Project 2:
For the Queen To Use

Project due by April 15 via a Zoom demonstration.

Overview

In this project, you will develop a View and ViewModel for the chess model you created in Project 1, plugging into the multi-game WPF application developed in lecture.

General Architecture

The WPF window `GameChoiceWindow` declares an array of game types can can be played, each of which derives from `IWpfGameFactory`. Each `IWpfGameFactory` can create a `IWpfGameView` object that contains a `IGameViewModel` and WPF `Control` for displaying its game in the application, as well as create a few `IValueConverter` objects for displaying things like the current player and board advantage.

ChessViewModel:

You will need to write this class, using `OthelloViewModel` and `TicTacToeViewModel` as examples. The class must implement `IGameViewModel`, which declares a few properties and events that must be implemented:

- `CurrentAdvantage` – returns the board's current value.
- `CurrentPlayer` – returns the board's current player.
- `CanUndo` – returns true if it is legal to call `UndoLastMove` (if there is at least one move in the history).
- `UndoMove()` – undoes the last move.
- `GameFinished` – this **event** must be invoked after applying a move, if that move means the end of the game. For chess, that means a move that ends in checkmate or stalemate. **You do not need to change the Model for this.**

This class should maintain private variables to track a chess game's state, using the `ChessBoard` class you wrote in Project 1. You will need to expose parts of the game state as properties that your UI View can bind to, again following the example in Othello. I recommend a `ChessSquare` class that exposes a `ChessPiece` instead of an integer for the piece at the given square location.

This class should not maintain its own board "advantage", current player, etc.; when asked for these things in property getters, the view model should simply turn around and ask the private chess board from the model. However, you should not expose the private `ChessBoard` object to the View... **remember**, the View **must** go through the ViewModel in order to affect the game state.

ChessView:

Create a WPF `UserControl` named `ChessView` that is responsible for drawing a chess game, using `OthelloView` and `TicTacToeView` as examples. You **must** follow these guidelines:

1. Find and download a set of PNG images for the 12 different chess pieces (6 types, 2 players). Copy them into the `Cecs475.BoardGames.Chess.WpfView` project's `Resources` folder, then use `Add Existing Item...` in Visual Studio to add them to the same project. After they are added, select the files in Visual Studio and go to the Properties window (F4 key). Set the Build Action of the files to Resource.
2. You control should display an 8x8 uniform grid of squares. You will need a converter to convert a `ChessSquare` into an `Image` control... see the `War` example. The syntax to load an image placed in your Resources folder, set to a Build Action of Resource, named "MyImage.png" is:
   `new BitmapImage(new Uri("/Cecs475.BoardGames.Chess.WpfView;component/Resources/MyImage.png", UriKind.Relative))`

3. To select a move, the player must take two steps: first, they click the piece they want to move with the mouse; then then click the square they want to move the piece to.
   (a) Upon clicking a piece that is owned by the current player, the View should draw that piece with a red background, showing that it has been "selected".
   (b) If a piece has been selected, then you need to indicate when a square that the mouse enters is a possible move for the selected piece, by drawing it with a light green background.
   (c) If the user clicks on a square that is not a possible move, then the selected piece is "cancelled" and no longer selected.
   (d) If the user clicks on a square that is a possible move for the selected piece, then the move is applied via the ViewModel.
4. Your grid must alternate the background color of the squares, like a real chess board, but overridden by other colors based on user actions and the state of the board:
   (a) If the square contains the king of the current player, and the current player is in check, then the square gets a yellow background.
   (b) If the square has been "selected", it should be drawn red.
   (c) If a square is currently selected, and the mouse is hovering over a square that is the end position of a possible move starting at the selected square, then the hovered square should be drawn light green.
   (d) If no square is currently selected, and the mouse is hovering over a square that is the start position of at least one possible move, then the hovered square should be drawn light green.
   (e) Otherwise, choose one "dark" color and one "light" color, and return the appropriate color for the given board position.
   (f) You will need to write a multiconverter and bind it to a multibinding on several properties of the `ChessSquare` class, as discussed in lecture. Hopefully you have noticed that we just described four criteria for determining square colors, and can translate (most of) those into properties: if the square is a king in check, if the square is selected, if the square is hovered, and "otherwise".
5. If the user attempts to move a pawn into the final rank (necessitating a pawn promotion move), then you must show a new window asking them to select a promotion for the pawn. The window must follow these specifications:
   (a) The window's constructor should be passed the `ChessViewModel` used by the View, as well as the start and end positions of the attempted pawn move.
   (b) The window has no title bar, no control menu (upper left corner), no minimize/maximize/close buttons (upper right corner), and is not resizable. Play with `Window` properties to achieve this effect.
   (c) The window has a single label that says "Promote pawn to:", followed by the four valid promotion target pieces. The pieces should be colored according to the current player's turn, and sorted in increasing order by `GetPieceValue`.
   (d) Hovering over one the pieces highlights it with a different background color.
   (e) Clicking a piece applies a move to the ViewModel to do the selected promotion, then closes the window.
6. The View code should not have any direct interaction with the Model code (`ChessBoard`). It should use data binding to properties of the ViewModel whenever appropriate.

What You Need To Do

1. **Obtain the starter code for the project**. Follow the link on BeachBoard to the Project 2 Signup page on GitHub. Click the "Accept this assignment" button and follow whatever instructions are required of you to initialize the project. You will end up with a **private** repository named

`project2-yourgithubname` in the `csulb-cecs475-2020sp` GitHub organization where the lecture notes are stored. You will then **clone** that repository to your development machine.

2. **Create chess-specific projects in the solution**. See below. All of the work you do will be in Cecs475.BoardGames.Chess.WpfView, **except** for modifying the GameChoiceWindow.
3. Create the `ChessViewModel` class skeleton as C# class. Start by defining a `ChessSquare` helper class, then generate the list of 64 objects so the View can bind to the that list. Implement placeholders for any `IGameViewModel`-required members.
4. Create the `ChessView` layout skeleton. (Create this file using Visual Studio to add a new WPF -> WPF User Control.) Create the ViewModel as a static resource. Create the uniform grid that binds to the `ChessSquare` collection of the ViewModel and generates the 64 squares. Make each square a single solid color with no piece image to start.
5. Implement the `IWpfGameFactory` interface as class `ChessGameFactory`. Return `null` for the converters (for now). Return an appropriate value from `CreateView()`.
6. Modify `GameChoiceWindow.xaml` so that it:
   (a) Adds a new `xmlns:chess` in the `Window` tag, under `xmlns:othello`. Read your work closely when you do this.
   (b) Adds a new entry in the Window Resource array, of type `chess:ChessGameType`.
7. Now, launch the application and make sure you can select Chess to see a blank chessboard. You can now test your work after implementing each additional requirement.
8. Write a multiconverter to convert a `ChessSquare` to a `SolidColorBrush`. Use a multibinding to bind all the properties of the `ChessSquare` that you need in order to determine its background color. Start out by only returning a dark/light color based on position; work on the "selected" / "hovered" colors after you incorporate those features.
9. Write a converter to convert a `ChessSquare` to an `Image` control by loading the appropriate image from your project resources. Use the converter to finally draw pieces on your grid.
10. Create an event handler for the user clicking on a square. Write an implementation that "selects" the given square if it belongs to the current player. (Hint: `ChessSquare` needs a `IsSelected` property. A `Border`'s `DataContext` is bound to the `ChessSquare` it represents.) You may need to add member variables to the view class to keep track of "what Square is currently selected". Update your color multiconverter for selected squares.
11. Create an event handler for the mouse entering and leaving squares, that highlights them if they are possible moves that start at the already-"selected" square. Again, you will need a `IsHighlighted` property. Update your color multiconverter again.
12. Create an event handler for mouse-up on a square, which applies a move from the "selected" square to the clicked square if it is a possible move. You may need to add new properties or methods to the ViewModel.
13. Create a new WPF `Window` for pawn promotions, and show that window as a dialog (user must finish with the dialog before returning to the main program). Once the user selects a promotion piece, apply the corresponding move and close the dialog. You may need to add new properties or methods to the ViewModel.
14. Implement `CanUndo` and `UndoMove` in `ChessViewModel`; make sure the UI element for these works correctly.
15. Go back and implement any skeletons you left in `ChessViewModel`.
16. Write a converter to convert an integer into a string describing which player it represents. Construct and return an instance of this type in `ChessGameFactory`.

17. Write a converter to convert an integer into a string describing which player is winning, in the format "`{player} has a +{value} advantage`". Show "`Tie game`" for a value of 0. Construct and return an instance of this type in `ChessGameFactory`.

Creating the Chess Projects

The starter code comes with Visual Studio projects for Othello and TicTacToe. You need to create projects for Chess:

1. The starter code has a Cecs475.BoardGames.Chess.Model project with the same starter code as Project 1. You will need to copy your final chess code from Project 1 to this folder, overriding the files that are there. Do the same for the Cecs475.BoardGames.Chess.View project, even though we won't be using the console view in this assignment.
2. Create a New Project (right click on "src" in Visual Studio's Solution Explorer, Add, New Project). Select **C#** as the language, **Windows** as the platform, **Desktop**. Select the project type **WPF User Control Library** ***(.NET Framework)***. Give the project a Name of **Cecs475.BoardGames.Chess.WpfView**. Make sure the Location reads with ***root*\src**, where *root* is the root folder of your git repository. Click OK.
3. Right-click on Cecs475.BoardGames.Chess.WpfView in your Solution Explorer, and select Add, Reference... Select **Projects**, then **Solution** on the left. Check **Cecs475.BoardGames.Model**, **Cecs475.BoardGames.WpfView**, and **Cecs475.BoardGames.Chess.Model** as References.
4. Add **Cecs475.BoardGames.Chess.WpfView** as a Reference to Cecs475.BoardGames.WpfApp. The solution should build without errors.
5. Follow the guidelines in What You Need To Do. `ChessView`, `ChessViewModel`, `ChessGameType`, and all converters belong in Cecs475.BoardGames.Chess.WpfView.

Submitting For Grading

We will not use a robotic grader for this assignment; you will instead live-demo to me during lab. I will check your code to make sure it complies with this specification. If it does not, you will need to fix your errors and demo to me again, until I accept your work.