CAB402 Programming Paradigms

Assignment 1

Due: 18th April 2019

Worth: 30%

The purpose of this assignment is to develop your functional programming skills by developing a non-trivial application using F#. The principal focus will be on learning to program in a "*pure*" functional style – i.e. without making use of any mutable state. In order to contrast this pure functional style of programming with the more traditional *imperative* (impure) and *object-oriented* paradigms, you will develop three separate implementations of the same software component:

1. A pure functional implementation using F#
2. An impure F# implementation that uses mutable state to improve performance
3. An object-oriented C# implementation

Game Theory

Computers can now play many games (such as chess) better than even the best human players. They do so using a branch of Artificial Intelligence called *Game Theory* (https://en.wikipedia.org/wiki/game_theory). Game Theory has also been used in more serious domains to help make optimal business and military decisions. Game Theory is most commonly applied in so called "*zero-sum*" games, i.e. for games in which you are competing against an opponent and a potential move that is "*good*" from your perspective is "*bad*" from their perspective and vice versa. In other words there are no "*win-win*" scenarios.

Minimax Algorithm

The basic algorithm used for determining the optimal move in such zero-sum games is called *Minimax*. It involves exploring a *search tree* that enumerates all possible moves you could make, and for each of those, all possible moves that your opponent could make, and for each of those, all possible moves you could then make and so on. For simple games like Tic-Tac-Toe it is possible to consider all possible combinations of moves all the way to the end of the game, where either one player has won or it is a draw. These ended games becomes the leaf nodes of the search tree. It is easy to assign a *score* to these leaf nodes, we simply assign a score of +1 if we win, -1 if we lose or 0 if it is a draw. We then use these leaf node scores to assign a score to the nodes higher in the search tree. If we are at odd level of the search tree that corresponds to where we get to choose our move then clearly we want to select the move that is best from our perspective, so we choose the branch leading to the child node that has the highest score and assign that highest score as the computed score for this parent node. Similarly, if we are at an even level in the search tree that corresponds to where our opponent gets to choose their move then we assume that they will play intelligently/optimally and choose a move/branch that is best for them, which is the one that will be worst from our perspective. So, at the even levels we are choosing the maximum score, while at the odd levels we are choosing the minimum score – hence the name *Minimax* (https://en.wikipedia.org/wiki/minimax). See Figure 1 for an example of how scores are computed in a bottom up fashion based on the given leaf node scores.

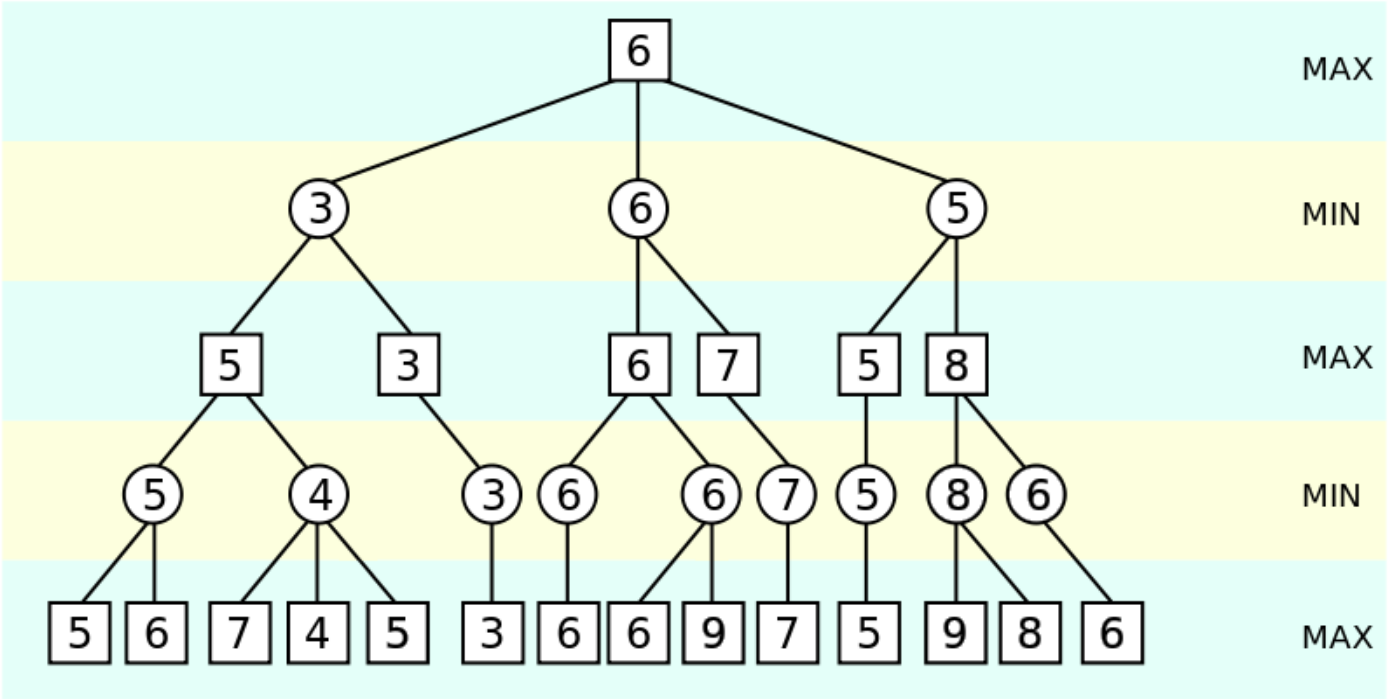


Figure 1 https://en.wikipedia.org/wiki/File:AB_pruning.svg

Heuristic Scores

In simple games like Tic-Tac-Toe we are able to explore all possible combinations of moves all the way to the end of the game – so we are guaranteed to always play optimally and never lose. In more complex games such as chess there are too many possible combinations of moves to explore all the way to the end of the game. We can still apply the same basic process of building a Minimax search tree, however rather than extending all the way to the end of the game, we just stop after some fixed number of levels. The leaf nodes in this search tree do not necessarily correspond to the end of the game, so it is not always possible to compute a score that perfectly captures which game situation is best. We instead use a heuristic score that aims to approximately capture how good a given game situation is from the perspective of one of the players. In chess for example, we would typically assign a different weight to each of the pieces we have remaining, for example a queen might be worth *1000* points, while a Knight is only worth *350* points (https://www.chessprogramming.org/point_value). More sophisticated scoring functions would take into account not just the pieces that each player has, but how they are arranged in terms of controlling various regions of the board.

Tic-Tac-Toe

In this assignment, we will apply the Minimax algorithm to the simple game of Tic-Tac-Toe (https://en.wikipedia.org/wiki/tic-tac-toe) where it is possible to consider all possible combinations of moves to the end of the game, so we do not need to worry about setting a maximum search depth and our heuristic scoring function will give us a perfect evaluation for the leaf nodes.

Skeleton Solution

You have been provided with a skeleton solution that you are required to use. It consists of one Visual Studio Solution that contains nine (9) separate Visual Studio projects. The first Visual Studio project is for an F# class library called GameTheory. The `GameTheory` project contains an F# file called `GameTheory.fs` that contains a function called `MiniMaxGenerator`. Your first task it to complete the implementation of this function using a pure functional style – i.e. without making use of any mutable state. This function is designed to be generic, to work for all possible zero-sum games (e.g. Chess), not just Tic-Tac-Toe. `MinimaxGenerator` in GameTheory is a *higher-order function*, it takes functions as inputs and produces a `Minimax` function as its output. The functions provided as input are used to compute:

- a heuristic score for any game situation,
- determine which player's turn it is in a given game situation,
- whether a game is over,
- enumerate all possible moves from a given game situation
- apply a move to a game situation to create a new game situation

The system is kept pure by never changing a game state. We instead have an `applyMove` function which creates a new game state based on a previous game state and a particular move. The previous game state does not change, it just gives rise to a new (separate) game state. This approach is central to functional programming.

The Minimax function that is returned is of type: (`'Game->'Player->Option<'Move>*int`), i.e. a function that takes a current game state and a player (from who's perspective we are choosing the best move) and returns a tuple containing the best move (if one exists) and the score associated with this game state. If we are at the end of the game then the best move will be *None* as there are no further legal moves possible once the game is over.

Unit Tests

To help you test if your `MiniMaxGenerator` function has been implemented correctly we have provided you with some unit tests in one of the Visual Studio Projects called `GameTheoryTests`. In order to run these tests within Visual Studio just build the solution, open the "`Test Explorer`" window (Test Menu/Windows/Test Explorer) and run the `GameTheoryTests`. These tests are based on the search tree in Figure 1.

Alpha-Beta Pruning

The problem with MiniMax is that it explores an exponential number of search tree nodes (if at each game situation there are *m* possible moves then a search tree of depth *n* will contains m^n nodes). However, if we are clever we can avoid exploring many of these nodes while never missing the best possible move. The trick is to introduce two new parameters *alpha* and *beta* to our recursive `MiniMax` function (https://en.wikipedia.org/wiki/alpha-beta_pruning). The parameter *alpha* is used to keep track of the lowest possible score that this node might be given and the parameter *beta* is used to keep track of the highest possible score that this node might be given. If we are applying alpha-beta pruning to a Tic-Tac-Toe game where the heuristic score function will always return *+1*, *-1* or *0*, so the initial call to `Minimax` will give *alpha* a value of *-1* and *beta* a value of *+1*. However, as we make recursive calls and get deeper into the search tree, the lower bound *alpha* and the upper bound *beta* become closer together. For example if we are at a level in the tree where we are maximizing and come across a move with a score *s* greater than the lower bound *alpha*, then *s* will become the new *alpha* lower bound. Similarly if we are minimizing and come across a move with a score *t* less than the upper bound *beta*, then *t* will become the new upper bound *beta*. If at any stage *beta* becomes less than or equal to *alpha* then we can stop searching the remainder of the child branches as we have already determined the best possible move from that node. In this way we can prune away significant segments of the search tree without giving up on finding the optimal

move. See Figure 2 for an example of how alpha beta pruning allows us to avoid exploring the nodes marked gray. For Tic-Toe-Toe, rather than searching *526,906* nodes using basic MiniMax, we instead need to explore only *16,087* nodes – resulting in a massive time saving.

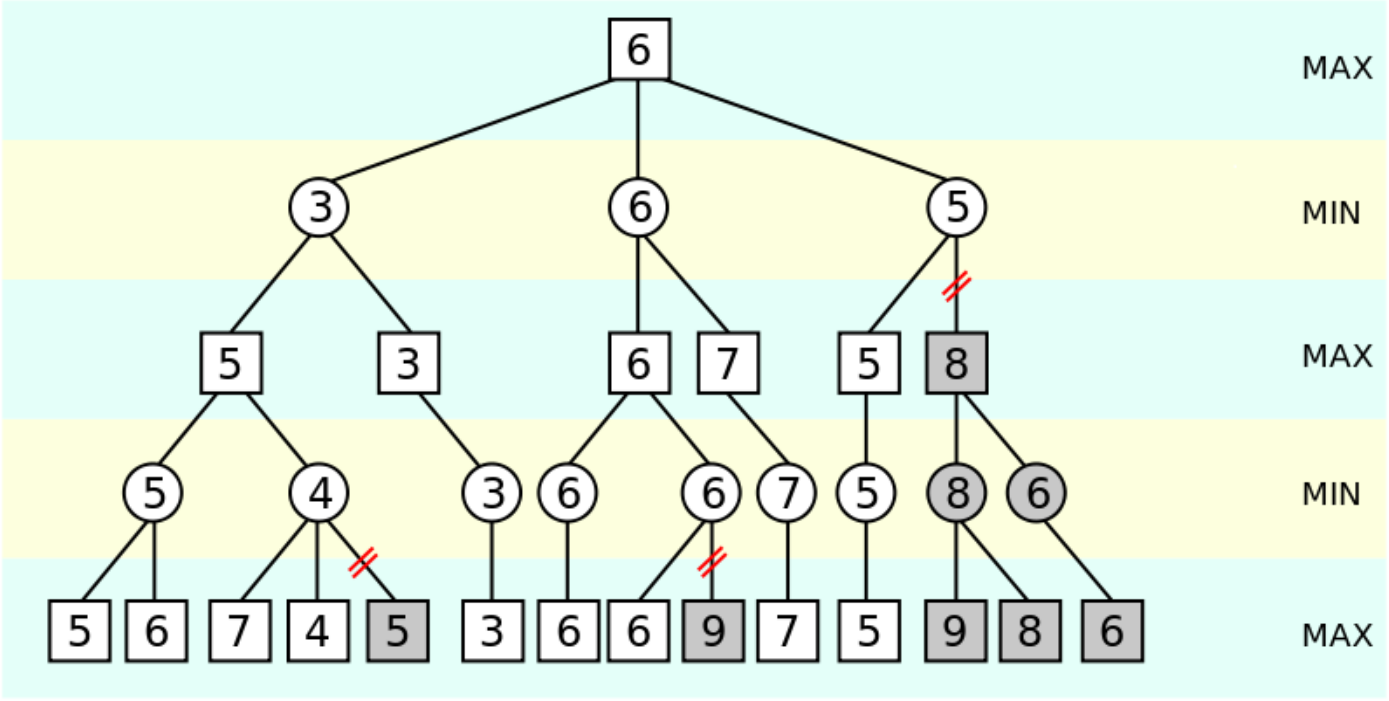


Figure 2: Alpha Beta Pruning

Your second task is to implement the `MiniMaxWithAlphaBetaPruningGenerator` function in a pure functional style.
[Tip: this is actually one of the more complex tasks in the assignment – so you may wish to defer it until you have completed some of the simpler tasks that follow].

Tic-Tac-Toe Model

Next we will use these generic Game Theory functions to help implement a Tic-Tac-Toe game. A classic Tic-Tac-Toe game is played on a 3 x 3 grid, but we generalize the game slightly to allow it to be played on a grid of size N x N (where N is called the "*size*" of the game). To win the game, a player must complete an entire straight line of N squares. The line can be one of N possible rows, N possible columns or 2 possible diagonals. Your code, therefore should not contain the *magic number 3* – everything should be based on the more general case of an N x N board.

In the `FSharpTicTacToeModel` project you will find an F# file `TicTacToePure.fs` that contains skeleton implementations for many of the Tic-Tac-Toe related functions that you will need to pass to the Minimax function in the Game Theory library. The `FSharpPureTicTacModel` module also contains functions `Minimax` and `MinimaxWithPruning` that you will implement by calling the `MiniMaxGenerator` and `MiniMaxiWithAlphaBetaPruningGenerator` functions from the `GameTheory` library. Before you implement those functions you will need to first design the F# data structures that you will use to represent the state of a Tic-Tac-Toe *game*, a Tic-Tac-Toe *move* and a Tic-Tac-Tac *player*. The Tic-Tac-Toe game state includes not just the state of the board (and its size), but also which player's turn it is. The Tic-Tac-Toe moves are represented via the (*row*, *column*) *coordinates* of each square, with the top left square having coordinates (0, 0). In Tic-Tac-Toe the two players are *Nought* and *Cross*.

`TicTacToePure.fs` also contains a helper function called `GameOutcome` that you must implement. It takes a game as input and returns a `TicTacToeOutcome` as defined in the `TicTacToeInterfaces` project. `TicTacToeOutcome` is an F# discriminated union type designed to convey the outcome of the game. If one of the players has won then the outcome will be `Win` with the `winner` field set to the player who won and the `line` field telling us the coordinates of the squares corresponding to the winning line (row, column or diagonal). If neither player has won then the outcome will be either `Draw` or `Undecided`. The game should be considered a draw if every possible line (row, column or diagonal) contains both a Cross and a Nought. The `GameOutcome` function should be used to help implement both the `GameOver` function and the `HeuristicScore` function. The `GameOutcome` function should be implemented with the help of the `Lines` function and the `CheckLine` functions. The `Lines` function returns a sequence containing all of the lines on the board: horizontal, vertical and diagonal. The `CheckLine` function takes as input both a game and a single line and tells us (based on considering just that one line alone) if the game has been won, drawn or undecided. Other helper functions should also be created as needed.

Everything in the `TicTacToePure.fs` must be implemented in a pure functional style, i.e. without using any mutable state.

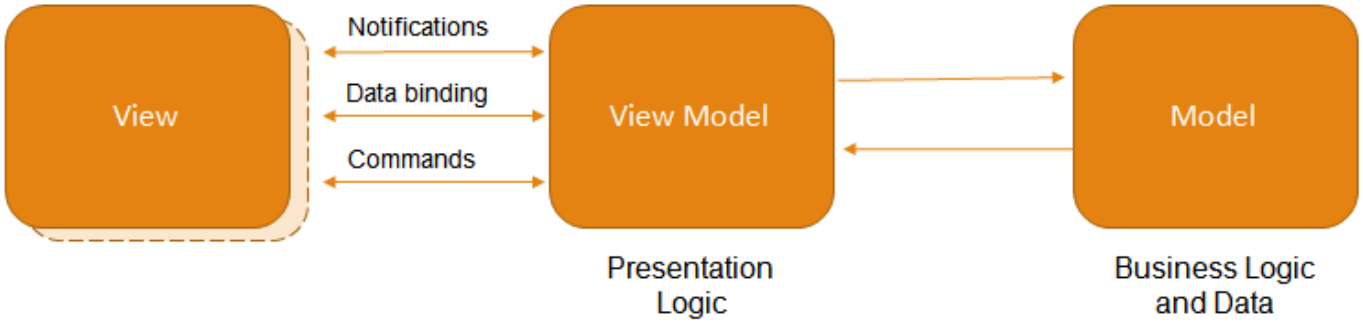
The `TicTacToeModelTests` project contains unit tests that you can use to help test your code.

MVVM (Model-View-ViewModel)

So far we have implemented just the basic functions needed for a game of Tic-Tac-Toe. Those functions need to be incorporated into a GUI app that will actually play a game of Tic-Tac-Toe against a human opponent. The GUI app will use the *Model-View-ViewModel* (MVVM) architectural pattern. See:

- https://en.wikipedia.org/wiki/Model-view-viewmodel,
- https://docs.microsoft.com/en-us/windows/uwp/data-binding/data-binding-and-mvvm,
- https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm,
- https://blogs.msdn.microsoft.com/msgulfcommunity/2013/03/13/understanding-the-basics-of-mvvm-design-pattern/)

The GUI app is split into 3 components: The *Model*, The *View* and the *ViewModel*. The model provides data structures for representing objects in the domain as well as basic "*business logic*" for the application domain. In our case, the application domain is Tic-Tac-Toe, so the F# component that we have implemented so far will be used as the Model in our MVVM application. The View is the part of the application that determines what the user sees on the screen and what user interactions (clicks, keyboard, gestures, etc) are allowed. The ViewModel acts as a bridge between the View and the Model:



https://documentation.devexpress.com/WPF/15112/MVVM-Framework

The ViewModel provides an abstract view of the User Interface, it exposes information via *properties* that should be presented to the user (somehow) and *commands* which are the different abstracted operations that the user might trigger via the user interface (somehow). For example, our Tic-Tac-Toe ViewModel exposes a property called `Message` that is designed to provide the user with information about who's turn it is and if the game has been won or drawn. Our Tic-Tac-Toe View chooses to display that property via a Label control in the footer of the main window. Our Tic-Tac-Toe ViewModel also exposes commands for starting a new game and for when the user selects a square for their next move. Our Tic-Tac-Toe View chooses to have a menu item at the top of the main window that when chosen triggers the `NewGame` command of the ViewModel, whereas the Select square command is triggered when the user clicks one of the square user controls that is empty.

The ViewModel also interacts with the Model, for example when the `NewGame` command is triggered it calls the `GameStart` function of the Model. Our model is implemented in a pure functional style, so it contains no mutable state. The ViewModel component however is stateful. It contains a private property called `gameState` that uses the Game type defined in the Tic-Tac-Toe Model component. When either the human or the computer makes a move, the ViewModel calls the `ApplyMove` function from the Tic-Tac-Toe Model component to obtain a new Tic-Tac-Toe game object and overwrites the old value of the `gameState` property. The `gameState` property of the Tic-Tac-Toe ViewModel is mutable, while everything in the pure F# implementation of the Tic-Tac-Toe Model remains immutable.

One of the nice things about having most of the presentation logic contained in the ViewModel and abstracted from the actual user interface is that we can easily perform automated unit tests and integration tests on the ViewModel as it does not involve a GUI. Our automated tests can directly trigger commands corresponding to abstract user actions and then observe how the state of the game and user interface changes by inspecting the properties that get directly bound to user interface elements. The `TicTacToeViewModelTests` Visual Studio project contains unit tests and integration tests designed to test the ViewModel.

Multiple View Implementations

The View is the only component of the application that depends on the technology used to create the user interface. In this app we choose to implement the View component using the Microsoft *Windows Presentation Framework* (*WPF*). The View needs to know about the ViewModel, but the ViewModel doesn't need to know about the View. So, it is possible to have multiple alternative implementations of the View component. We could have implementations of the View using:

- *WPF* for use on Microsoft Windows Desktops,
- *UWP* for use on newer Microsoft platforms such as Windows 10, Xbox, Surface or Mobile devices,
- *Xamarin* for cross platform Android and IOS devices

We could even create an implementation of the View that had a command line user interface rather than a graphical user interface.

All of these different implementations of the View could reuse the exact same ViewModel and Model without needing to make any changes. There is nothing in the ViewModel or the Model that relates to WPF, UWP or Xamarin. In implementing the ViewModel and the View we have specifically targeted the new .NET Standard 2.0 (https://docs.microsoft.com/en-us/dotnet/standard/net-standard). This open standard is supported by both Microsoft's .NET Framework, but also by the open source .NET Core (https://en.wikipedia.org/wiki/.NET_Core) which works across multiple platforms including Windows, IOS and Linux. Only the WPF View component is tied to the Microsoft Windows platform, but it could be easily replaced by a cross-platform alternative.

View Binding

The ViewModel doesn't know about the View, but the View does need to know about the ViewModel. We generally try to connect the View to the ViewModel in a declarative manner. Firstly, the layout and components of the View's user interface is specified in a declarative manner using a user interface mark-up language called *XAML* (pronounced Zamel)

https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/xaml-overview-wpf

It is a tag based language (similar to HTML) that is used to specify which user interface elements are contained on the page and how they are laid out relative to one another.

See for example `TicTacToeMainWindow.xaml` in the `TicTacToeWPF` Visual Studio project. We will see, for example, a Label named `MessageLabel` that is docked to the bottom of the main window. In the corresponding C# code behind file (`TicTacToeMainWindow.cs`) is the declarative code that binds this `MessageLabel` control of the View to the `Message` property of the ViewModel:

```
this.Bind(this.ViewModel, vm => vm.Message, view => view.MessageLabel.Content);
```

The `Message` property of the ViewModel is annotated with the *Reactive* attribute:

```
[Reactive] public string Message { get; set; }
```

Which means that whenever the set method of this property is called, it will automatically trigger a property changed event which will automatically notify all subscribed property change listeners (including the View), so the text displayed in the label control on the view will be automatically updated whenever the ViewModel changes the value of its `Message` property.

Multiple Model Implementations

Just like we could have multiple alternative implementations of the View (that all use the same ViewModel and View), similarly we could have multiple alternative implementations of the Tic-Tac-Toe Model (that all use the same ViewModel and Views). In our case we are actually going to create multiple alternative implementations of the Model for the purpose of comparing and contrasting different programming paradigms. We already have one pure F# implementation of the Model. You will also create a separate impure F# implementation of the Model (in the provided `TicTacToeImpure.fs` file) in which you will use mutable state to try to create a more efficient (faster) alternative implementation.

The `ApplyMove` function of your pure F# implementation was of type: `GameState->Move->GameState`, however in your impure F# implementation you can make your `GameState` type mutable, i.e. when you apply a move, rather than returning a new `GameState`, the `ApplyMove` function changes the existing `GameState`. The impure `ApplyMove` function would be of type: `GameState->Move->Unit` (i.e. the function returns "`void`" rather than a new `GameState`. If that is the case, then you'll probably also need to add an `UndoMove` function of type `GameState->Move->Unit` that undoes the state change (taking the square that was selected and changing it back to empty). Your impure F# implementation will therefore not be able to reuse the existing `Minimax` functions from the `GameTheory` library (since these are implicitly designed to work with immutable game state). So you will need to implement a new `Minimax` function (with alpha beta pruning). The implementation of this `Minimax` function can itself use mutable state (for example to progressively update the alpha and beta parameters) in order to create a more efficient implementation. It is not necessary, however, for this new `Minimax` function to be placed in a separate project, nor made generic to work for games other than Tic-Tac-Toe.

You will also create a C# implementation of the Model (in the `CSharpTicTacToeModels` project) designed to showcase object-oriented design and implementation best practices.

Since some of these model implementations are functional and others are object-oriented, in order for the ViewModel to make use of these different Models, we need to apply the Façade design pattern (https://en.wikipedia.org/wiki/facade_pattern) in order to provide a uniform interface. The `TicTacToeInterfaces` Visual Studio project defines such an interface. Each implementation of the Tic-Tac-Toe model will have its own data structures to represent: the current state of the Tic-Tac-Toe game (which might or might not be mutable), a Tic-Tac-Toe move and a Tic-Tac-Toe player. The Façade interface for the Tic-Tac-Toe Model is therefore a generic interface parameterized with the types used to represent the Tic-Tic-Toe Games, Moves and Players. Because the ViewModel also need to deal with these different Tic-Toe-Toe Games and Moves in a uniform manner, we also define an `ITicTacToeGame` and `ITicTacToeMove` interface that each of those types are required to implement (as appropriate).

At the bottom of the TicTacToePure.fs file you will find implementations for these Facades. We actually want to expose two separate implementations, one that uses the basic Minimax algorithm and a second which uses the more efficient alpha beta pruning algorithm. These two implementations are the same except for that function, so we define an abstract base class called Model from which BasicMiniMax and AlphaBetaPruning are derived. Note, that these class definitions at the bottom of the file are provided only to provide a façade to the ViewModel. In particular, we are not attempting to be object oriented in our implementation of the Pure F# Model – *it consists purely of functions – not methods*. Also, note how the Façade classes don't contain any real implementation code themselves. All the work is done in the F# functions defined above and the Façade classes simply call those functions. The same should be true of the Façade class in the impure F# implementation and in the C# implementation. In the case of the C# façade class, it should simply call methods defined in the other domain classes – following principles of object-oriented design.

The ViewModel class needs to hold a reference to one of these TicTacToeModel interfaces, however the interface is generic (with type parameters for the Game, Move and Player types). In order to avoid making the ViewModel a generic class, we instead split the ViewModel implementation into two parts. The fields and methods that need to deal with values of type TicTacToeGame, TicTacToeMove and TicTacToePlayer are moved into a separate ViewModel *"child"* class. The child class is generic, but the parent ViewModel class is not generic. In the constructor of the ViewModel class, an instance of this child class is created corresponding to each of the different Façade classes providing alternative implementations of the model:

```
// add pure F# model using basic MiniMax algorithm
models.Add(new TicTacToeViewModelChild<FSharpPureTicTacToeModel.GameState,
                                       FSharpPureTicTacToeModel.Move,
                                         FSharpPureTicTacToeModel.Player>(this,
                                                      new FSharpPureTicTacToeModel.BasicMiniMax()));

// add pure F# model using MiniMax with alpha beta pruning
models.Add(new TicTacToeViewModelChild<FSharpPureTicTacToeModel.GameState,
                                       FSharpPureTicTacToeModel.Move,
                                       FSharpPureTicTacToeModel.Player>(this,
                                         new FSharpPureTicTacToeModel.WithAlphaBetaPruning()));

// add impure F# model using MiniMax with alpha beta pruning
models.Add(new TicTacToeViewModelChild<FSharpImpureTicTacToeModel.GameState,
                                       FSharpImpureTicTacToeModel.Move,
                                       FSharpImpureTicTacToeModel.Player>(this,
                                         new FSharpImpureTicTacToeModel.WithAlphaBetaPruning()));

// add impure C# model using MiniMax with alpha beta pruning
models.Add(new TicTacToeViewModelChild<CSharpTicTacToe.Game,
                                       CSharpTicTacToe.Move,
                                       CSharpTicTacToe.Player>(this,
                                         new CSharpTicTacToe.WithAlphaBetaPruning()));
```

This list of models contained in the ViewModel is bound to a Combo Box control in the view so that the user can dynamically select which implementation of the model they wish to make use. Whenever a new Model is chosen, a new game is automatically started, since the existing game state contained within the previous ViewModel child would not be compatible with the newly selected model. This method of abstracting what is provided by a particular component so that different implementations can be easily substituted is referred to as "*dependency injection*" (https://en.wikipedia.org/wiki/dependency_injection).

Node Counting

In order to allow us to compare the performance statistics of these different model implementations we inject some diagnostic code into our model implementations. (Such diagnostic code would typical be commented out in the final "*production*" version of a system). The diagnostic code that we inject is designed to count how many search tree nodes we explore when performing the minimax algorithm. In particular, we want to be able to observe how many fewer nodes are explored if we enable the alpha beta pruning implementation. This diagnostic code is inherently stateful, so to avoid desecrating our pure F# implementation we wrap this code into a separate component contained in the NodeCounter Visual Studio project. Each of the model implementation projects will contain a reference to this project. All four implementations of the MiniMax function must make a single call to the Reset() method before performing the actual search and must call the Increment() method each time a new node of the search tree is explored. The ViewModel then uses the Count property of the NodeCounter to retrieve the total number of nodes visited after the call to the MiniMax function.

The WPF view displays diagnostic information containing both the number of nodes explored and the time elapsed while searching. We expect the number of nodes explored to be the same regardless of whether we are using the F# pure version, the F# impure version or the C# version (assuming that in each case we are comparing the variant making use of alpha beta pruning), however the execution time for the C# implementation is likely to be faster and the F# impure version should be faster than the F# pure version.

To run the final application, set TicTacToeWPF as the StartUp project and press Start.

What to Submit

1. A **1-2 page PDF report** *discussing your experience* and comparing the three approaches (F# pure functional, F# impure functional and C# object-oriented imperative). Your comparison should include a *discussion* on the efficiency and effectiveness of the three approaches. You should discuss which was easier to program, which code is easier to read, understand and maintain, which is more concise and which produces more efficient code. Provide measures were applicable to support your claims.

2. A **zip file** containing your entire Visual Studio Solution folder.

   You should work with the skeleton visual studio solution provided and make changes only to the following projects:

   - `GameTheory`
   - `FSharpTicTacToeModels`
   - `CSharpTicTacToeModels`

   Note:

   - All 41 occurrences of `NotImplementedException` should be replaced by an actual implementation.
   - Extra helper functions and methods should be added as needed. Complex functions and methods should be broken down into simpler functions. Ensure you use meaningful names for all functions, methods, parameters and variables.
   - Ideally, all 1959 tests should pass.

   To reduce file size when submitting you should remove the following:

   - `.vs` folder
   - `packages` folder
   - `bin` folders
   - `obj` folders
   - Visual Studio solution files except for `TicTacToe.sln`

   Everything remaining should be added to a compressed zip file with a name of the form `n1234567.zip`

   Note: **this must be a zip file**, not a tar file, not a rar file, not an iso file, etc.

   When we mark your assignment we will:

   1. Unzip your zip file
   2. Double click on `TicTacToe.sln` to open your solution in Visual Studio 2017
   3. Perform a Rebuild Solution (this should cause any required packages to download as necessary).
   4. Run all of the tests via the Test Explorer Window
   5. Start your `TicTacToeWPF` application and test that it behaves as expected, selecting in turn each of the models from the drop down combo box, trying different game sizes and observing the performance diagnostics.
   6. We will then browse your source code and assess it against the assessment criteria.