# Software Engineering Design - Coursework
# Design and Implementation of Multiplayer Othello

Michal Srb and Thomas Rooney

December 3, 2012

## 1 Introduction

The task that we have been given is to design and implement a computer-based version of Othello, that is playable between at least two human players.

To implement this, we have decided to leverage web technologies to build a web application, where players can challenge one another and play a game of Othello. The application includes a virtual lobby and stack of one player's current games.

We have followed three main principles during the design and implementation of the project. Firstly, we focused on minimizing API to bare minimum for every interface. Secondly, we employed top-down programming - we always wrote caller's code first before writing a concrete implementation. And lastly, we followed Test-Driven Development technique when developing core logics. Hence, we followed the presented pattern when designing and implementing the main game logic of Othello:

1. Write example calling code - simple Command-Line interface UI
2. Write a mock implementation to test correctness of API design
3. Use Test Driven Development- writing tests for each feature (desing previously)
4. End up with lightweight and functional implementation

## 2 Testing

In accordance with Test Driven Development principles, one of the first stages around building the project was to produce a comprehensive test framework, which could be used throughout the project to ensure that the behaviour of the system, and its components, are correct. To do this, we have utilised the Jasmine Javascript testing framework, so that we could quickly write tests and maintain behaviour whenever we changed bits of implementation. We have created a spec for each of the core classes and used several tests for each method. We have also used Jasmine's feature of nested `beforeEach` setup calls.

```
C:\Code\othello>jasmine-node --coffee --matchall test
.........................
```

```
Finished in 0.047 seconds
28 tests, 42 assertions, 0 failures
```

## 3 Designing the Othello Core Game Logic

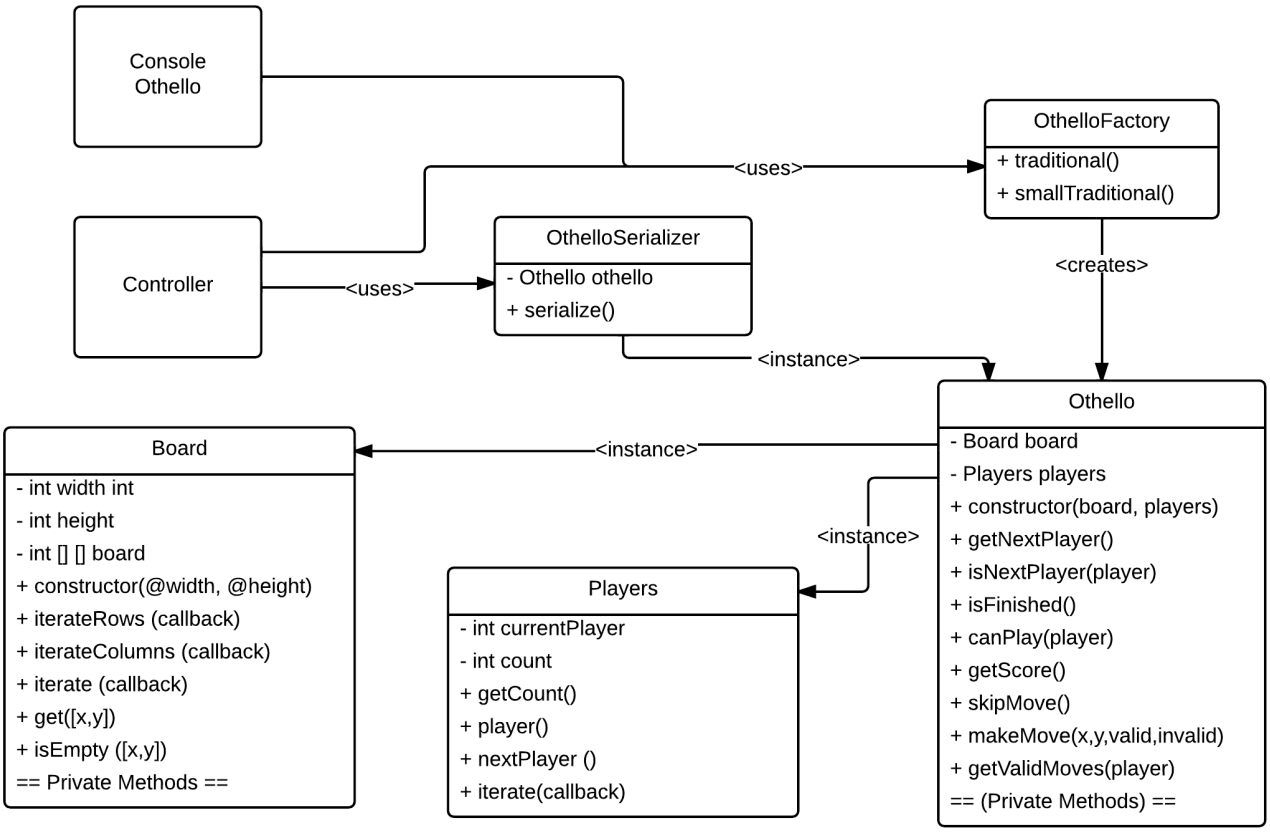
The Othello game logic is relatively simple, but we worked in a top-down style to design the classes. We first considered how to build a console interface to the game class, and what sort of IO should be available to play the game. The requirements for this class was:

- Notifying the player who's turn to play it is, or notifications that there are no valid moves and that a player is being skipped.
- Playing a move, given a move is valid
- Outputting the score of the game.
- Telling the players that the game has finished when it there no valid moves left.

Whilst building these methods, we produced a list of responsibilites that the main Othello class should have held:

- Keep track of which player's turn it is.
- Keep track of the stones on the game board.
- Decide on validity of moves.
- Keep track of game progress (score and finishing conditions).
- Change state of the game - enabling players to make moves.

Using this same principle, we recursively abstracted into the game logic, eventually producing a design that equates to the following diagram:




Utilising this design has given us a multitude of advantages. We have kept the design class general enough to use different board sizes and different numbers of players. We have separated the data representation of board and players, and appropriate accessor methods into respective classes. Through `Composition` we provided a single interface for executing the game.

We have utilised the `Iterator` pattern several times. The reason for this is it allows us to abstract away from the internal representation of the data at each stage of the object hierarchy. Doing this lowers the rigidity of the code, as no information about the representation matters for the caller's implementation.

We were also planning on creating a HistoryOthello class, allowing players to replay their game and undo moves. This could be implemented via the `Decorator` pattern, without any impact on the current game implementation.

The console version can be viewed and played via the `ConsoleOthello` class.

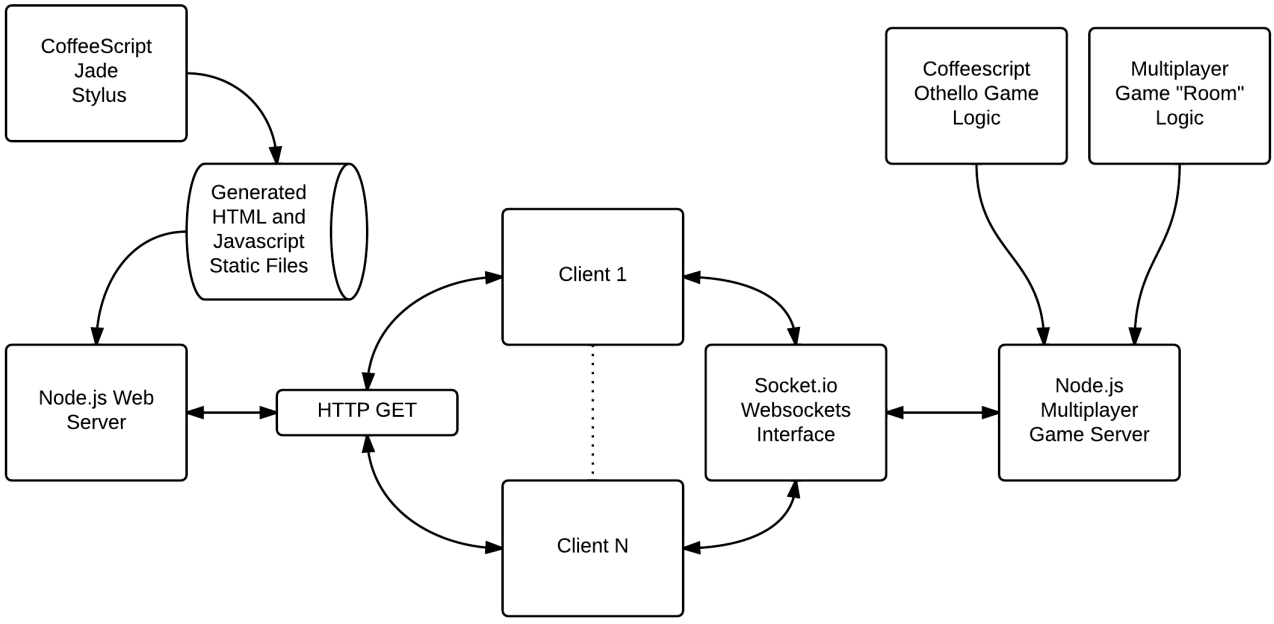
```
Michal(w),  where would you like to place next stone?
X:  7
Y:  4
You can't place your stone there!
Michal(w),  where would you like to place next stone?
X:  7
Y:  5
  0 1 2 3 4 5 6 7
0 w w w w w w w b
1 w w w w w w w b
2 w w w w w w w b
```

```
3 w w w w w w w b
4 w w w w w w w b
5 w w w w w w w w
6 w w w w w w w b
7 w w w w w w w b
Game has ended, final scores:
Michal: 57
Tom: 7
```

We have used composition to provide a Serializer class which takes any Othello instance and provides the game state representation to be sent to clients or save in memory. We have also used a pseudo Factory class to produce different Othello instantiations.

## 4 Othello on the Web

The next stage in the production of our gaming server was the design of networking logic, and implementation of a client side UI to play the game on, and challenge other players on the same web page. We have followed the `Model-View-Controller` architecture, whereas our core game logic became the Model, servers serve as Controllers populating the client Views in browser. Thanks to WebSockets we design a simple publish-subscribe scheme, where the clients push messages with player's moves and servers push the updated game state. We have purposely followed the `Tell Don't Ask` methodology, where confirmations are done through callbacks on sent messages instead of more tradional HTTP requests.



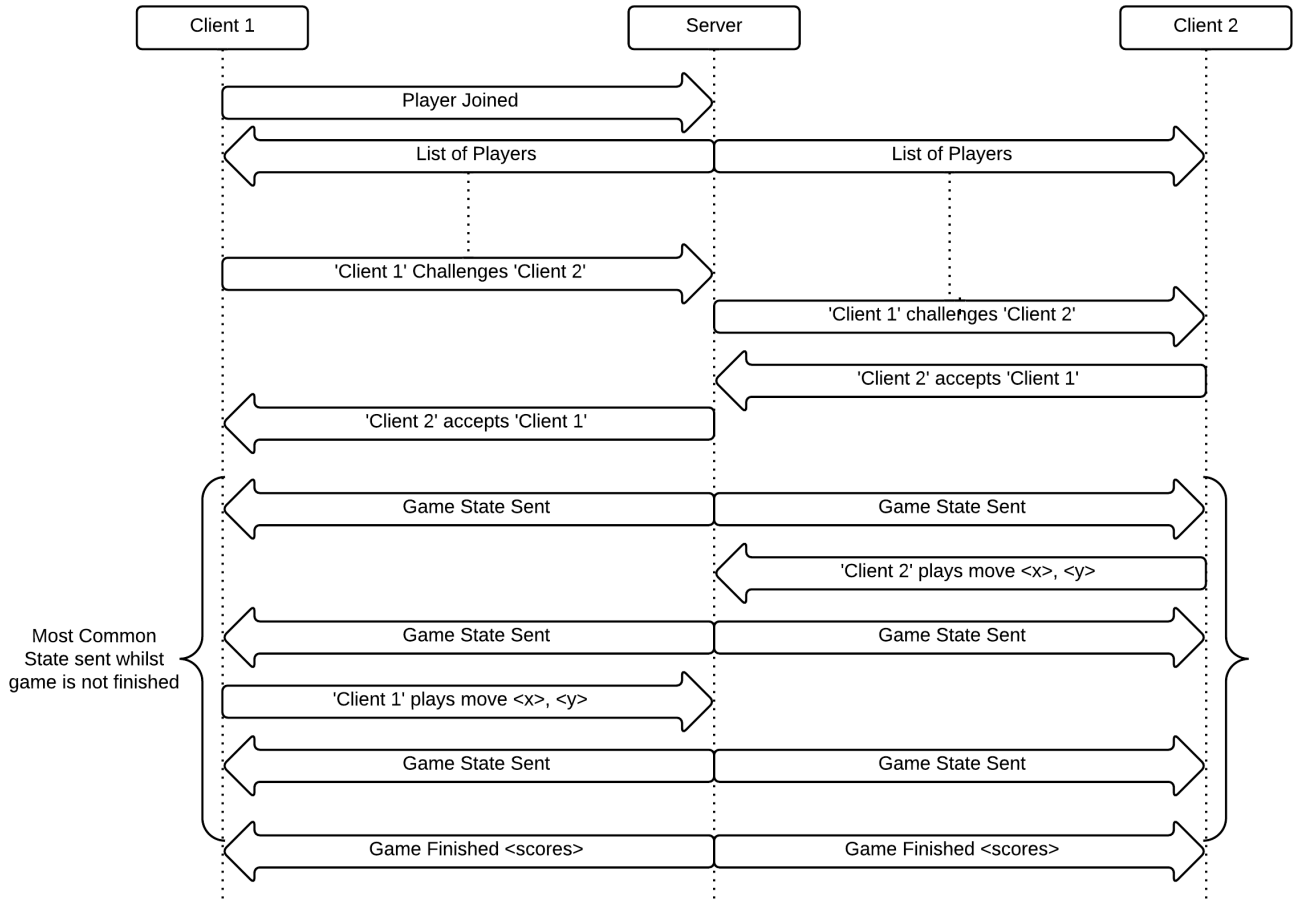

For implementing a complex web application we have utilised many current web technologies:

- We used Coffeescript solemnly to implement both back and front end of our application, as well as the game logic. We were able to leverage the advantages of using Javascript in the browser and in Node.js, and in addition produce more readable code and stick with familiar classical OOP.

- Node.js allows the entire application to be written in Coffeescript (javascript natively) and provides the perfect environment for modern frameworks.
- Express.js allows the web server to be split into more manageable sections: views, routes, static scripting files and external node modules.
- JQuery and JQuery UI Libraries have reduced the workload on producing the lobby, reducing complex HTML operations into a single javascript call.
- socket.io is an interface to web sockets that provides very simple listeners and emitters, written in the same way for both Node.js server and browser client. This is incredibly powerful, and reduces the networking code down immensely such that it is a very simple transformation between design and implementation.
- Paper.js gives us a more powerful interface to the web canvas framework, allowing our Othello GUI code to be written more succintly, with clickable objects being supported natively via the listener callback pattern.

Producing the Web server was done in a slightly differerent way to the way we produced the console game. We first built a proof of concept of the socket.io framework - with a simple box that would send to the server and write a message on the console.

After we had our proof of concept produced, we designed the network schematics for how we wanted the communication between clients and the server to roughly follow. The Connection flowchart is available below.



Then, from this proof of concept we built upwards, iteratively improving until we had a full lobby backend. Alongside this work, we each played around with the various canvas libaries until we settled upon paper.js (we also considered processing.js). After the lobby was done, we used this same iterative improvement process on the GUI until we had our finished product.

# 5 Conclusion

For this coursework, we believe we have produced a simple design, which is easily extensible into differing applications. In designing each component, we attempted to adhere to good `Object Oriented` principles of encapsulation and composition over inheritence.

We have ensured that what we have produced behaves correctly, via utilising the unit testing framework.

We have ensured that our design is clear, and that mostly self-documenting code is accompanied by comments where appropriate.

We have ensured that our design is scalable, via a split between HTTP servers and the websocket backend. The websocket backend is built in such a way that it could be easily scaled via a MapReduce style framework, simply splitting games and sockets between multiple servers.

We have ensured that we have a flexibility in our design, via multiple levels of abstraction away from internal representations of data. This means that our code is easily maintainable and extensible with minimal effort.

# 6 Screenshot