Project 3, Fascicle 1:
Reflecting Upon a Semester of Pain

Entire project due by May 21. **Suggested completion date for this Fascicle:** April 26

Overview

Project 3 involves 4 general milestones:

1. Load game types from a known directory at runtime, instead of forcing the application to know each suppoted game type at compile time.
2. Develop a Web service that lists and allows downloads of game assemblies that are compatible with our code.
3. Hook our WPF application into this Web service, downloading files at runtime so that they can be loaded automatically.
4. Develop an AI opponent so that any board game supported by our WPF app can be played by a single human against a sophisticated AI.

Rather than wait until we have learned everything needed to do parts 2 and 3, we will instead get started on part 1 with this fascicle.

Reflection

We have noted often that our board game application doesn't really care what kind of game being played. In Project 1, a console application used the `IGameBoard` and `IGameView` interfaces to handle the execution and display of a game, without directly talking to `ChessBoard`. In Project 2, a WPF application created GUI controls via the `IGameType` interface methods and added them to a general game-playing window, again without directly talking to othello, tic tac toe, or chess objects... with one exception.

The `GameChoiceWindow` of Project 2 hard-coded a lisit of supported game types in an array in the window's WPF resources. To add a new game to the application, we had to reference the game type's assemblies/projects in Visual Studio, reference the appropriate namespace in WPF, and construct an object of the appropriate `IGameType` class by name. All of these tasks have to be performed at compile-time, meaning we must know every single game type we want to support when we compile the program.

But what if we want our application to support *any* class that implements `IGameType`, regardless if we know about it at compile time? For that we need to use **reflection**: the ability of a .NET language to discover information about .NET types **at runtime**. Reflection will allow us to find all classes implementing `IGameType`, instantiate an object of those types, and add them automatically to a list of possible games.

Architecture

Recall that the CLI will, by default, look for assemblies in the same folder as an application that references them, or in the GAC if they can't be found. We aren't going to add games to the GAC, since they won't conceivably be useful to anyone else. Instead we will create a special folder alongside our WPF application that will contain DLLs of games that we can play. At program launch, we will use the `Assembly` class in C# to load any DLLs we find in that directory, then scan those DLLs for any type implementing `IGameType`. As long as we communicate the name of that folder, anyone will be able to develop a new board game implementation and drop it in our folder in order to let our application play that game.

Getting Started

1. Make a copy of your Project 2 to extend for this new assignment.

2. **Remove** the references to your Othello and Chess classes from the `Cecs475.BoardGames.WpfApp`'s References list. You should only have the `BoardGames.WpfView` and `BoardGames.Model` projects listed as references (besides the official .NET references).
3. **Clean your solution**.
4. In `GameChoiceWindow.xaml`, remove the `xmlns` references to your game projects. Remove the `Window.Resources` in its entirety. Change the `GamesList ItemsSource` to a `DynamicResource` with the same key. Compile the project.
5. In Windows Explorer, navigate to your project's `src\Cecs475.BoardGames.WpfApplication\bin\Debug` folder. Make a new folder here called `games`.
6. **By hand**, find the `.dll` assembly files for your four game-related projects (View and Model for the two game types). They will be in `bin\Debug` folders inside their repsective projects. (Shortcut: the View projects have references to Model projects and will copy the Model assemblies to their own `Debug` folder.) Copy the four .DLLs to the `games` folder you made above.
7. **Each time you change one of your Model or View files and recompile**, you will need to manually copy the associated DLL to the `games` folder. I can show you a way to automate this but it's complicated.
8. In the constructor for `GameChoiceWindow`, implement a procedure to find all `IWpfGameFactory` classes using reflection:
   (a) Declare a `Type` variable representing the Type of `IWpfGameFactory`.
   (b) Enumerate all files in the `games` folder. Use a **relative path**: .NET will know to look for a folder named `games` in the directory of the application's executable.
      i. For any file that ends with `.dll`, use `Assembly.LoadFrom` to load that assembly into the current AppDomain.
   (c) Enumerate over all assemblies loaded into the current AppDomain, using `AppDomain.CurrentDomain.GetAssemblies()`.
   (d) For each assembly, enumerate over all types defined in that assembly with `.GetTypes()`.
   (e) Filter the list of types to only contain those types `t` such that `IGameType` is assignable from `t`.
   (f) `GameChoiceWindow` doesn't want a list of `Type` objects, it wants you to construct an instance of the classes represented by each `IWpfGameFactory`. You will need to read how to invoke the default constructor of a `Type` object in order to map the sequence of types from (e) into a sequence of objects.
   (g) Set the sequence of objects from (f) as a dynamic resource of the window with the key `"GameTypes"`.
   (h) Profit! Run the application and see that the list of game types is now built automatically. Temporarily remove a View assembly from the `games` folder to see that that game type is no longer allowed. In the future, I will give you DLLs for Tic-Tac-Toe and Connect Four and your application will need to load them automatically without recompiling.