OBJECT CALISTHENICS ♥ F#

Comparing TicTacToe implementations in C# and F#

The C# implementation is the one we created during the course. It was implemented in a test-driven manner while respecting the rules of Object Calisthenics.

The F# implementation is based on the C# implementation in order to be as similar as possible while utilizing F#'s type system

OBJECT CALISTHENICS APPLIED TO F#

A lot of the rules of Object Calisthenics is directly applicable to F# which isn't that surprising given that F# is a "functional-first, general purpose, strongly typed, multi-paradigm programming language that encompasses functional, imperative, and object-oriented programming methods" - Wikipedia.

1. Only One Level Of Indentation Per Method

   Extracting methods and composing them can be done with the |> operator:

   ```
   user |> Authenticate |> Authorize |> SignIn
   ```
2. Don’t Use The ELSE Keyword

   Instead of using an early return we can just use types like Option, Result or Choice;

   ```
   user
   |> Option.map (fun user -> user.Name)
   |> Option.defaultValue "John Doe"
   ```
3. Wrap All Primitives And Strings

   ```
   type FirstName = FirstName of string
   ```
4. First Class Collections

   ```
   type Name = Name of string list with
       static member StartsWith letter users = …
   ```
5. One Dot Per Line
6. Don’t Abbreviate

   Not only can we avoid abbreviations, we can avoid PascalCasing as well:

   ```
   let ``This is an actual function name – call me with your name to get a greeting`` =
       sprintf "Hello, %s!"
   ```
7. Keep All Entities Small
8. No Classes With More Than Two Instance Variables
9. No Getters/Setters/Properties

C#

TicTacToe.cs

```
namespace src
{
    public class TicTacToe
    {
        private Player _currentPlayer = Player.X;
        private readonly Board _board = new();

        public Player GetCurrentPlayer() =>
           _currentPlayer;

        public void PlaceMarker(Square square)
        {
            if(_board.IsSquarePlayed(square))
                return;

            _board.PlaceMarker(square,
_currentPlayer);

           AlternatePlayer();
        }

        private void AlternatePlayer()
        {
            if (_currentPlayer == Player.X)
            {
                _currentPlayer = Player.O;
                return;
            }

            _currentPlayer = Player.X;
        }

        public Player GetWinner() =>
            _board.GetWinner();
    }
}
```

Board.cs

```
namespace src
{
    internal class Board
    {
        private Player[] _playedSquares = new Player[9];

        private Square[][] _winningConditions =
        {
            new[] {TopLeft, TopMiddle, TopRight},
            new[] {MiddleLeft, Center, MiddleRight},
            new[] {BottomLeft, BottomMiddle, BottomRight},
            new[] {TopLeft, Center, BottomRight},
            new[] {TopRight, Center, BottomLeft},
            new[] {TopLeft, MiddleLeft, BottomLeft},
            new[] {TopMiddle, Center, BottomMiddle},
            new[] {TopRight, MiddleRight, BottomRight}
        };

        public bool IsSquarePlayed(Square square) =>
           _playedSquares[(int) square] != Player.None;

        public void PlaceMarker(Square square, Player player) =>
           _playedSquares[(int) square] = player;

        public Player GetWinner()
        {
            foreach (var winningCondition in _winningConditions)
            {
                var player = GetPlayerAtSquare(winningCondition[0]);
                if (HasSamePlayer(winningCondition, player))
                    return player;
            }

            return Player.None;
        }

        private bool HasSamePlayer(Square[] winningCondition, Player player)
        {
            return winningCondition.Select(GetPlayerAtSquare).All(p => p == player);
        }

        private Player GetPlayerAtSquare(Square square) =>
           _playedSquares[(int) square];
    }
}
```

Player.cs

```
namespace src
{
    public enum Player
    {
        None = 0,
        X,
        O,
    }
}
```

Square.cs

```
namespace src
{
    public enum Square
    {
        TopLeft,
        TopMiddle,
        TopRight,
        MiddleLeft,
        Center,
        MiddleRight,
        BottomLeft,
       BottomMiddle,
        BottomRight
    }
}
```

F#

TicTacToe.fs

```
module FSharp.TicTacToe

open FSharp.Model

type TicTacToe() =
   let mutable currentPlayer = X
   let mutable playedSquares : PlayedSquares = []

   let tryGetFreeSquare (square : Square) =
        playedSquares
        |> List.map fst
        |> List.tryFind ((=)square)
        |> function
            | Some _ -> None
            | None -> Some square

   let placeMarker =
        Option.map (fun square ->
            playedSquares <- (square, currentPlayer) :: playedSquares
            square)

   let alternatePlayer : Square option -> unit =
        Option.iter (fun _ ->
            currentPlayer <- match currentPlayer with
                             | X -> O
                             | O -> X)

   member x.GetCurrentPlayer() = currentPlayer

   member x.PlaceMarker(square) =
        tryGetFreeSquare square
        |> placeMarker
        |> alternatePlayer

   member x.GetWinner() =
        tryFindWinner playedSquares winConditions
```

Model.fs

```
module FSharp.Model

type Player =
    | X | O

type Square =
    | TopLeft    | TopMiddle    | TopRight
    | MiddleLeft | Center       | MiddleRight
    | BottomLeft | BottomMiddle | BottomRight

type PlayedSquare = Square * Player

type PlayedSquares = PlayedSquare list

type WinCondition = PlayedSquares -> Player option

type WinConditions = WinCondition list

type TryFindWinner = PlayedSquares -> WinConditions -> Player option

let winConditions : WinConditions =
    let threeInRow row =
        List.map row >> List.choose id >> (function | [ player ; _ ; _ ] -> player |> Some | _ -> None)

   [threeInRow (function | TopLeft,    X | TopMiddle,    X | TopRight,     X -> X |> Some | _ -> None)
    threeInRow (function | MiddleLeft, X | Center,       X | MiddleRight,  X -> X |> Some | _ -> None)
    threeInRow (function | BottomLeft, X | BottomMiddle, X | BottomRight,  X -> X |> Some | _ -> None)
    threeInRow (function | TopLeft,    X | MiddleLeft,   X | BottomLeft,   X -> X |> Some | _ -> None)
    threeInRow (function | TopMiddle,  X | Center,       X | BottomMiddle, X -> X |> Some | _ -> None)
    threeInRow (function | TopRight,   X | MiddleRight,  X | BottomRight,  X -> X |> Some | _ -> None)
    threeInRow (function | TopRight,   X | Center,       X | BottomLeft,   X -> X |> Some | _ -> None)
    threeInRow (function | TopLeft,    X | Center,       X | BottomRight,  X -> X |> Some | _ -> None)
    threeInRow (function | TopLeft,    O | TopMiddle,    O | TopRight,     O -> O |> Some | _ -> None)
    threeInRow (function | MiddleLeft, O | Center,       O | MiddleRight,  O -> O |> Some | _ -> None)
    threeInRow (function | BottomLeft, O | BottomMiddle, O | BottomRight,  O -> O |> Some | _ -> None)
    threeInRow (function | TopLeft,    O | MiddleLeft,   O | BottomLeft,   O -> O |> Some | _ -> None)
    threeInRow (function | TopMiddle,  O | Center,       O | BottomMiddle, O -> O |> Some | _ -> None)
    threeInRow (function | TopRight,   O | MiddleRight,  O | BottomRight,  O -> O |> Some | _ -> None)
    threeInRow (function | TopRight,   O | Center,       O | BottomLeft,   O -> O |> Some | _ -> None)
    threeInRow (function | TopLeft,    O | Center,       O | BottomRight,  O -> O |> Some | _ -> None)]

let tryFindWinner playedSquares winConditions =
    winConditions
    |> List.map (fun winCondition -> winCondition playedSquares)
    |> List.choose id
    |> List.tryExactlyOne
```

F# TICTACTOE

- Very declarative
  - A player is either a cross or a nought

    ```
    type Player =
        | X | O
    ```
  - Squares can be formatted in a highly readable way

    ```
    type Square =
        | TopLeft    | TopMiddle    | TopRight
        | MiddleLeft | Center       | MiddleRight
        | BottomLeft | BottomMiddle | BottomRight
    ```
  - A played square can be expressed as a square with a player and all played squares becomes a no brainer

    ```
    type PlayedSquare =
        Square * Player

    type PlayedSquares = PlayedSquare list
    ```
  - A winning condition is something that given a set of played squares will return either some player or none

    ```
    type WinCondition = PlayedSquares -> Player option

    type WinConditions = WinCondition list
    ```
  - This leads us to the winner.A winner can be expressed as something that given a set of played squares and a set of winning conditions will return either some player or none

    ```
    type TryFindWinner = PlayedSquares -> WinConditions -> Player option
    ```
- No if.. then.. Else..

  -
    ```
    member x.PlaceMarker(square) =
            tryGetFreeSquare square
            |> placeMarker
            |> alternatePlayer
    ```

OBJECT CALISTHENICS ♥ F#

F# seems incredibly suitable to the rules of Object Calisthenics.
The focus of Object Calisthenics is "maintainability, readability, testability, and comprehensibility" and the 9 rules are almost built into F#.

1. Only One Level Of Indentation Per Method
   This is really to avoid the indentation arrow problem. F# has solved this with Option, Result and so on.
2. Don't Use The ELSE Keyword
   Solved with Option, Result and so on.
3. Wrap All Primitives And Strings
   Powerful type system with sum types, product types, aliases and tupples
4. First Class Collections
   Type system with static member functions
5. One Dot Per Line
   F# separates data from behavior thus removing the problem with knowing the friends of your friends entirely.
6. Don't Abbreviate
   ``This is a valid function name in F#. No abbreviations here``.
   As naming is the most difficult problem in software development it's nice not having to do them in PascalCase
7. Keep All Entities Small
   F# has removed almost all boiler plate code which helps to keep things small
8. No Classes With More Than Two Instance Variables
   Well this might feel like cheating; F# tries to avoid classes, objects and variables all together. But it really helps when that is in fact what you try to avoid.
9. No Getters/Setters/Properties
   With F# separating data and behavior and being immutable by default this becomes so easy

THANK YOU

I find that Object Calisthenics is a really helpful tool to use when refactoring. I can imagine it being useful when doing code-reviews as well.

Thanks to Allessandro and Marco for introducing us to it and guiding us along the way.

I really enjoyed this course and I hope I get to run and fly as well ☺

QUESTIONS?