C220 Software Engineering: Design Coursework

Harshwardhan Ostwal Joseph Crowe

We have written an implementation of the board game Othello using the OCaml[1] programming language. OCaml is a dialect of ML[2] that has been enriched with object-oriented and imperative features, making it a powerful language with many uses. OCaml has a strong static type system, allowing much of a program's expectations to be expressed with types, and this sense of correctness is verified by OCaml's compiler.
For information regarding how to run this OCaml code, please refer to 'Running' section at the end of this report.

Module Structure

The concerns of our program are separated into a number of encapsulated modules, each in a file of the same name, so that these components can be independently written, understood and tested. Most modules also house a single class of the same name. The modules are summarised in the list below. In the following sections, we will go into more detail about how the components interact.

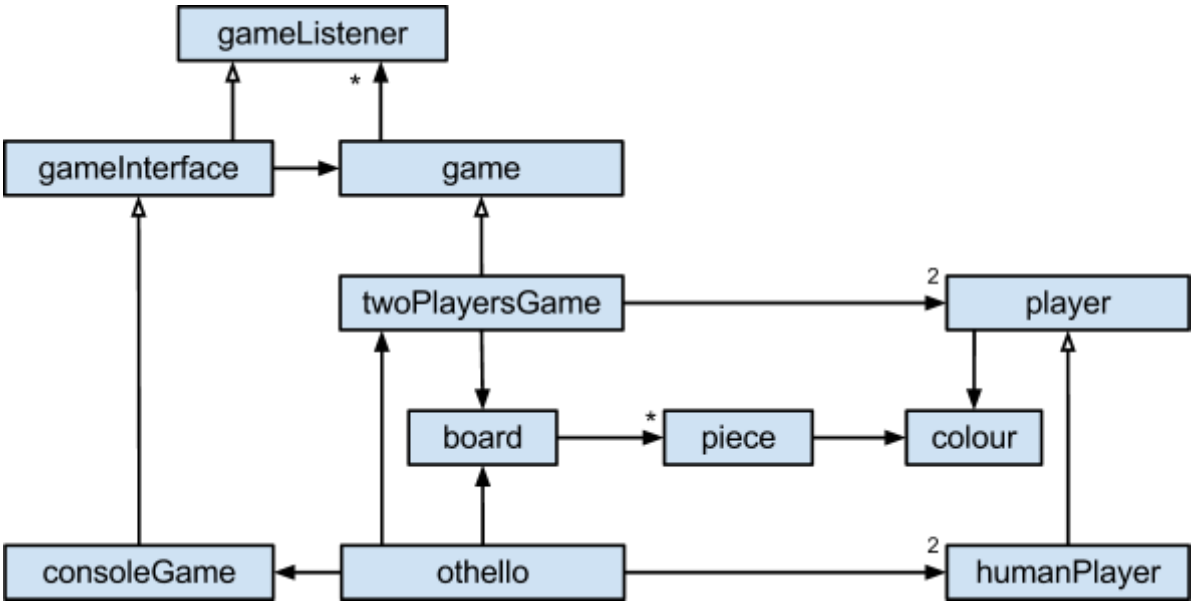
- **Colour** Enumerates the possible colours of an Othello piece
- **Board** Holds the state of an Othello board
- **Game** Specifies an abstract model of an ongoing game of Othello
- **GameInterface** Specifies an abstract presentation of a game
- **Player** Specifies an abstract control interface to a Game
- **ConsoleTest** A trivial implementation of Game, for isolated testing
- **TwoPlayersGame** Our primary implementation of Game
- **ConsoleGame** Our primary implementation of GameInterface
- **HumanPlayer** Our primary implementation of Player
- **Othello** Provides the main executable

Class Structure

Below is a UML-inspired diagram showing the relationships between the classes (some of which are actually OCaml *records* and *variants*) in our program. Filled arrowheads denote aggregation, and empty arrowheads denote inheritance. More details about the purpose of each class can be read from the comments preceding the class declarations in the source code.

[1] http://ocaml.org

[2] http://en.wikipedia.org/wiki/ML_%28programming_language%29



Model-View-Control Architecture

The Game, GameInterface and Player components form the respective Model, View and Control of an instance of the **MVC** design pattern.

A Game manages the business logic and central processing of the game's state. We implement this as a TwoPlayersGame, which alternately takes moves from two opposing players, and enforces a standard set of rules.

A GameInterface responds to changes in the Game's state, presenting some representation of the state. It also acts as a wrapper for Games, delegating the call to `start` so that code interacting with GameInterface need not know the underlying details of Game. We implement this as a ConsoleGame, which prints to standard output a representation of the board using ASCII characters, along with English messages describing other events.

A Player manages the delivery of input from some agent participating in the game. We implement this as a HumanPlayer, which reads moves in a standard human-readable format (the specification of this format resides in the Board module, because it is shared with the ConsoleGame module) from standard input, printing prompts and error messages also to standard output.

This was done to make the design flexible: each of the concrete implementations we provide could readily be replaced with alternative implementations, without any changes required in the other components. For example:

- We might implement an alternative ruleset with, for example, two pairs of players on each side, who take turns to move, by writing a new subclass of game.

- The game might be presented through a graphical user interface instead of the console, by writing a new subclass of gameInterface.
- An AI-controlled computer player might be introduced, through a new subclass of player.
- A remote opponent whose moves are received over some network protocol could similarly be implemented as a subclass of player.

Game Event System

In the relationship between game and gameListeners instances (of which the relationship between Game and GameInterface is a specialisation), we have demonstrated the **Observer** pattern. It is possible for any object implementing the requirements of a gameListener to be registered to receive game events from an instance of game. This leaves our design open to modifications involving an arbitrary number of listeners, and makes it simple to introduce new game events as required.

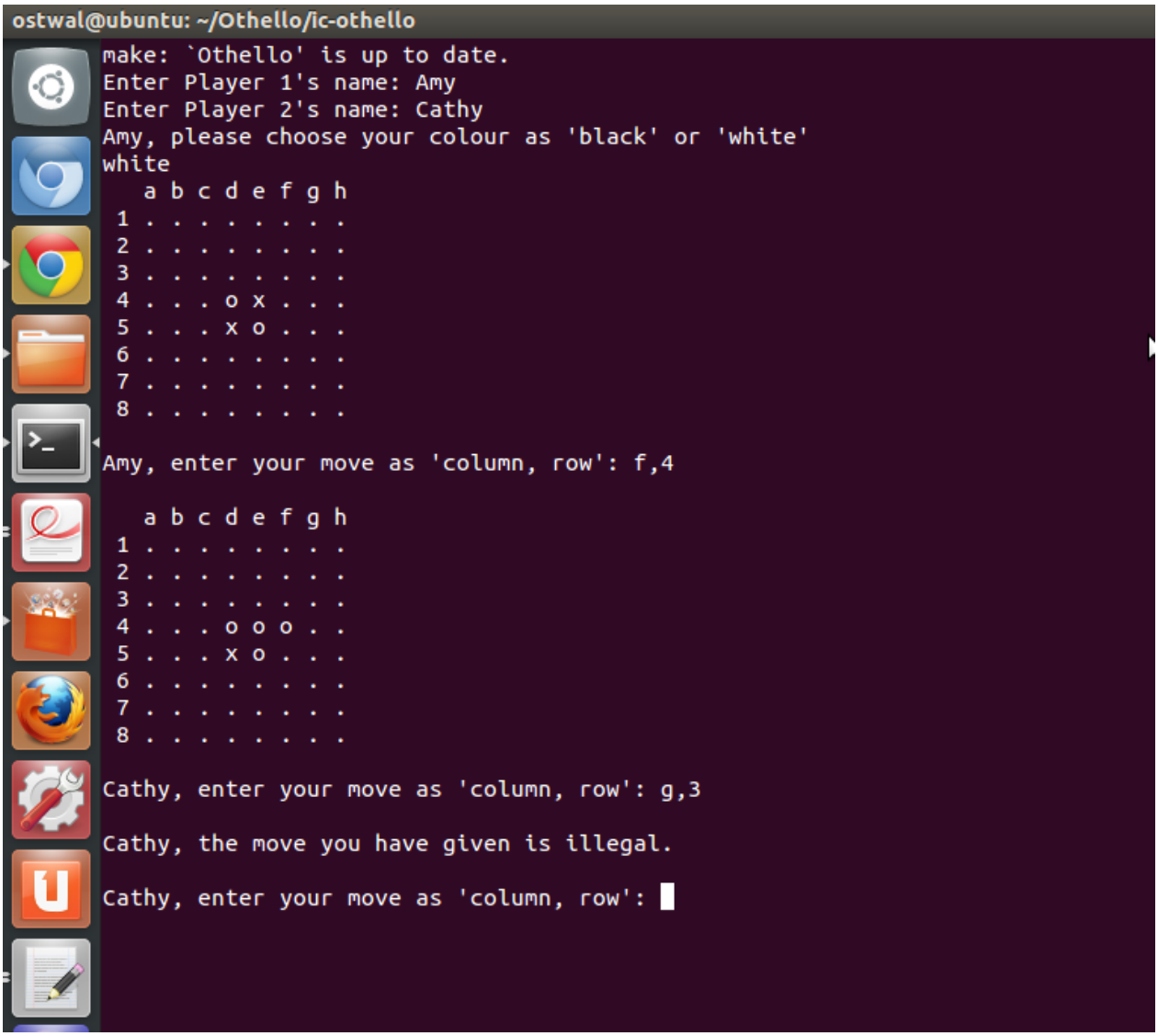
The game events are listed and described in the comments in the Game module.

Front-End Components

The Othello module contains the class othello, which encapsulates our three concrete system components in a single working unit. It thus acts as a wrapper for the comparatively complex underlying system, and could conceivably be integrated into a larger system without difficulty.

Description

The present Othello is a simple game to be played between two human players on a single terminal. At the start, the game-board gets shown as a square matrix of dots '.' of the size given inside the program. 4 central dots are replaced with 'x's and 'o's representing two black and two white pieces respectively. Players input their moves (e.g. d,5) through the keyboard following the game instructions. An invalid move prompts the player to re-enter their move. And if the current player doesn't have any option available on the board, the game automatically skips them and asks the next player to play another time. When both the players consecutively don't have any possible option for playing, the game terminates and gives out the result.



Building and Running the Game

We include a `Makefile` that can be used with GNU `make` to compiled (using `ocamlc`) a binary of our implemented front-end for the OCaml runtime system. This could easily be adapted to build a native binary using `ocamlopt`, but the interpreted version suffices for testing and demonstration.

- To build the executable, run `make` (or `gmake`, on certain systems, e.g. Windows).
- To run the program, run `./Othello`.