First Steps for your Spreadsheet Application
Cpt S 321 Homework Assignment
Washington State University

Submission Instructions:

- Create a branch called "Branch_HW4" and work in this branch for this assignment.
- **When you are done, merge the branch back to the master (NEW REQUIREMENT!)**
- Once you are done and before the deadline, tag the version that you would want us to grade with the assignment number (for example, "HW4").
- On Canvas -> Assignments -> Submit the link to your repository (a link to the tag or the branch works) by the HW deadline.
- **IMPORTANT: The HW must be tagged by the due date and a link to that tag needs to be submitted via Canvas in order to receive a grade.**

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:

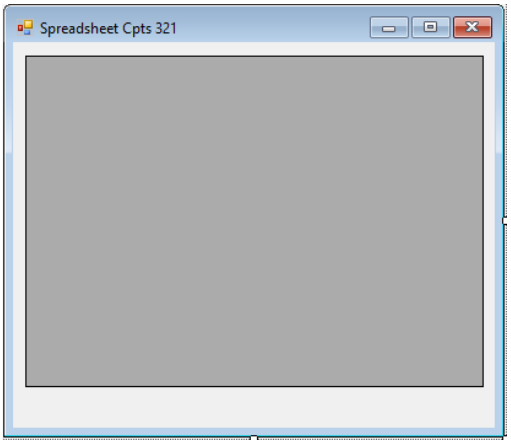
Read each step's instructions *carefully* before you write any code. StyleCop should be enabled for all your projects, including your test project.

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 implementing 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.

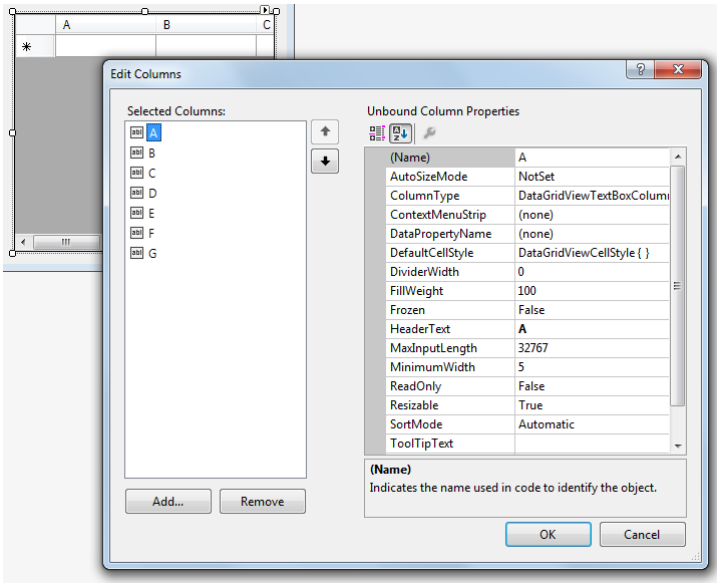
Using Visual Studio (or Rider), create a WinForms (or Avalonia) application and name it "**Spreadsheet_FirstName_LastName**" (without quotes). So for example if your name is Alex Smith then the project name would be **Spreadsheet_Alex_Smith**. As usual, your name and ID should also appear in the header comments of every code file.

A. If you are using WINFORMS, follow the instructions below:

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.

Set the AllowUserToAddRows and AllowUserToDeleteRows properties to false. We will create a fixed number of rows and columns for our spreadsheet.

B. If you are using AVALONIA, follow the instructions below:

- **a.** After creating the application, install Avalonia.Controls.DataGrid using the NuGet package manager.
- **b.** In the App.axaml file, add the DataGrid style:

```
<Application.Styles>
    ...
    <StyleInclude Source="avares://Avalonia.Controls.DataGrid/Themes/Fluent.xaml" />
</Application.Styles>
```

c. Then you can add the DataGrid in the MainWindow.axaml file as such:

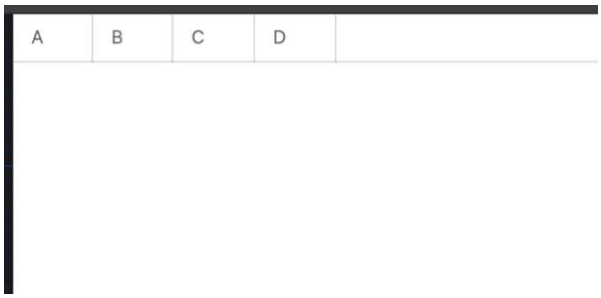
```
<DataGrid Name="SpreadsheetDataGrid" AutoGenerateColumns="False">
</DataGrid>
```

At this point nothing will be visible – this is normal.

d. Let's add some columns. Within the DataGrid tag add the following 4 columns:

```
<DataGrid.Columns>
    <DataGridTextColumn Header="A"></DataGridTextColumn>
    <DataGridTextColumn Header="B"></DataGridTextColumn>
    <DataGridTextColumn Header="C"></DataGridTextColumn>
    <DataGridTextColumn Header="D"></DataGridTextColumn>
</DataGrid.Columns>
```

You should see the following result:



Once you have completed A or B above, follow the instructions below.

Step 1 – Create Columns A to Z with code:

- In the constructor of your UI class, call a method InitializeDataGrid (that you will implement) in which you will programmatically create columns A to Z.
- WinForms: If you've added some columns in the designer you'll want to clear those first (see the Clear method). There is also an Add method so investigate this.
- Avalonia:

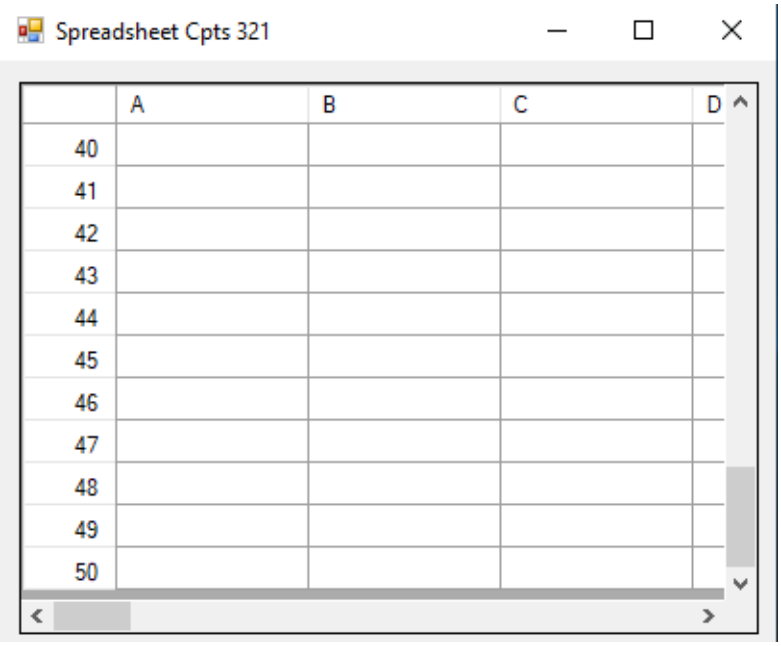
- Clear the columns that you added manually and now do it programmatically.
- Since we are manipulating the view, we will edit the MainWindow.axaml.cs file. The data grid will be accessible with the name you gave in the MainWindow.axaml file.
- In the MainWindow.axaml add the HorizontalScrollBarVisibility and VerticalScrollBarVisibility set to Auto to enable scrolling when needed.
- For the InitializeDataGrid we can use the DataGridTextColumn or DataGridTemplateColumn. (The DataGridTextColumn will be sufficient for this HW but in later HWs the DataGridTemplateColumn will be needed.)
- In order to properly call the InitializeDataGrid() we can use a reactive statement. We need to change the MainWindow parent class to `ReactiveWindow<MainWindowViewModel>` and then we can use WhenAnyValue in the MainWindow constructor (see the additional code provided for Avalonia - Part 1).

Step 2 – Create Rows 1 to 50:

- Again, in code, create 50 rows programmatically. Do this *after* you've created the columns.
- WinForms: Much like the columns, the DataGridView control also has a Rows property that will allow you to do this.
- Avalonia:
  - In Avalonia the rows of the DataGrid correspond to the data, each Row is to be considered as an object. In our case, the object will be an array of data corresponding to columns A to Z. Hence to add rows we need to bind the Items property of the DataGrid to a property in the MainWindowViewModel. Make sure the rows are numbered 1 to 50, not 0 to 49 (see the additional code provided for Avalonia - Part 2). At this point the preview will look like this:

  - Finally, we need to add Row headers and grid lines (see the additional code provided for Avalonia - Part 3).

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



Step 3 – Create the logic engine class library (this is a separate project in the same solution!) and reference it:

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 (e.g., "SpreadsheetEngine"). Click the "OK" button to create it.
4. In the WinForms or Avalonia app, add the class library to the references.

You will have a solution in your IDE that has two projects in it at this point. One is a WinForms or an Avalonia 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 UI code. So **do not** reference System.Windows.Forms or Avalonia classes in that DLL project.

Step 4 – Create and implement the "Cell" class

In the class library code, create and implement the "**Cell**" class. This class represents one cell in the worksheet. Implement this class to obey the rules listed below.

- 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 the constructor and returned through the get)

- Add a ColumnIndex property that is read only (set in the 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 properties promote encapsulation. So make sure that the corresponding field (i.e., member variable) is marked as protected.
    - The getter can just return the protected field
    - 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 protected field and fire the PropertyChanged event,

        Note: We have discussed how to do this in class so check the lecture notes. More details are given 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 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.

    Note: Again, we have discussed how to do this in class so check the lecture notes.
- It must have a "**Value" property** that's also a string. Like for the **Text** property this needs to be associated with a protected field that is exposed through the 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: As there is a protected field this means that inheriting classes can see this field. Inheriting classes should NOT be publicly exposed to code outside the class library.
    - To summarize: the **Value** property is a getter only and you'll have to think of a way to allow the **Spreadsheet** class (more details on this below) to set the value, but no other class can.

Step 5 – Create and implement the "Spreadsheet" class

- In the class library code implement the "**Spreadsheet**" 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.
  (NOTE: We will talk about the factory design pattern in class very soon; you do not need to implement the pattern now but you will need to do it in a later HW. At this point simply make sure that you create all the cells in the spreadsheet.)
- Have a constructor for the spreadsheet class that takes a number of rows and columns. For the UI in this assignment we made a fixed number of rows and columns, but we don't want the 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 publicly declared cell class non-abstract.
- Make a **CellPropertyChanged** event in the spreadsheet class. This will serve as a way for the outside world (like the UI) 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

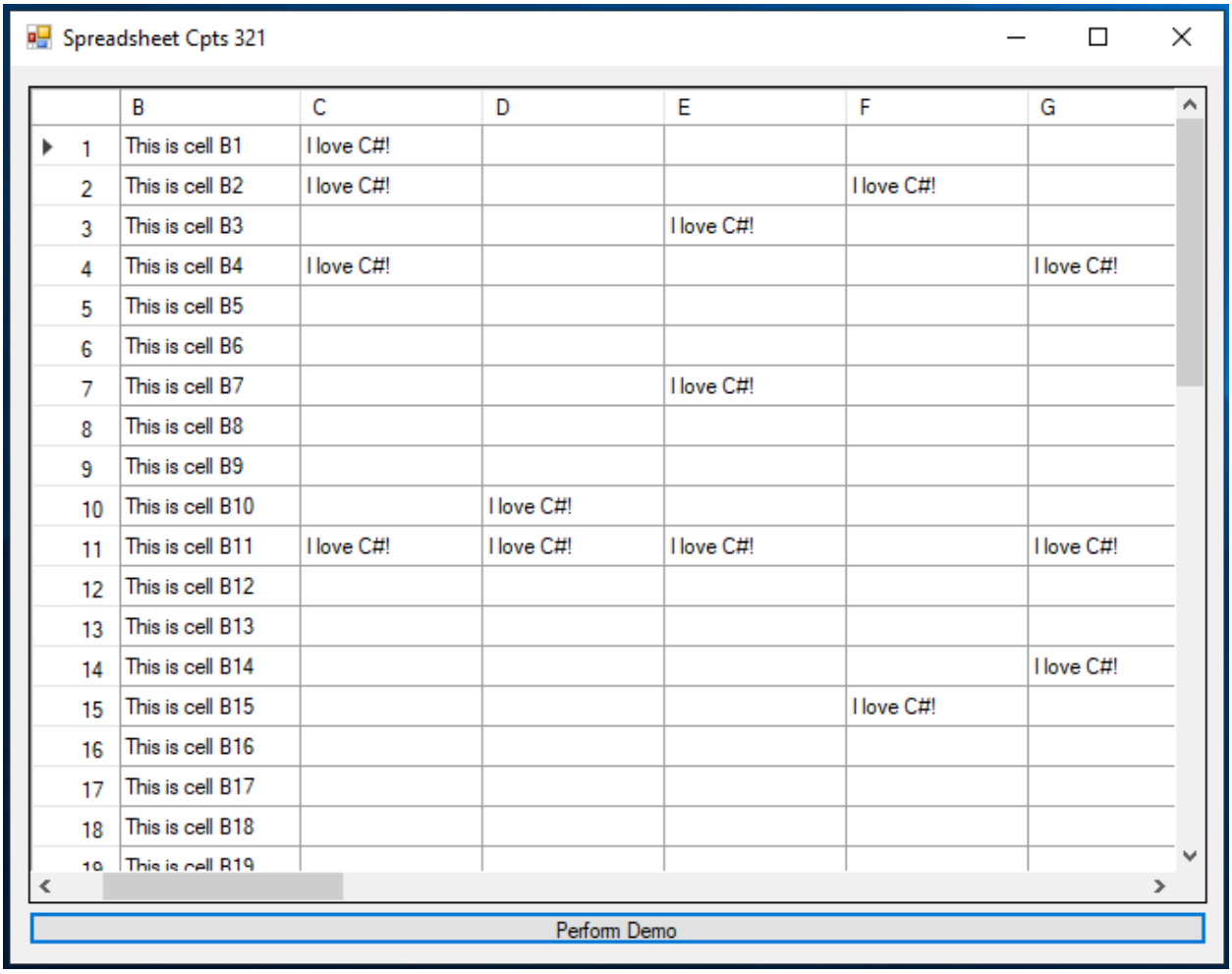
- The rules are if the text of the cell has just changed then the **Spreadsheet** is responsible for updating the **Value** of the cell.
- If the **Text** of the cell does NOT start with '=' then the value is just set to the text.
- Otherwise, the value must be computed based on the formula that comes 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 is **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.
- Also, you do not need to deal with indirect references yet; we will add this requirement in a later homework assignment (HW7). Having said that, feel free to implement it now if you want!

Step 7 – Link the UI project (WinForms or Avalonia) to the DLL and do a demo

- Add a Spreadsheet object as a member variable to your main UI or ViewModel class – Form1 (for WinForms) or MainWindowViewModel (for Avalonia).
- Initialize this spreadsheet object in the 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.
  - WinForms: 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.
  - Avalonia: To have the value updated in the DataGrid, we need to adapt the InitializeDataGrid method to tell the DataGrid Column how it should be rendered.
  - 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.
- **Demo**: Create a button in the UI 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.



	B	C	D	E	F	G
1	This is cell B1	I love C#!				
2	This is cell B2	I love C#!			I love C#!	
3	This is cell B3			I love C#!		
4	This is cell B4	I love C#!				I love C#!
5	This is cell B5					
6	This is cell B6					
7	This is cell B7			I love C#!		
8	This is cell B8					
9	This is cell B9					
10	This is cell B10		I love C#!			
11	This is cell B11	I love C#!	I love C#!	I love C#!		I love C#!
12	This is cell B12					
13	This is cell B13					
14	This is cell B14					I love C#!
15	This is cell B15				I love C#!	
16	This is cell B16					
17	This is cell B17					
18	This is cell B18					
19	This is cell B19					

Point breakdown (the assignment is worth 10 points):

- 5 points for implementing the correct functionality as stated above
  - 1 point for steps 1-3 as follows: Step 1: 0.4, Step 2: 0.4, Step 3: 0.2
  - 4 point for steps 4-7 (1 point for each step)

And as usual:

- 1 point: For a "healthy" version control history, i.e., 1) the HW assignment should be built iteratively, 2) every commit should be a cohesive functionality, 3) the commit message should concisely describe what is being committed, 4) you should follow TDD – i.e., write and commit tests first and then implement and commit the functionality.
- 1 point: Code is clean, efficient and well organized.
- 1 point: Quality of identifiers.

- 1 point: Existence and quality of comments.
- 1 point: Existence and quality of test cases.

General Homework Requirements	
Quality of Version Control	● Homework should be built iteratively (i.e., one feature at a time, not in one huge commit). ● Each commit should have cohesive functionality. ● Commit messages should concisely describe what is being committed. ● TDD should be used (i.e, write and commit tests first and then implement and commit functionality). ● Include “TDD” in all commit messages with tests that are written before the functionality is implemented. ● Use of a .gitignore. ● Commenting is done as the homework is built (i.e, there is commenting added in each commit, not done all at once at the end).
Quality of Code	● Each file should only contain one public class. ● Correct use of access modifiers. ● Classes are cohesive. ● Namespaces make sense. ● Code is easy to follow. ● StyleCop is installed and configured correctly for all projects in the solution and all warnings are resolved. If any warnings are suppressed, a good reason must be provided. ● Use of appropriate design patterns and software principles seen in class.
Quality of Identifiers	● No underscores in names of classes, attributes, and properties. ● No numbers in names of classes or tests. ● Identifiers should be descriptive. ● Project names should make sense. ● Class names and method names use PascalCasing. ● Method arguments and local variables use camelCasing. ● No Linguistic Antipatterns or Lexicon Bad Smells.
Existence and Quality of Comments	● Every method, attribute, type, and test case has a comment block with a minimum of <summary>, <returns>, <param>, and <exception> filled in as

	applicable. ● All comment blocks use the format that is generated when typing "///" on the line above each entity. ● There is useful inline commenting in addition to comment blocks that explains how the algorithm is implemented.
Existence and Quality of Tests	● Normal, boundary, and overflow/error cases should be tested for each feature. ● Test cases should be modularized (i.e, you should have a separate test case for each thing you test - do not combine them into one large test case). ● *Note: In assignments with a GUI, we do not require testing of the GUI itself.*