COMP3211 Course Project

API Design Document

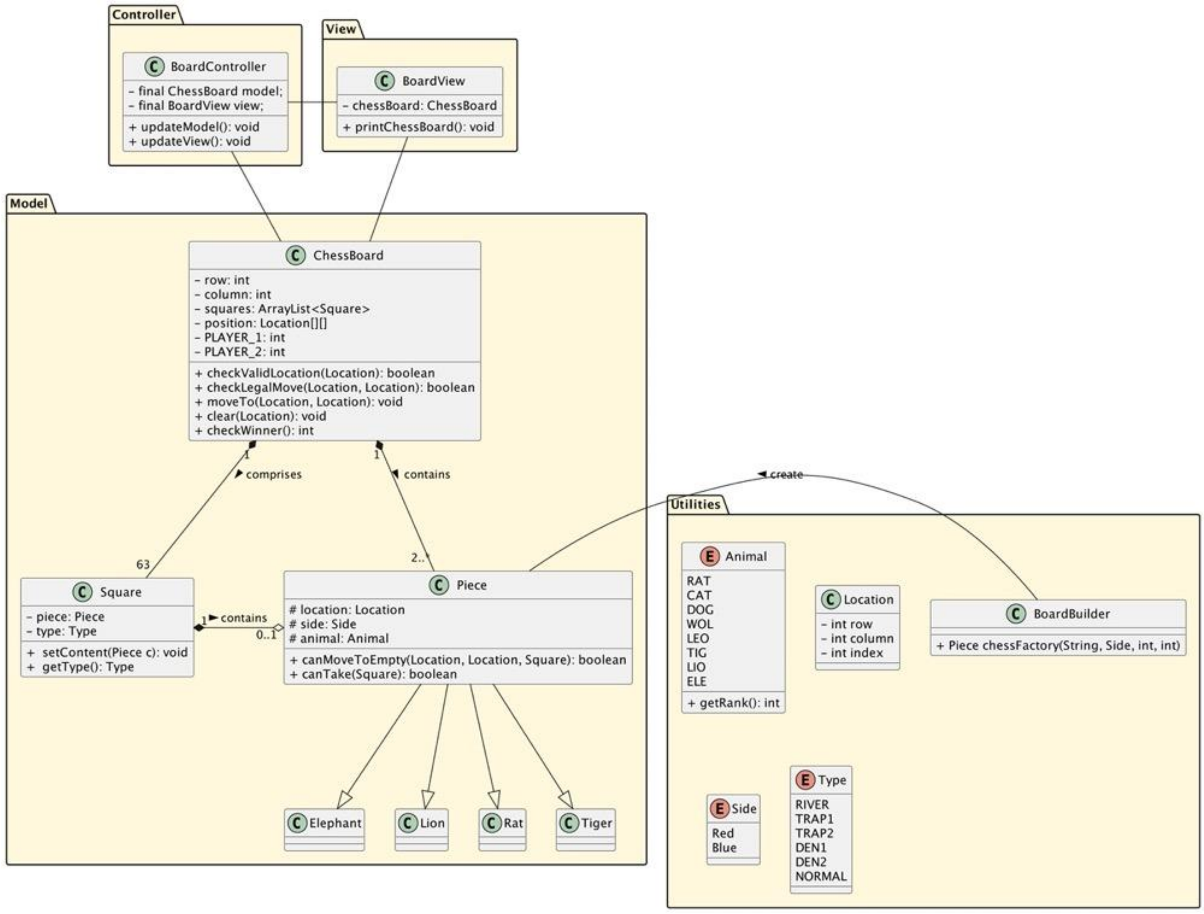
Group 30:

YE Jianyang	*20083865D*
HE Zhejun	*20084054D*
ZHANG Mingkun	*20099019D*
GAO Zhilin	*20104379D*

1. Code Components

See the attached “api” folder.

2. Class Diagram

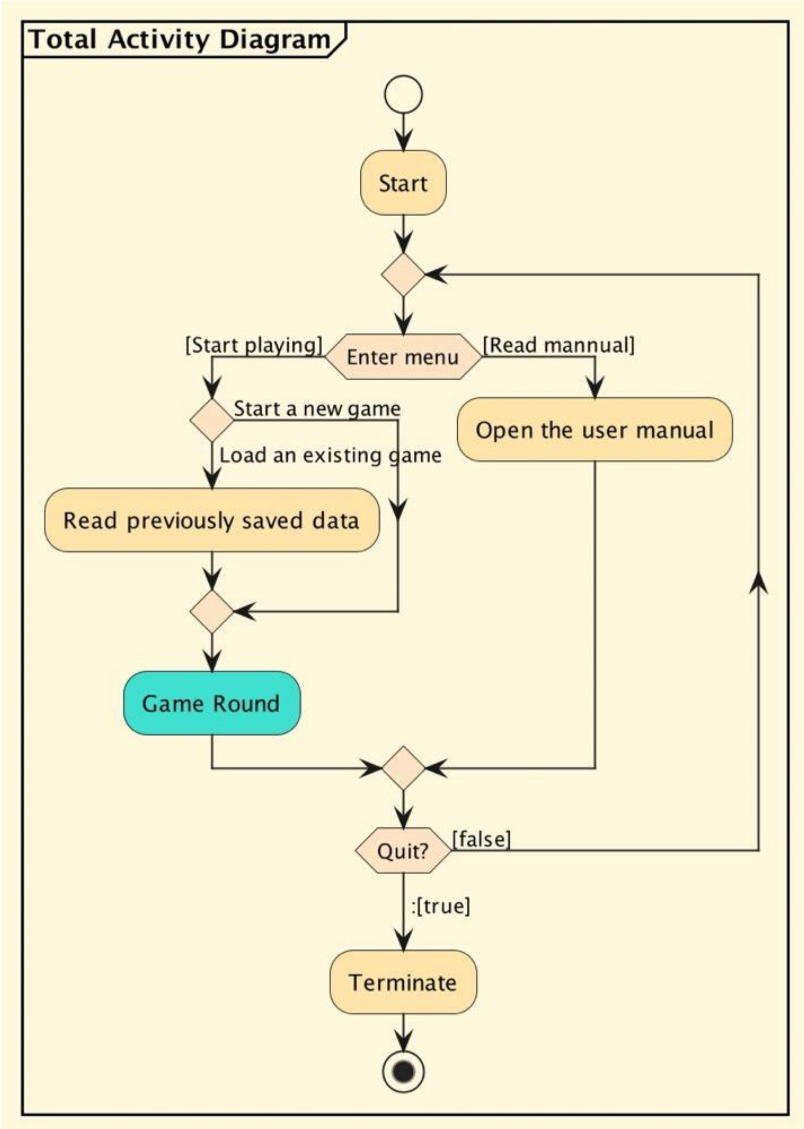


3. Activity Diagram

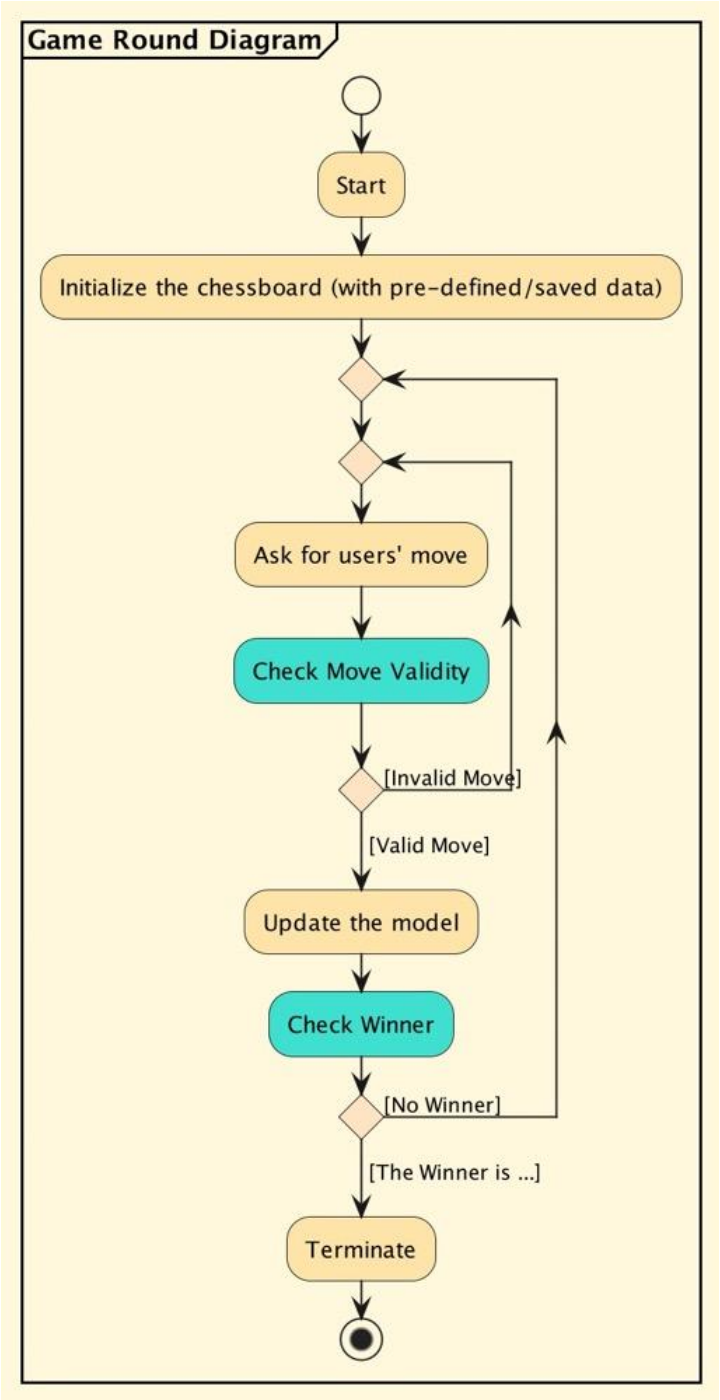
In this part, diagrams describing the total user activity are presented. Since some parts of the user activity are complicated, they are broken down into four diagrams: Total Activity Diagram, Game Round Diagram, Check Move Validity Diagram and Check Winner Diagram.

If the diagrams are not clear enough to check, please see the attached “diagrams” folder.

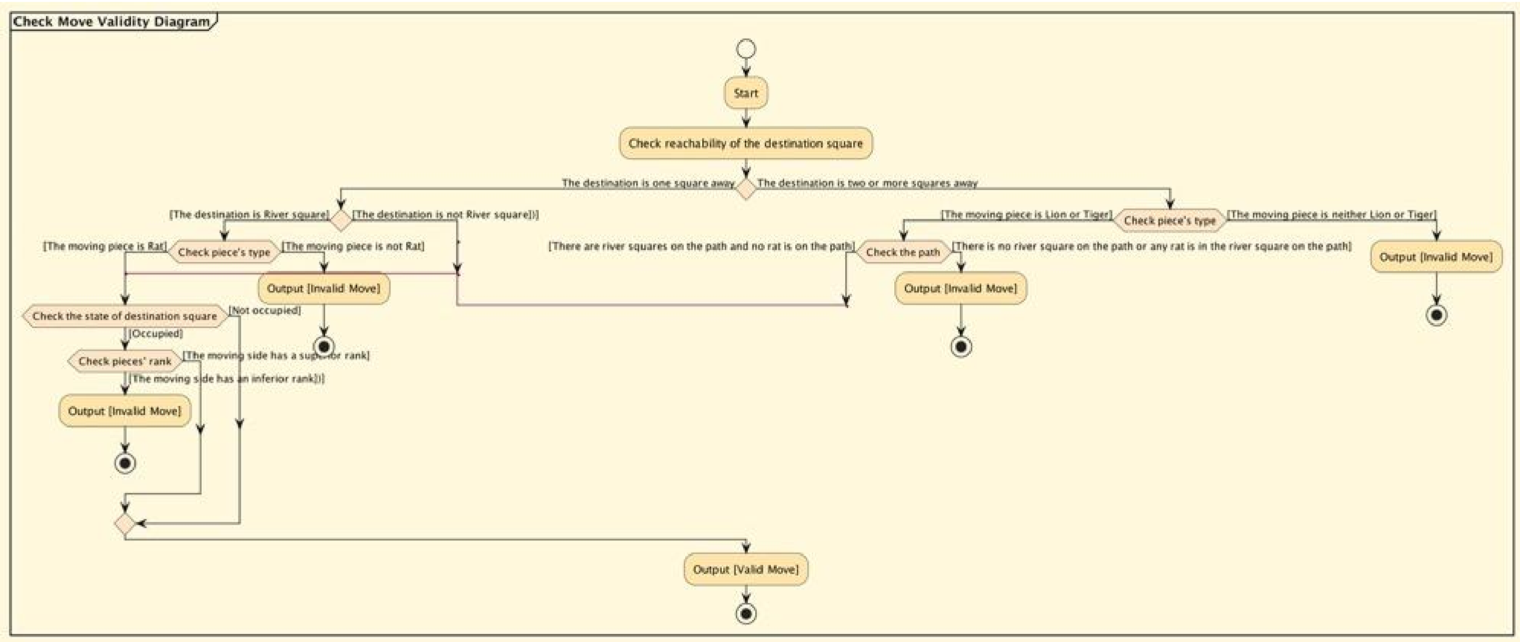
Total Activity Diagram



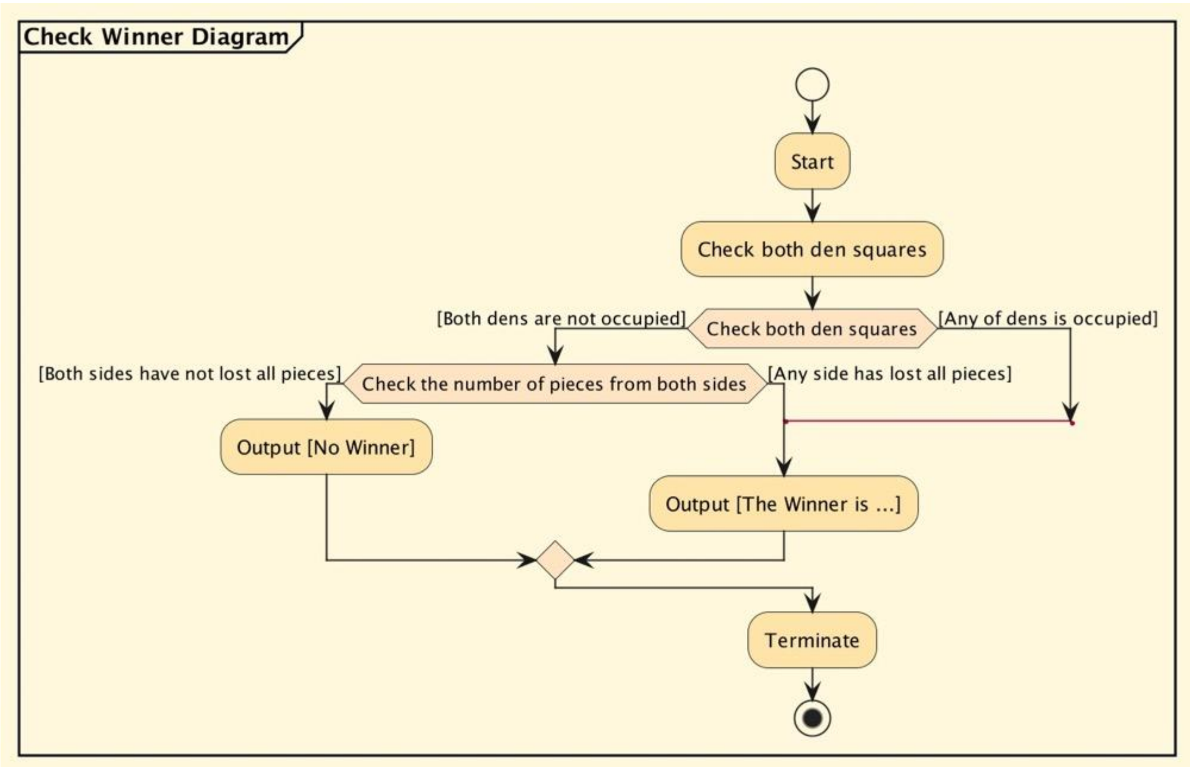
Game Round Diagram



Check Move Validity Diagram



Check Winner Diagram



4. Important Design Decisions

I. Structural Design

When designing class structure, we are following Object-oriented Programming (OOP) Design Principles & Design Patterns, and we will be explaining how our design observes them:

(1) Composite Reuse Principle
(2) Demeter Principle
(3) Liskov Substitution Principle
(4) Open Closed Principle
(5) Single Responsibility Principle
(6) Factory Pattern (Design Pattern)

Classes can be easily designed to model real-world objects. Since the behaviors of pieces are not simple, the class `Piece` is designed to represent a most simple piece, acknowledging that some pieces have special features. It contains methods that describe low level activities and used in all pieces, for example, determining whether it can take another piece given the path is valid.

We follow the **Liskov Substitution Principle** here. We let `Rat`, `Elephant`, `Tiger` and `Lion` extend `Piece` since they are subtypes of the general concept of a piece. In this way, it is convenient to implement some special rules (such as rats can enter the river) simply by overriding methods in `Piece`, while retaining other features that are in common. Note that there are no such special rules for cats, dogs, wolves and leopards, so we can represent them by class `Piece` directly, rather than specially define new classes.

We implement **Factory Pattern** for instantiating `Piece`. To construct pieces from their string representations, we design a static method called `chessFactory()` in the `BoardBuilder` class. What we are trying to do is to remove the instantiation of the actual implementation class from the client code, i.e., in the `ChessBoard` class.

In the beginning, we thought it suffices to use an `int` to represent a square. However, later we found that this causes problems. For example, the square has a attribute type (enum), while the piece does not have. It needs to be stored in `Square`, not `Piece`. Meanwhile, when a square is printed, the result not only depends on the type, but is also affected by the piece on it. Therefore, we create a class `Square` that stores the type and a `Piece` content as well. We also override `toString()` for `Square` so that it is printed in a comprehensive manner.

Although the chessboard is simply an aggregation of squares, we think it should be a class too, as sometimes high-level rules are not verifiable by a single `Square` or `Piece`. As an example, a tiger cannot know by itself whether there is a rat in the river before it jumps.

Hence, the `ChessBoard` class shall represent the chessboard and be responsible for providing interface for player-level functions of the whole model.

In addition, we define a `Location` class to better facilitate the conversion between row-column **coordinates** (x,y) and the **index** in the squares `ArrayList`, which are interchangeable descriptions of a location. This avoids the chaos of methods sometimes using row and column as parameters or return values, sometimes using indexes.

In a nutshell, our overall class design is observing **Composite Reuse Principle. Single Responsibility Principle and Demeter Principle**. Every class, function in our program have responsibility over a single, specific part of that program's functionality. As for **Demeter Principle** and **Composite Reuse Principle**, We are also make every object have least knowledge of other components, and using aggregation to connect them, which lowers the coupling of classes.

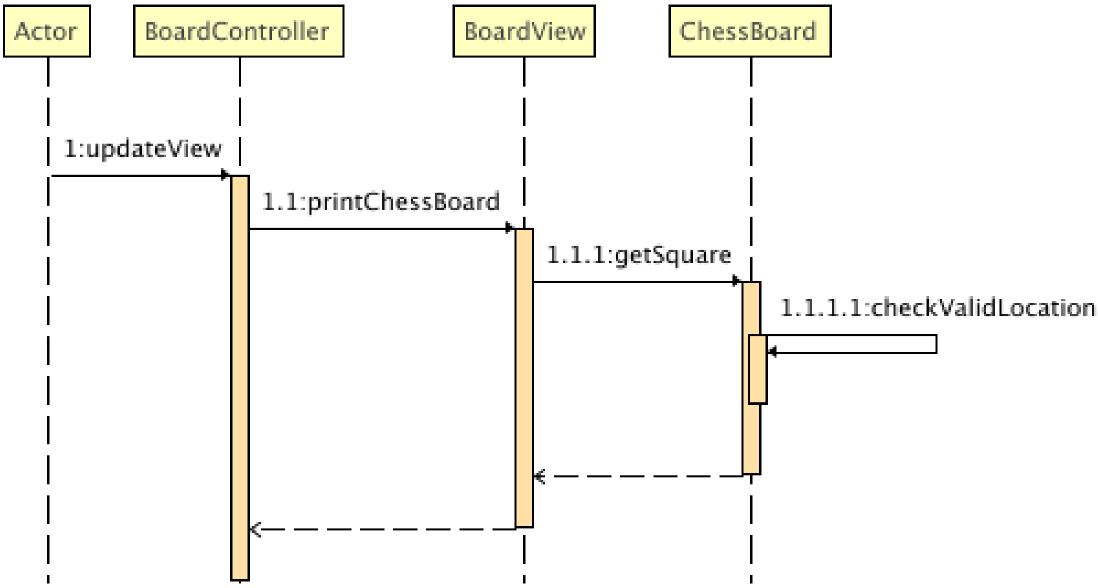
II. Functional Design

We consider that the most complicated part of the game is to check whether the move specified by the player is valid, as it involves a large number of factors. For a move to be legal, first, the target location should be reachable from the origin. That is, it has to be adjacent to the origin (except when jumping across the river) and it cannot belong to a type that is not open for entry. Second, if there is a piece on the location, the moving piece must be capable of capturing it. This means the target should be an enemy piece either having an inferior rank or being trapped. Third, a rat in the river cannot attack or be attacked unless by the enemy rat which is also in the river. Fourth, if a tiger or lion is about to jump over the river, it would fail when a rat is in the water.

We divide these conditions into two groups: the first two are low level features that largely depend on the moving piece itself and can be verified by the piece independently, whereas the other two are high level features that only the chessboard knows. Then we design `canMoveToEmpty()` and `canTake()` in `Piece` and let the chessboard determine `canMove()` in general.

5. Appendix

Sequence diagrams generated by a plugin of IntelliJ IDEA. See the “appendix” folder for more details.



BoardController.UpdateView()