Othello Software Design

Peter Hamilton Sarah Tattersall

November 10, 2011

1 Introduction

1.1 What is Othello?

Othello (also known as Reversi) is a board game involving abstract strategy and played by two players on a board with 8 rows and 8 columns. It has a set of double sided pieces which are black on one side and white on the reverse. The aim of the game for each player is to have the majority of their coloured pieces showing at the end of the game, turning over as many of the opponent's pieces as possible [1]

2 Design Decisions

2.1 Language

For the purpose of this exericse we have decided to write our game using Ruby. One reason for this is that we have been learning Ruby during the timespan of the course and we thought it would be a nice way to fully experience using the language. Another reason for choosing to implement Othello in Ruby is that Ruby is consise, flexible and highly readable as a language and thus leads to quicker development. Although Java has better performance, we felt that this factor was not as important for our game.

2.2 Models & Classes

2.2.1 Game

We decided that the game class should know only about a board and the players. It should contain only the methods that are needed to play the game but none which would calculate the legalaties of the game. As such it will obtain the number of human players desired for the game, create these players, create a new board and loop through obtaining each players moves. The legalaties of the game are left up the the board class. As an extension we decided a board could range between 4 to 20 cells squared. At the begining of a game the user will specify an even number in this range for the size of their board. This allows the user to dynamically alter the overall time-span of the game, with the idea being that a larger board will probably lead to longer and more complex gameplay. We chose 4 as the minimum size because the boards initial pieces need a board of size 2x2, thus 4 is the next even size up from this. We also limited a board to 20 squares so that the board is not too exessive.

2.2.2 Board

A board is comprised of many cells. Instead of making a one-dimensional array we decided it would be simpler to made a two-dimensional array as this makes indexing rows and columns far easier.

We also decided that the board should be where the logic of making a move happens. The board initilizer method has a default value of 8 for it's size as specified in the rules [1] but in order to be able to accommodate for a larger board it can be of any size.

2.2.3 Cell

A cell keeps track of which player occupies it, we decided to do this so that when performing the logic it was clear exactly which player owned a cell. When the flip method is called on a cell a new owner is passed in as an argument.

2.2.4 Player

Player is our super class to Human and AI players. We decided to do this as it maintains the 'is-a' relationship as stated by inheritance. We used inheritance as both Human and AI players will use much of the same code and decided this method was clearer in this case over a composition implementation. [2] As such the Player parent class performs all methods except get move. We decided that the player should each have a get move method rather than the game implement this. This design descision is because in the case of AI there will be no user input. Thus it is neater to keep it encapsulated in the player class. As our game is only a 2D textual representation we decided that the player's colours of black and white could be represented by 'X' and 'O' respectively to make the visulisation as clear as possible. We refer to them as 'Player X' and 'Player O' when displayed so that there is no confusion. A player also has a variable count which keeps track of the how many pieces the player has on a board. When a cell changes ownership it decrements the count of the previous owner and increments the count of the new. We felt this was a good way of working out easily if there is a draw/who has won rather than sweeping over the board and counting the pieces owned by each at the end.

2.2.5 Human Player

We decided to implement the human player 'get_move' method asking for rows and columns separately so that the input is kept as simple as possible. We use the Ruby method 'to_i' which casts the input to an int. If the input is not an int it returns 0. This works well for our design as we ask for cell indexes from the user to be from 1 upwards. Thus when we subtract one to turn it into array indicies 0 is out of bounds and will fail the inbounds error checking. When a user enters a row we decide to check that it is inbounds before allowing them to enter a column so that they instantly know if they have made an error. Once the cell has been chosen we use the board to check if it is a legal move and allow them to play.

2.2.6 AI Player

The AI player has a very basic implementation. It searches through each row and column and finds the first possible move.

2.3 UML Diagram

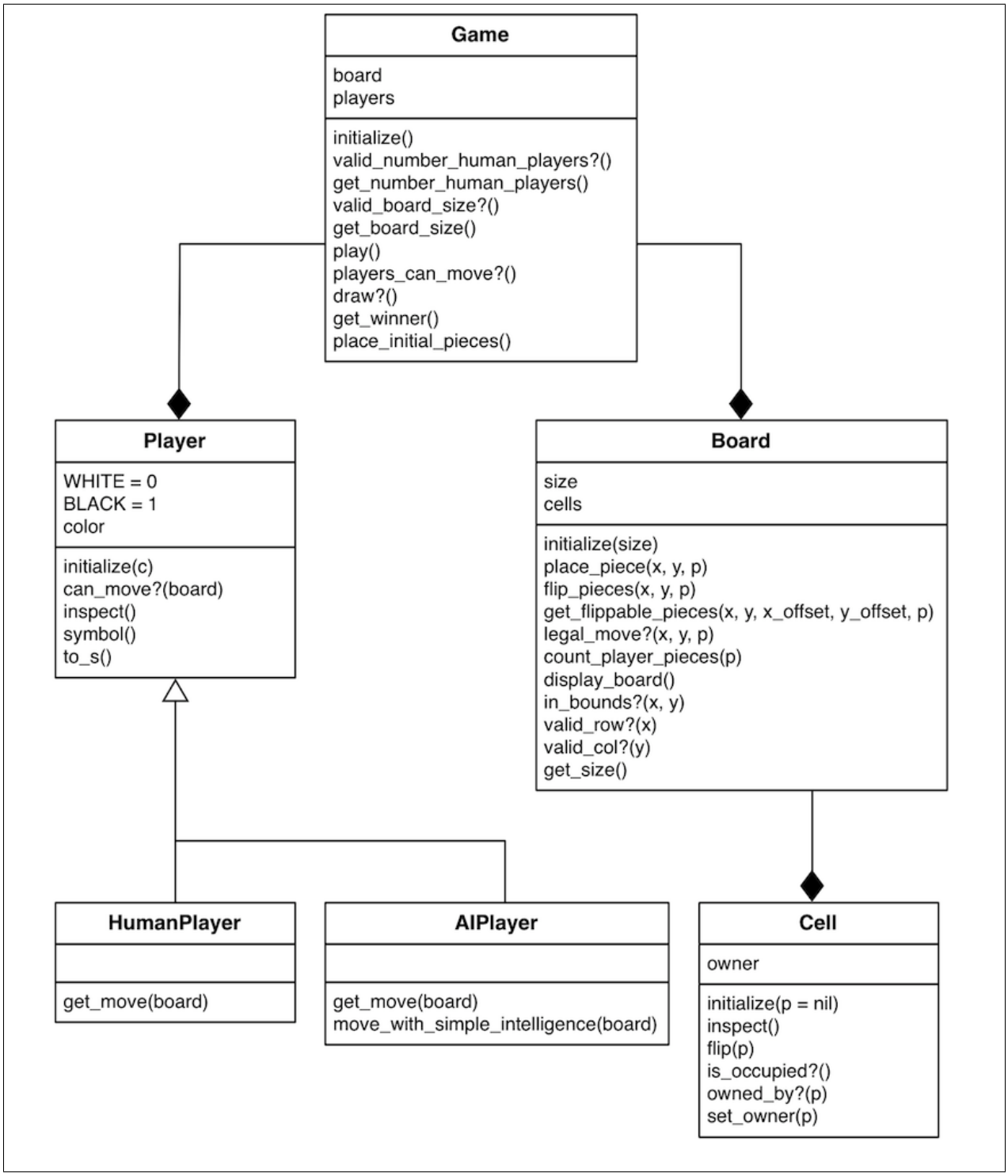

Figure 1: UML Class Diagram of our Othello Implementation

3 Testing with RSpec

In order to test that our game worked to it's specifications we decided to use RSpec to design Othello with a "Test Driven Development" (TDD) approach. We wrote our tests at the same time as developing each section of the program, then wrote the necessary code to make those tests pass. When writing further code, we repeatedly ran the test suite to ensure all the tests which had previously been successful were not made to fail by the new code we were writing.

3.1 What we tested

3.1.1 Game

- In it's newly created state
  - creates an 8x8 board
  - has two players
  - correctly validates the number of human players allowed
  - correctly validates the board size
  - places initial pieces correctly
- In the middle of gameplay
  - correctly identifies whether one or both players can move
- In a winning game state
  - correctly identifies whether one or both players can move
  - correctly identifies the winner
- In an drawing game state
  - correctly identifies whether one or both players can move
  - correctly identifies a draw

3.1.2 Board

- On default creation
  - size should be 8x8
  - size should be the same as the height of the board
  - size should be the same as the width of the board
- On variable board size creation
  - size should be 120x120
  - size should be the same as the height of the board
  - size should be the same as the width of the board
- During gameplay
  - can't place with no pieces to flip
  - flips horizontal pieces correctly
  - flips vertical pieces correctly
  - flips diagonal pieces correctly

3.1.3 Cell

- In all states
  - should initialise with nil owner by default
  - should set the owner correctly if given
  - should return the owner symbol correctly or space if none set
  - should set the owner correctly on flip
  - should correctly identify if it is currently occupied
  - should correctly identify if it is owned by a specific player
  - should be able to set the owner

3.1.4 Player

- On creation
  - should set the color correctly
  - should return 1/0 on inspect corresponding to black/white
  - should return the correct symbol (X - black, O - white)
  - should return symbol if represented as a string
- During gameplay
  - should recognise that they can play a move on the board
- At the end of the game
  - should recognise that they cannot play a move on the board

3.1.5 How it worked

Test Run Results	Code Inspection/Modification

`.....F.....`
 	
`Failures:`
 	
`1) Cell should be able to set the owner`
`Failure/Error: cn.set_owner(p1).owner.should == p1`
`expected: 0`
`got: nil (using ==)`
`# ./spec_cell.rb:52:in 'block (2 levels) in <top (required)>'`
 	
`Finished in 0.2885 seconds`
`33 examples, 1 failure`
 	
`Failed examples:`
 	
`rspec ./spec_cell.rb:51 # Cell should be able to set the owner`
	`class Cell`
	`...`
	`def set_owner(p)`
	`@owwwner = p`
	`return self`
	`end`
	`end`

`.....`
 	
`Finished in 0.64201 seconds`
`33 examples, 0 failures`
	`class Cell`
	`...`
	`def set_owner(p)`
	`@owner = p`
	`return self`
	`end`
	`end`

4 Screenshots

4.1 Inputting a move

```
   1, 2, 3, 4, 5, 6, 7, 8
1 [ ,  ,  ,  ,  ,  ,  ,  ]
2 [ ,  ,  ,  ,  ,  ,  ,  ]
3 [ ,  ,  ,  ,  ,  ,  ,  ]
4 [ ,  ,  , X, O,  ,  ,  ]
5 [ ,  ,  , O, X,  ,  ,  ]
6 [ ,  ,  ,  ,  ,  ,  ,  ]
7 [ ,  ,  ,  ,  ,  ,  ,  ]
8 [ ,  ,  ,  ,  ,  ,  ,  ]
Player X, please enter your move:
row:
3
col:
5
   1, 2, 3, 4, 5, 6, 7, 8
1 [ ,  ,  ,  ,  ,  ,  ,  ]
2 [ ,  ,  ,  ,  ,  ,  ,  ]
3 [ ,  ,  ,  , X,  ,  ,  ]
4 [ ,  ,  , X, X,  ,  ,  ]
5 [ ,  ,  , O, X,  ,  ,  ]
6 [ ,  ,  ,  ,  ,  ,  ,  ]
7 [ ,  ,  ,  ,  ,  ,  ,  ]
8 [ ,  ,  ,  ,  ,  ,  ,  ]
```

4.2 Validating a move

```
   1, 2, 3, 4, 5, 6, 7, 8
1 [ ,  ,  ,  ,  ,  ,  ,  ]
2 [ ,  ,  ,  ,  ,  ,  ,  ]
3 [ ,  ,  ,  ,  ,  ,  ,  ]
4 [ ,  ,  , X, O,  ,  ,  ]
5 [ ,  ,  , O, X,  ,  ,  ]
6 [ ,  ,  ,  ,  ,  ,  ,  ]
7 [ ,  ,  ,  ,  ,  ,  ,  ]
8 [ ,  ,  ,  ,  ,  ,  ,  ]
Player X, please enter your move:
row:
a
Invalid move, please try again
row:
-3
Invalid move, please try again
row:
100
Invalid move, please try again
row:
```

4.3 Identifying a win

```
   1, 2, 3, 4, 5, 6, 7, 8
1 [X, X, X, X, X, X, X, X]
2 [X, X, X, X, X, X, X, X]
3 [X, X, X, X, X, X, X, X]
4 [X, X, X, X, X, X, X,  ]
5 [X, X, X, X, X, X,  ,  ]
6 [X, X, X, X, X, O,  , O]
7 [X, X, X, X, X, X, X,  ]
8 [X, X, X, X, X, X, X, X]
Player X, please enter your move:
row:
6
col:
7
No more moves, Player X won 59-1!
```

4.4 Varying the board size

```
Please enter a board size (4-20) squared:
15
Not a valid board size. Please try again
Please enter a board size (4-20) squared:
16
    1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16
1  [ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]
2  [ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]
3  [ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]
4  [ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]
5  [ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]
6  [ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]
7  [ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]
8  [ ,  ,  ,  ,  ,  ,  , X, O,  ,  ,  ,  ,  ,  ,  ]
9  [ ,  ,  ,  ,  ,  ,  , O, X,  ,  ,  ,  ,  ,  ,  ]
10 [ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]
11 [ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]
12 [ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]
13 [ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]
14 [ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]
15 [ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]
16 [ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]
```

References

[1] Wikipedia: Reversi, October 2011.

[2] Stack overflow: Composition vs inheritance, October 2011.