Group Project

TCP2201
(Object Oriented Analysis and Design)

Myrmidon Chess Project

Trimester 1, 2018/2019

Group 7

Prepared by

<Team Leader>

Tey Wei Long, 1161101116, +60111-5017885, dannyongtey@gmail.com

<Team Members>

Yeo Yong Yaw, 1151105408, +6012-9236700, yongyawyeo98@gmail.com

Previnash Wong Sze Chuan, 1161101470, +6010-2866326, prevwong@gmail.com

Table of Contents

Introduction

The Myrmidon Chess is a chess game played on a 6 x 7 board, in which it differs from other existing chess game in the sense that the chess changes their forms every three turns. This project aims to utilize the Java programming language to implement the chess logic and create a user interface with Java's built in UI libraries.

Objectives

The end goal is to create a bug-free and user-friendly Myrmidon Chess game that two players can play on. By working on this project, we also aim to accomplish the following objectives:

- Learning to collaborate on a coding project through proper communication and collaboration tools like Github;
- Incorporating design patterns in our project, thereby deepening our understanding;
- Following proper coding conventions and standards to create efficient program code;

Design Patterns and Object Oriented Concepts

Our project mainly follows the MVC architectural pattern and the Singleton design pattern, briefly described below:

Model-View-Controller

An architectural pattern traditionally used for GUIs, now becomes widely used in the world of web development as well, with popular frameworks like Ruby on Rails, Laravel etc. available. In our design, the MVC is defined as follows:

- View - The GUI, ie. the chessboard and the menus seen by the user in which the user can interact with.
- Controller - The functions that accept inputs from the user, thereby manipulating the model and/or updating the view.
- Model - The attributes not directly seen by the user, and is manipulated through the controller.

Singleton

We also implement the singleton design pattern. This design pattern ensures that there can only be one and only one instance for a particular class. The chessboard is implemented as a singleton since only one chessboard is needed in a gameplay. The implementation details are documented in the following sections.

Compiling and Usage Instructions

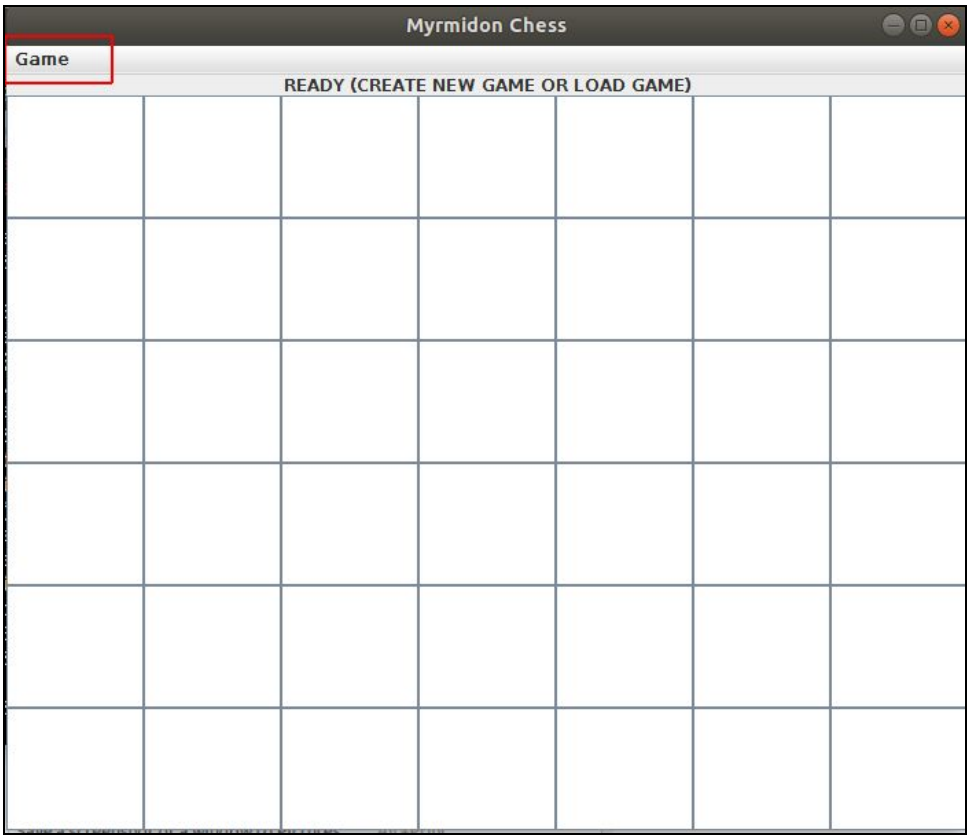
Command Line Instructions

1. Compile all files. Command: *javac *.java*
2. Run the Runner file. Command: *java Runner*

Usage Instructions

1. Following the compiling instructions above, the program should now run, and the GUI should appear, showing the chessboard.
2. From thereon, there are two ways to start game:
    a. Start a new game. *Menu > New Game*
    b. Load a saved game. *Menu > Load Game*
3. During the gameplay, should you wish to save the game, go to *Menu > Save Game.* The program will remind you if you attempt to quit without saving.

The highlighted area is the Game Menu:



Features Explanation

In the following section, we briefly describe how the major features are implemented.

Game Play

In order to create a chess game, we will need objects that represents the board and the chess, hence the board and chess class.

The Board Class

The board class is a singleton class that contains most of the chess logic, including keeping track of the rounds, the player's turn, the UI elements, the attributes and methods that are relevant in running the game.

As a singleton, the board class has initializes its own object and provides a function getBoard( ) that returns the static board instance.

Private Constructor

The Board constructor is declared private to avoid external classes from calling the constructor, thereby creating their own board instances and break the singleton pattern. Meanwhile, no getter and setter methods are exposed. In other words, the class methods can only be invoked in the class itself, fulfilling the singleton pattern. Other than that, the constructor initializes its private variables as usual.

The Chess Class

The chess class is the barebone for any chess instance, containing attributes like chess type and player as well as methods that determines the valid moves of the chess. It has normal getter and setter methods to enforce proper encapsulation.

During the start of the game, the chess objects are initialized in the Board class, and stored in the board class as a HashMap, with it's key as the position, and the value as the chess object.

The Position Class

The position class is used more like a struct rather than an actual class, as its main purpose is to represent the XY coordinate of the chess board. However, in order to compare the equality of two positions, we have to overwrite the **equals( )** and **hashCode( )** methods, because Java treats two objects as different even if the underlying attributes have the exact same value.

The hash code of the position object is therefore, in this particular project, stored in the format of 'XY'. For instance, for the coordinates (4,3), it is stored as 43.

The SaveObject Class

The SaveObject class is instantiated every time a game is to be saved or loaded. The purpose of this class is to avoid saving unnecessary attributes (like UI elements) on the chessboard in the save game. The object only contains the following four information:

1. The HashMap containing the coordinates and the chess;
2. The current turn count;
3. The current player;
4. The current game status;

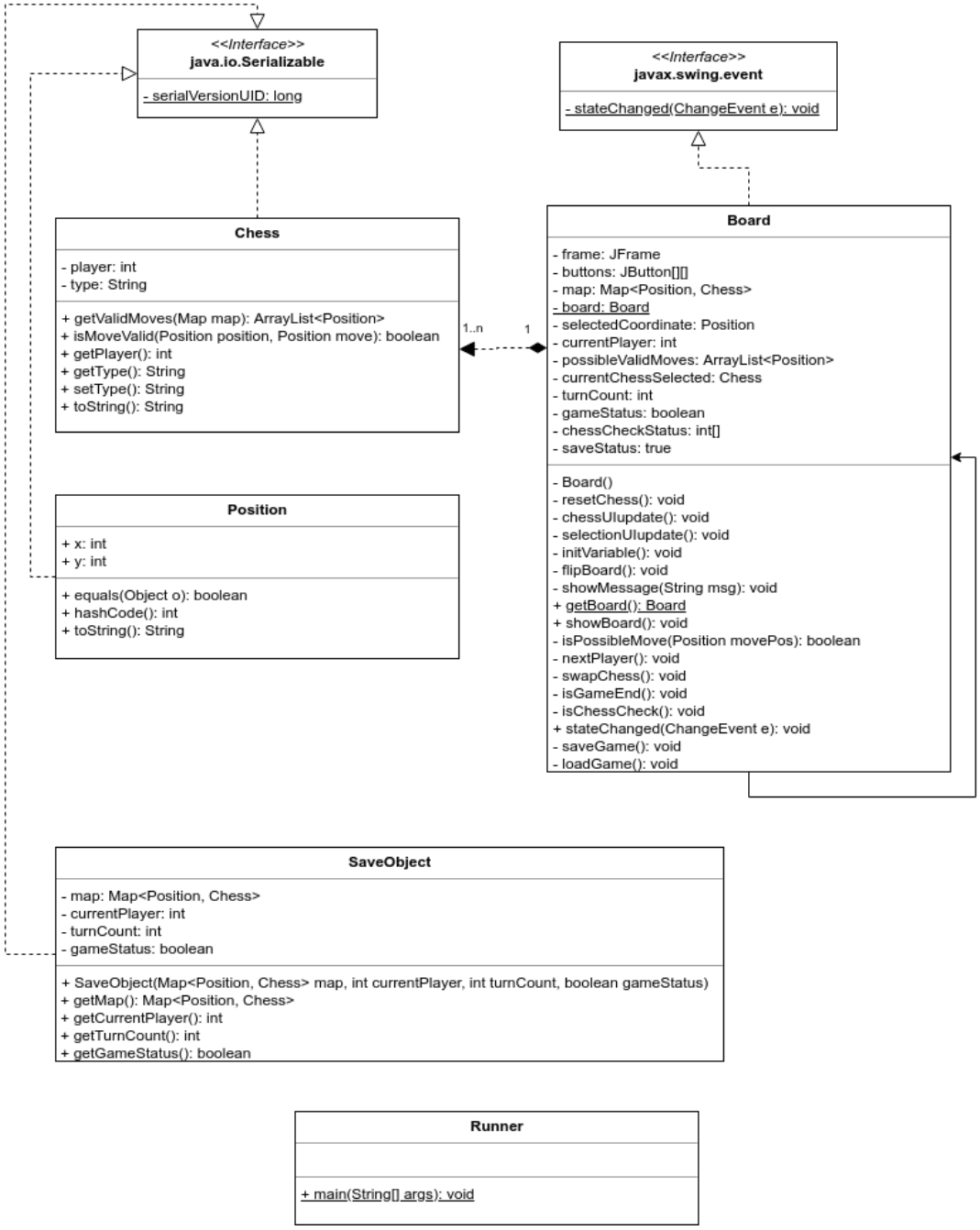
Serializable Interface

The serializable interface is used to enable the saving of SaveObject instances. It makes saving an object and reading it seamless and quality-loseless. The format of the file also prevents tampering from other users, as it is stored in Java's own binary serialization format.

Check Detection

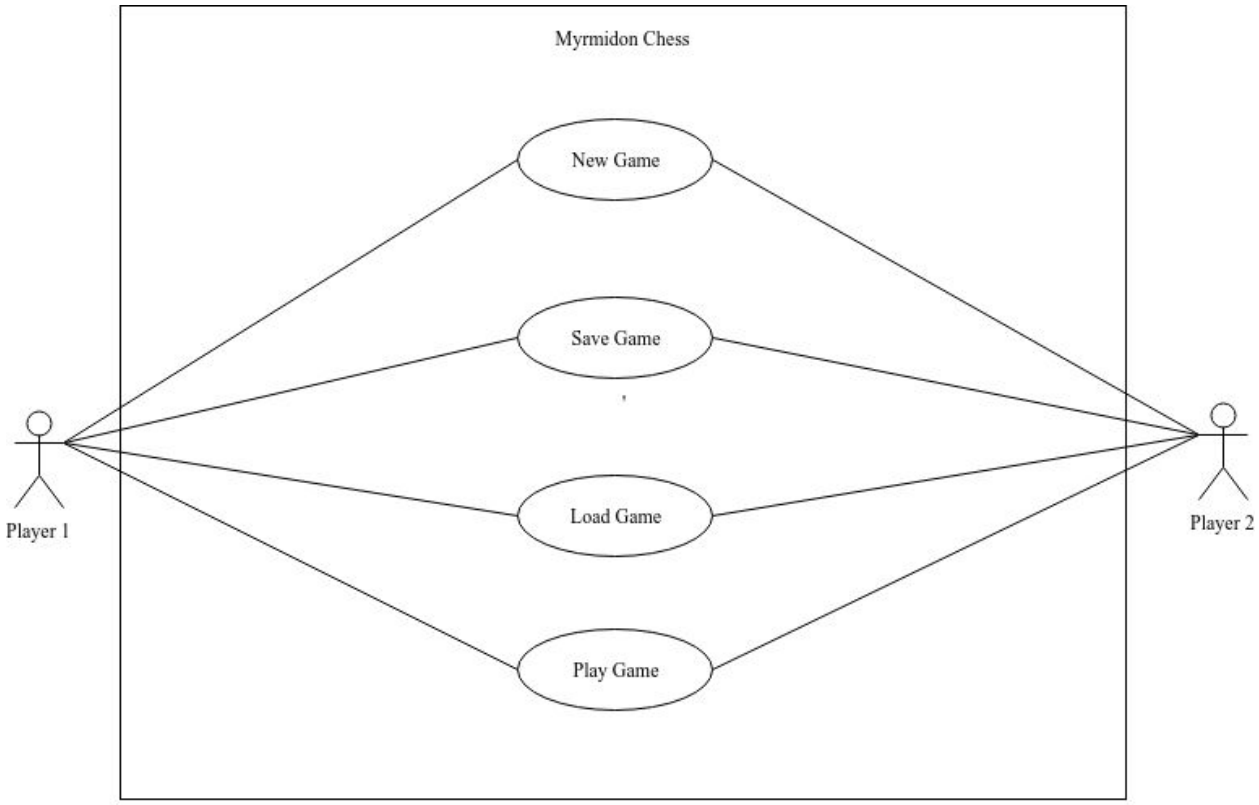
The check detection algorithm is an additional feature we add to the program. It calls the getValidMoves( ) after the chess has moved to see if the next possible moves of the chess overlaps with the Sun's position. If yes, a check message is issued.

UML Class Diagram



The class diagram above dictates the use of Singleton pattern in Board class. Only a single Board instance is created (inside and by the class itself) and is returned by the static method getBoard(). The role is to ensure only one board is active and in use at a time.

Use Case Diagram

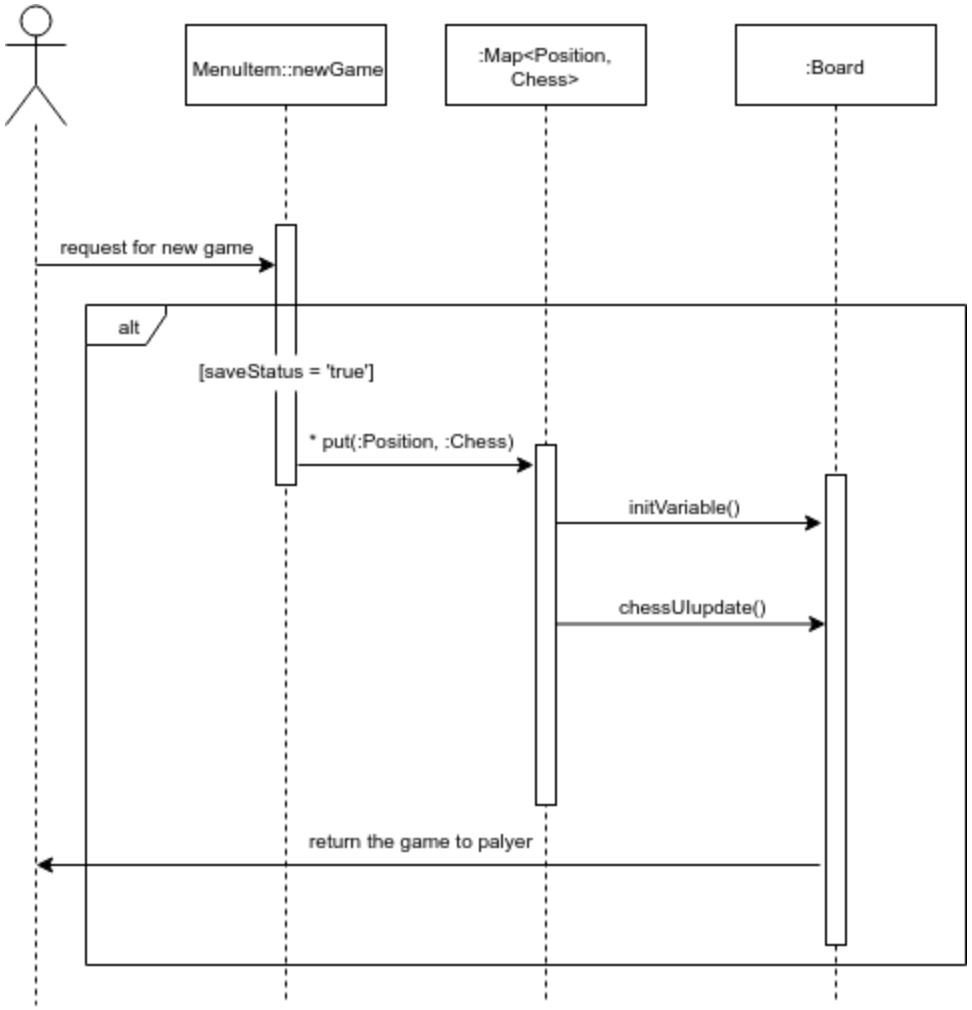


The use case diagram indicates that there are four main functions that the both players can execute (there's no host or admin feature in the game). They are:

1. Creating a brand new game
2. Saving an existing game
3. Loading a saved game
4. Playing the Myrmidon Chess game

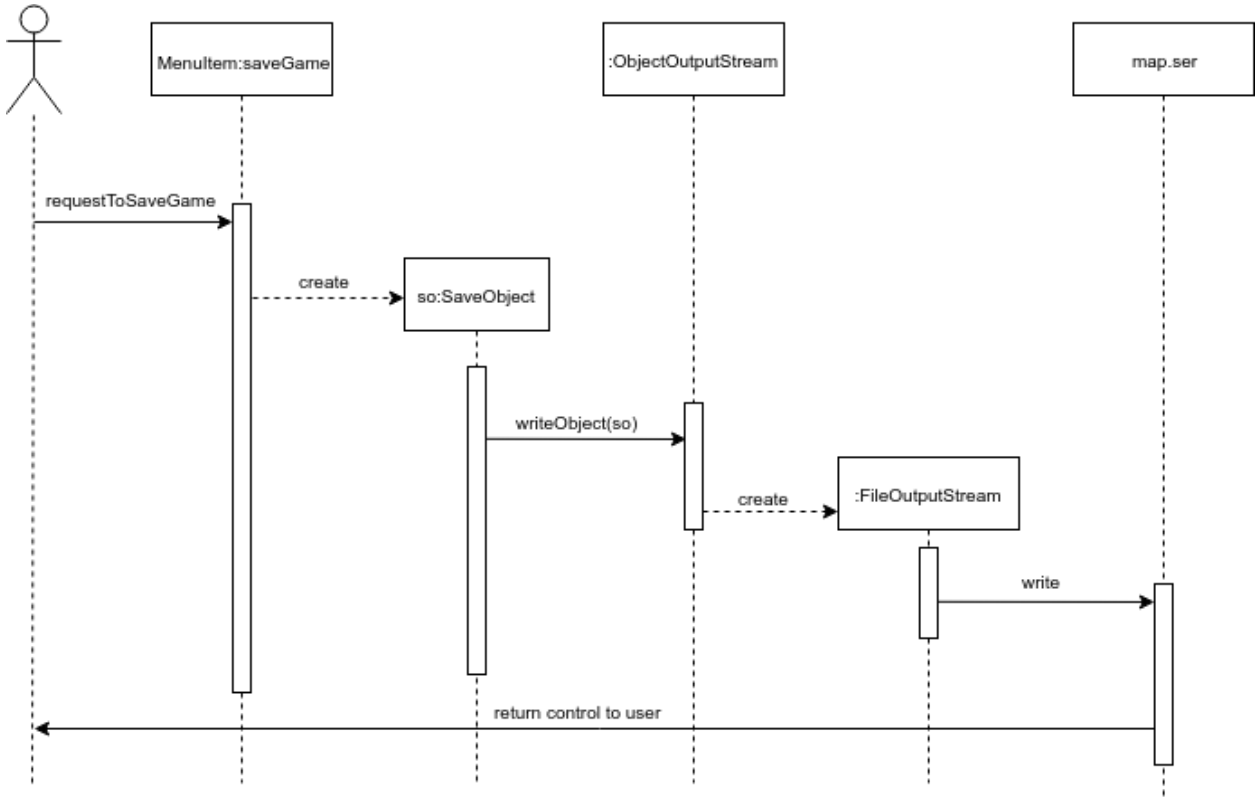
Sequence Diagram

New Game



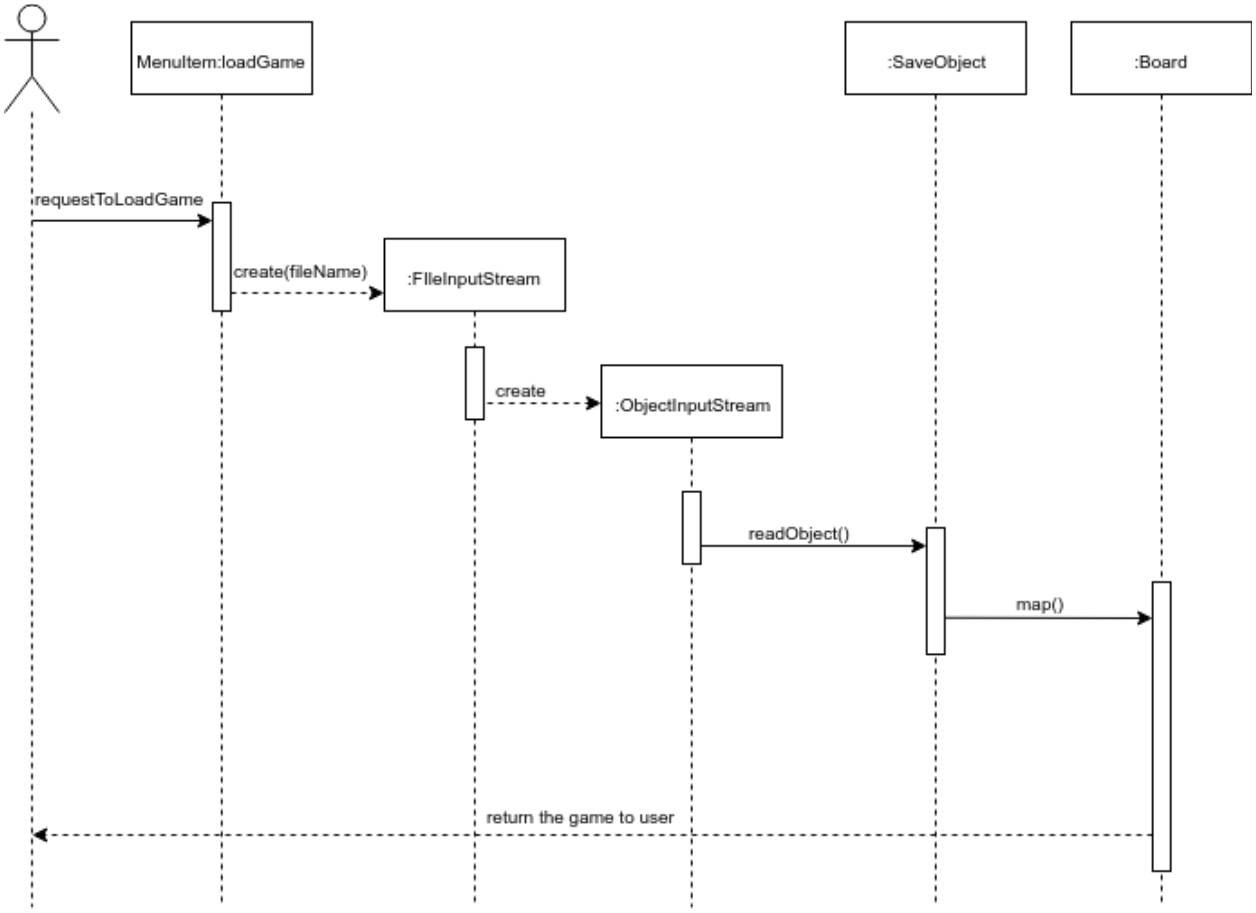
In a new game, when a player requests for a new game from the Menu Item, the program calls multiple times the put method on the chess map (hashmap storing all chess and their coordinates), to reset all their positions to default positions. The initVariable( ) method is invoked to reset all board variables like player turn, turn count, etc. The UI update function is then called to update the board UI. Afterwards control returns to the user.

Save Game



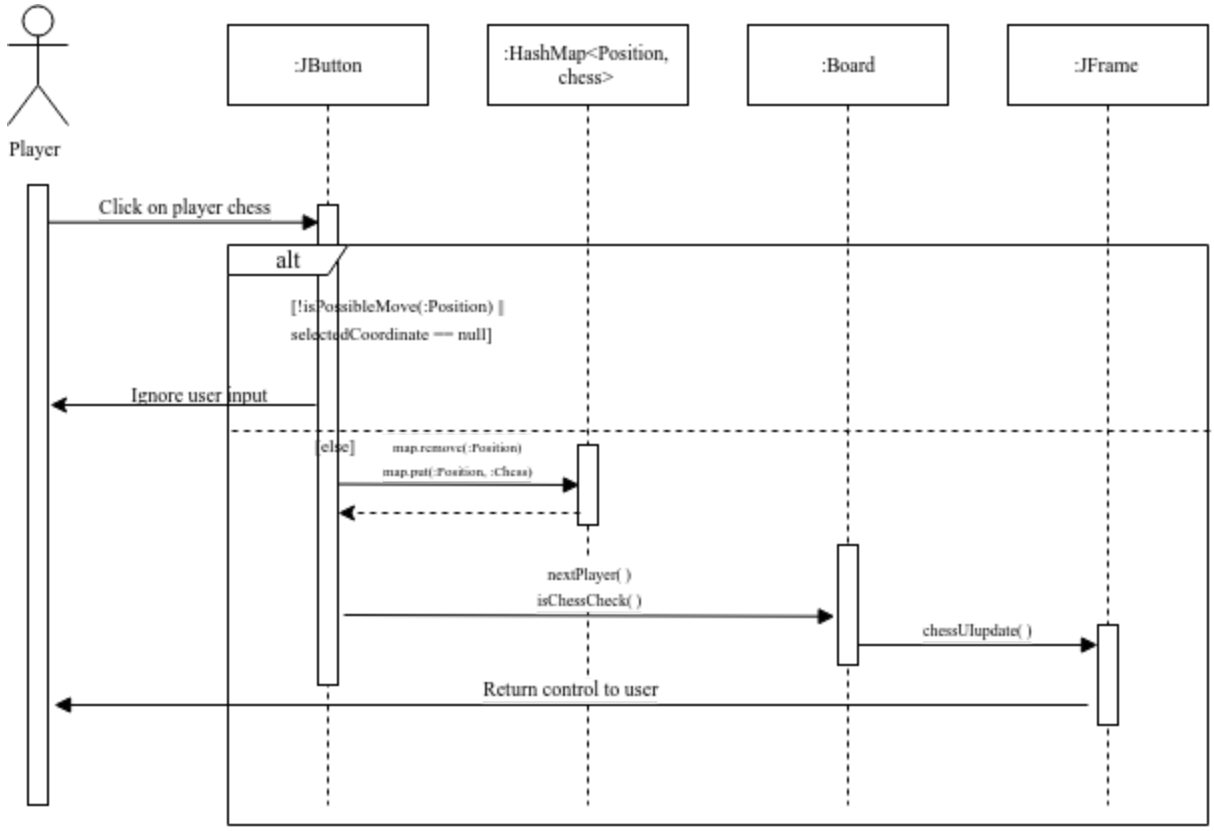
In the event of a save game, when a player requests for a save from the Menu Item, the program creates a save object containing only the attributes needed for a successful game save. Afterwards, it writes the save object to external file, map.ser. Afterwards the control returns to the user.

Load Game



In the event of a load game, when a player requests for a load from the Menu Item, the program creates a new FileIO object to read the serialized file. The file is then deserialized through the readObject( ) call and then the attributes are mapped to the Board class. Afterwards the control returns to the user.

Play Game



When a user wants to play the game, he first selects a chess. On the state change of the button (when the button is clicked), the program determines if the move is a valid move. If it is not, the program ignores the user input and returns the control to the user immediately. If the move is valid, the original coordinate of the chess is removed from the hashmap and replaced with the newly selected coordinate. Then, the nextPlayer( ) and isChessCheck( ) methods are invoked accordingly to toggle next player's turn as well as to check if the other player has been checked. The results are then updated in the Board's attributes and chessUIupdate( ) is called to update the buttons on the JFrame. Finally, control returns to the user.

Conclusion

A seemingly simple program to implement turns out to be a challenge to follow the proper conventions and come up with efficient code. It is in fact, simple to implement design patterns in the code, but making it work as intended takes time, and optimizing the code performance takes even more time.

We have learnt several lessons, summarized below:

1. Spend quality time planning for the program flow and logic will save us from headache later on. This includes creating flowcharts and pseudocode.
2. The time spent on implementing the actual code should account for around 20% of the time. Most time should be spent on researching, planning and testing for bugs. Using too much time to code will shorten our time to debug the program.

There are always ways to create a better program that runs more efficiently, that takes less lines of codes, and have less issues. Comparing to our previous work in Object Oriented Programing and Data Structure, we found significant improvements in the way we approach this assignment. True enough, this program definitely has room for improvements, and years later we might look back and laugh at how we wrote the code.

But as of now, it's important for us to realize that we have pushed our limits and this is the best we have produced.