Project 3, Fascicle 4:
Connecting the Dots

Entire project due by May 21. **Suggested completion date for this Fascicle:** May 18

Async Game AI

As demonstrated in lab, we want our Minimax game AI to run in a background task so that the main WPF UI doesn't lock up while the AI is computing is best move. To do that, you need to run the `FindBestMove` inside of a task using `Task.Run`, then `await` the return value of that task before applying the AI's move and rebinding the UI. Any function that uses `await` must be marked as `async` in its signature, so your view model's `ApplyMove` method will need to be `async`, and you must change the return type to `Task` instead of `void`. That means the MouseUp event handler that calls `ApplyMove` must also be `async`, so it can `await` the result of `ApplyMove`. Easy!

However, you must be careful to prevent the user from making changes to the board while the AI is doing its work. The Undo button definitely modifies the board, but so too does moving the mouse (which probably calls `GetPossibleMoves` when the mouse enters a square, and that function calls `ApplyMove`....). This can't be allowed, so you will need to modify your UI code to lock down all UI interactions while the `ApplyMove` is running. I recommend setting the `IsEnabled` property of your window to false before calling `ApplyMove`, and then back to true after `ApplyMove` returns. Then add checks in all UI methods to return if `IsEnabled` is false.

Game Types Web Service

I have published a Web API service consisting of single API call returning a list of game types that I have created (beyond the 3 in your main application). The API can be found at `https://cecs475-boardamges.herokuapp.com/api/games`; a GET request to that URL will return a JSON array of "game type" objects. One example of such an object:

```
{
  "Name": "Connect Four",
  "Files": [
    {
      "FileName": "Cecs475.BoardGames.ConnectFour.Model.dll",
      "Url": -a long url-,
      "PublicKey": "68e71c13048d452a",
      "Version": "1.0.0.0"
    },
    {
      "FileName": "Cecs475.BoardGames.ConnectFour.WpfView.dll",
      "Url": -a long url-,
      "PublicKey": "68e71c13048d452a",
      "Version": "1.0.0.0"
    }
  ]
}
```

The goal for this fascicle is to use a REST client in your WPF application to download this list of game types, use the `Url` of the game's files to download the DLLs for the game types into your application's `games` folder, then load those files alongside the other game types that your application already supports.

Strong Names

I have signed my Connect Four assemblies with a private key. Any assembly that is strongly named can only reference other assemblies that are also strongly named; therefore, by signing `Cecs475.BoardGames.ConnectFour.Model`, I also had to sign `Cecs475.BoardGames.Model`, `Cecs475.BoardGames.WpfView`, and `Cecs475.BoardGames.ComputerOpponen` If you attempt to load my Connect Four assemblies, .NET will complain that the assembly references a strongly-named `Cecs475.BoardGames.Model` assembly, but the only file it can find is yours without a strong name. Therefore you will have to strong-name these three assemblies as well.

But you can't just use any private key to strong name the assemblies... you have to use the same one I used. I have posted that key on BeachBoard in the Projects folder. You will need to download it to your Project 3's root folder (where the Solution file is), then change the Properties of **all your projects** to use the key for Signing. This key is not sensitive or important, and won't be used after this semester, but for good practice, you should **not** add it to your GitHub repository. Make sure to copy all your individual game files back to your `games` folder, as they will now be referencing signed versions of those assemblies.

Loading Strong-Named Assemblies

You can still use `Assembly.LoadFrom` to load a strong-named assembly, but that's ripe for exploitation: a hacker could replace your game DLLs with new fake games that access private resources on your computer when loaded. You will want to provide strong information about the assemblies you are loading, and let .NET make sure that the files it finds are actually the ones you are looking for.

This requires several steps:

1. Open `App.config` in your WPF application. See the bottom of this url for how to tell .NET where to search for DLLs: https://msdn.microsoft.com/en-us/library/823z9h8w.aspx
2. Change your loading code to use `Assembly.Load` instead of `LoadFrom`. `Load` has different semantics; it expects a string of the form

   ```
   assemblyname, Version=-number-, Culture=neutral, PublicKeyToken=-token-
   ```

   where

   (a) `assemblyname` is the name of the file, **without** the extension `.dll`

   (b) `-number-` is a version string, like `1.0.0.0`

   (c) `-token-` is the public key token of the file, which is listed in the example JSON above, assuming you used the same private key as me

Downloading the Game List

1. Add the NuGet package `RestSharp` to your WPF application. You may also want `Netwonsoft.Json`.
2. Add a new `Window` to your WPF application. This window will show a single label "Loading games, please wait...". When the window has loaded (hint: hook an event handler to the `Load` event of the window itself), it should use a `RestClient` as shown in lecture to connect to the API URL given above to download the game list **asynchronously**, using `async` methods and the `await` keyword.
3. Iterate through each game in the game list (don't assume there is only one game, as there is currently). For each game, you will need to construct a `WebClient` object and use the `DownloadFileTaskAsync` method to download the URL associated with each file in the game. (The documentation for `WebClient` will help.)
4. When the downloads have finished, close the "Loading" window and show the game choice window.
5. Change your WPF application so that the "starting" window is your Loading window, not the game choice window. Hint: `App.xaml`.
6. That's it for project 3!

Demoing Project 3

To demo your project, you will show me:

1. That your `games` folder does not have any Connect Four files in it.
2. That your assemblies are strong-named with the class private key.
3. That your app now shows a Loading screen that downloads the Connect Four files from the class web API.
   (a) This screen must use async methods correctly, so that it can be moved, resized, or even *closed* while the downloads are happening.
   (b) Your loading method must use strong name information to load the assemblies.
4. That your app can now play Connect Four against a computer AI.
5. That your app can play its other games against computer AIs, using async tasks so that the main UI does not freeze while the AI's moves are calculated.
6. That your AI is using alpha-beta pruning.