Undo/Redo System Implementation
Cpt S 321 Homework Assignment
Washington State University

Submission Instructions:

- Create a branch called "Branch_HW8" and work in this branch for this assignment.
- **When you are done, merge the branch back to the master.**
- Once you are done and before the deadline, tag the version that you would want us to grade with the assignment number (for example, "HW8").
- 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.**

Assignment Instructions:

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

At this point we have finished talking about all refactorings of the expression tree demo in class and you are supposed to have them all implemented in this assignment (i.e., factory method pattern, shunting yard algorithm, handle operator precedence/associativity explicitly, use reflection instead on hardcoding operators, throw descriptive exceptions, not have redundant code).

In this assignment, you will implement an undo and redo system for your spreadsheet application. You will also add the ability to choose the background color of a cell. Your undo system will support undo and redo actions for changing the cell's text or background color.

Part 1 – Add a background color property to cells with accompanying UI changes

Make sure you implement this in the same way you've dealt with other cell properties. You'll have a background color in the logic layer and changing that property will invoke a property-changed event that the UI will respond to.

In the logic engine:

- Add a **uint** property to the Cell class named "BGColor". Since it's in the logic engine, we must have a UI-independent data type, and **uint** certainly meets that requirement. Make the default background color white (which is 0xFFFFFFFF).
- Make sure that when it gets changed you invoke the property changed event.

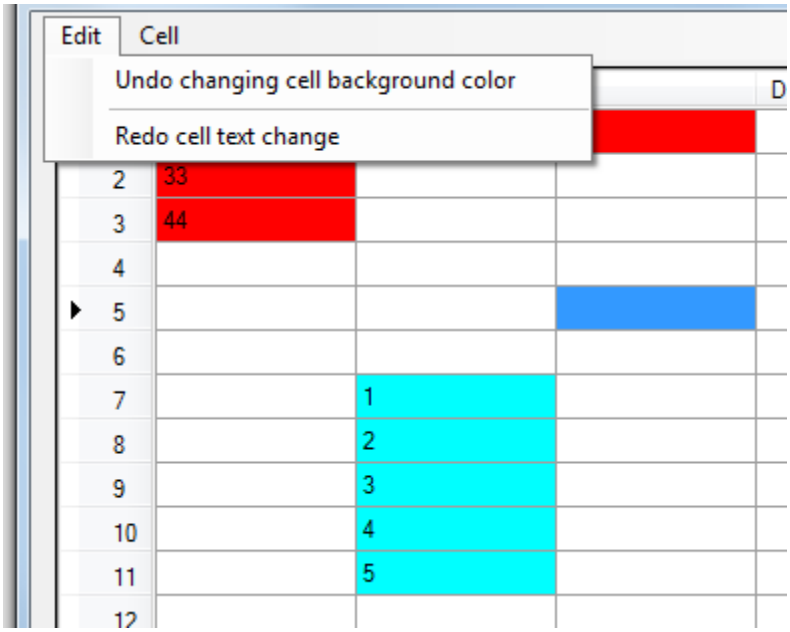
In the GUI application:

- Extend your event-handling code to respond to changes to cell background colors
- WinForms:
    - Update cell backgrounds in the DataGridView accordingly
    - Use the DataGridViewCell.Style.BackColor property
    - That property is of type System.Drawing.Color which has a FromArgb method
- Avalonia - to reflect the color set to a cell you will need to refer to the provided code as follows:
    - Changes to MainWindow.axaml and MainWindow.axaml.cs for cell selection and Row and Cell styling (see Part 1 in the additional code).
    - The cell background will be updated using the RowViewModelIBrushConverter.cs provided
    - You will need to Refactor your code to use the provided RowViewModel.cs and CellViewModel.cs
    - You will also need to refactor the MainWindowViewModel accordingly to handle selected cells (see Part 2 in the additional code).
- Add a button or menu option to change the background color of the selected cells. Change the color for all selected cells when the user selects this option and chooses a color
- WinForms:
    - Use a ColorDialog to prompt the user for a color. Remember that you should be setting the background color in the data/logic cell, then the UI should update in response to this change.
- Avalonia:
    - Download the nuget package AvaloniaColorPicker
    - To show the color picker dialog, refer to the way you handled open and save file dialogs in previous homeworks

Part 2 – Implement the undo/redo system

- Support undoing cell text changes and cell background color changes
- Every undo that is executed should automatically push an item onto the redo stack
- Neither the undo or redo stacks should be publically exposed. That is, don't do:
    - public Stack<UndoRedoCollection> Undos
- Declare it privately then offer the ability to add and execute undo commands through public functions
    - Have a public **AddUndo** function in the spreadsheet class (or workbook class if you decide to create one of those)
- There must be menu options or buttons that allow the user to undo or redo at any time.

    - If the undo stack is empty then disable the undo menu item.
    - If the redo stack is empty then disable the redo menu item.
- Recall that each item on the undo or redo stack should be a collection of simple command objects with an accompanying title that tells the user what is going to be undone/redone. Display that information in the menu items as shown in the screenshot below.
- The design must support the addition of new undo/redo functionality without disrupting or requiring changes in any existing functionality. For example, if you implement everything correctly in this assignment then in future assignments you could add a Border property to the cell and add undo/redo functionality for changing cell borders WITHOUT having to modify anything in the undo/redo classes you implemented for this homework. You would just need to add a new class or set of classes for the new functionality.



<u>Point breakdown (the assignment is worth 10 points):</u>

- 5 points for implementing the correct functionality

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.*