Text** of the cell does NOT start with '=' then the value is just set to the text.
- Otherwise the value must be computed based on the formula that comes after the '='.
    - Future versions (later homework assignments) will go much further with this but now we'll only support one type of formula.
    - Support pulling the value from another cell. So if you see the text in the cell starting with '=' then assume the remaining part is the name of the cell we need to copy a value from.
    - It's not required for this assignment, but in the future we'll need a way to deal with circular references (cell A gets value from B but B gets value from A), so keep that in mind.

Step 7 – Link the WinForms to the DLL and do a demo (2 points)

- Add a Spreadsheet object as a member variable to your main form's code.
- Initialize this spreadsheet object in the form's constructor so that it has 26 columns and 50 rows.
- Subscribe to the spreadsheet's **CellPropertyChanged** event. Implement this so that when a cell's **Value** changes it gets updated in the cell in the DataGridView.
    - The DataGridView has a **Rows** property and you can get individual DataGridView cells from that. Cells in the UI grid have a **Value** property that you should set to the **Value** of the logic engine cell.
    - Remember that what's actually changed when this event gets executed is a cell from the logic-engine spreadsheet. You're just updating the UI in response to that.
- Create a button on the form that, when clicked, shows a demo. This demo will illustrate how changing cells in the worksheet object triggers a proper UI update.
    - The demo should set the text in about 50 random cells to a text string of your choice. "Hello World!" would be fine or some other message would be ok too.

- Also, do a loop to set the text in every cell in column B to “This is cell B#”, where # number is the row number for the cell.
- Then set the text in every cell in column A to “=B#”, where ‘#’ is the row number of the cell. So in other words you’re setting every cell in column A to have a value equal to the cell to the right of it in column B.
- The result should be that the cells in column A update to have the same values as column B.

- Remember that all modifications are happening to objects from the logic engine/class library and the UI is just responding to such changes. You will not receive points for the demo if you’re directly setting the values in the DataGridView cells as opposed to setting them in the Spreadsheet object’s cells.