Revisions to the API design:

According to our latest design and TA feedback on Submission2, we have made some revisions to the API design document:

1. A clear Sequence diagram with a brief logical explanation.
2. A clear Activity diagram with concrete details. After our revision, it will be easy to map the logic in the diagram to implementation.
3. Add more detailed function parameters, return type and explanation (mark as yellow) to each function stated in this document.

COMP3211

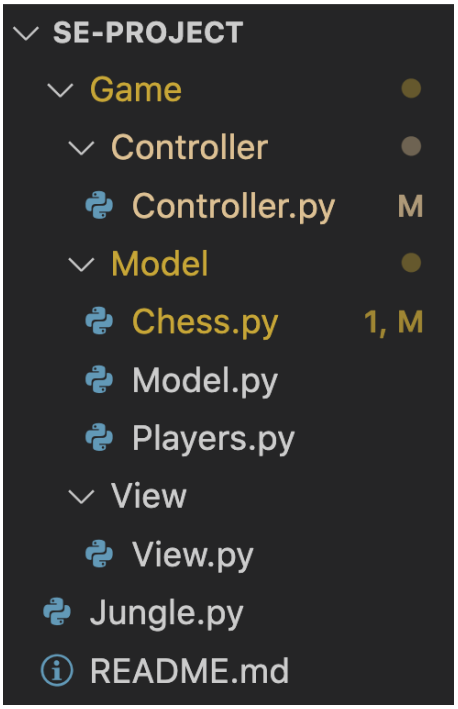
Software Engineering

Project Submission2

API Design Document

Members:

1 File Structure



Use python Jungle.py to run the game.

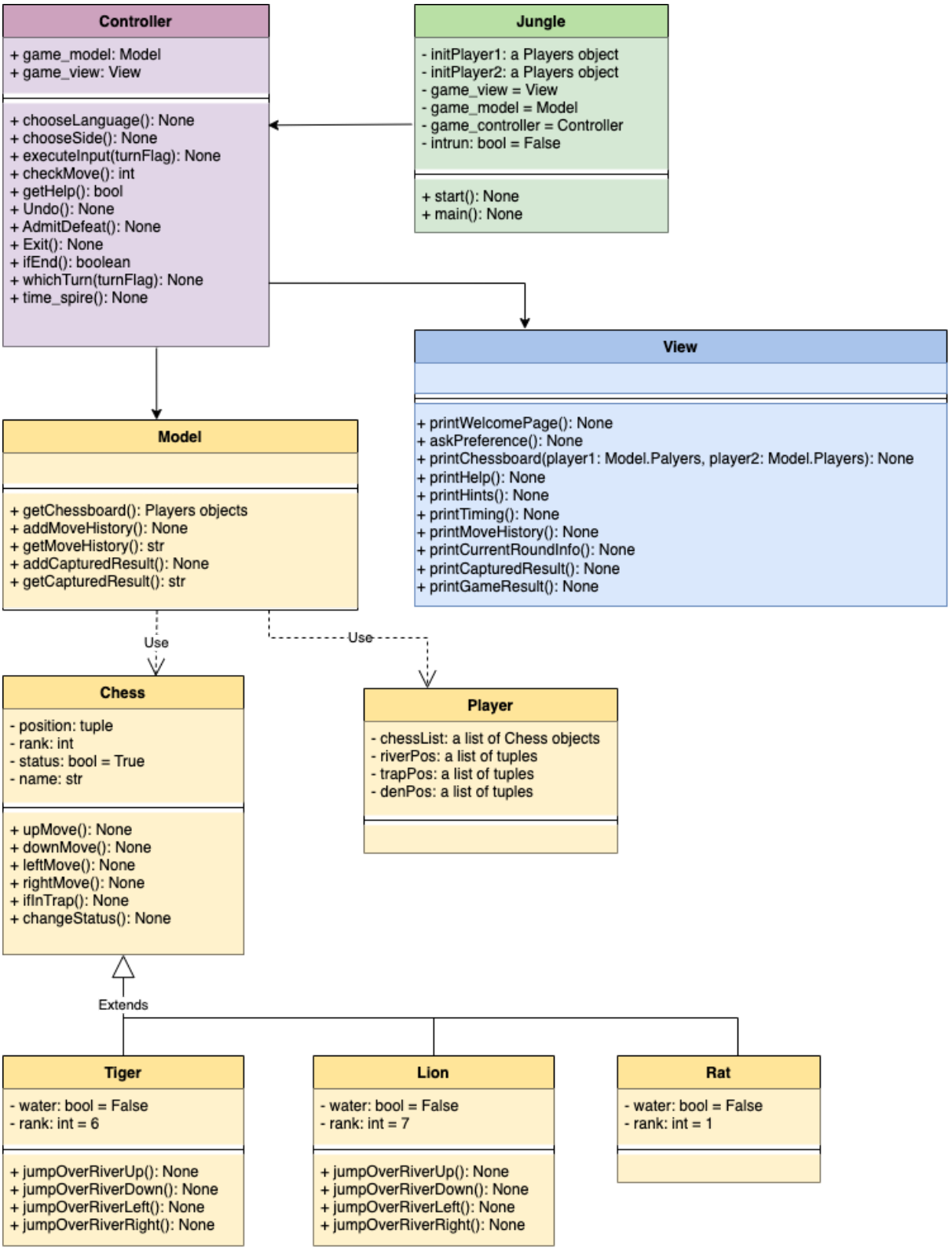
2 Design Diagram

The Jungle Chess Game is designed in the patterns of **Unified Modelling Language(UML)**, **Sequence design**, and **Activities design**.

2.1 UML diagram

In our design, the Jungle game's function is divided into 4 main classes: Jungle, Model, View, and Controller. The main function is in Jungle class. The Jungle class is the class where we can directly start the Jungle Chess game. The Controller class manages user interaction and passes these interactions to the View and the Model, while the View class manages how the system will be displayed to users and the Model class manages the system data and associated operations on that data. The Chess class can represent each chess object, except Lion, Tiger, and Rat. These 3 animals have different movements since they can go into/ jump over the river. Therefore, 3 other subclasses corresponding to Lion, Tiger, and Rat are extended to the Chess class. The Players

class represents a player's chessboard situation, including animal cheeses, river area, traps area, and den area.



2.2 Sequence diagram

The sequence diagram will lay out the internal running relation among classes and files. A transparent and exhaustive presentation of the whole system in the below figure majorly includes three stages:

- **Initiation stage:**

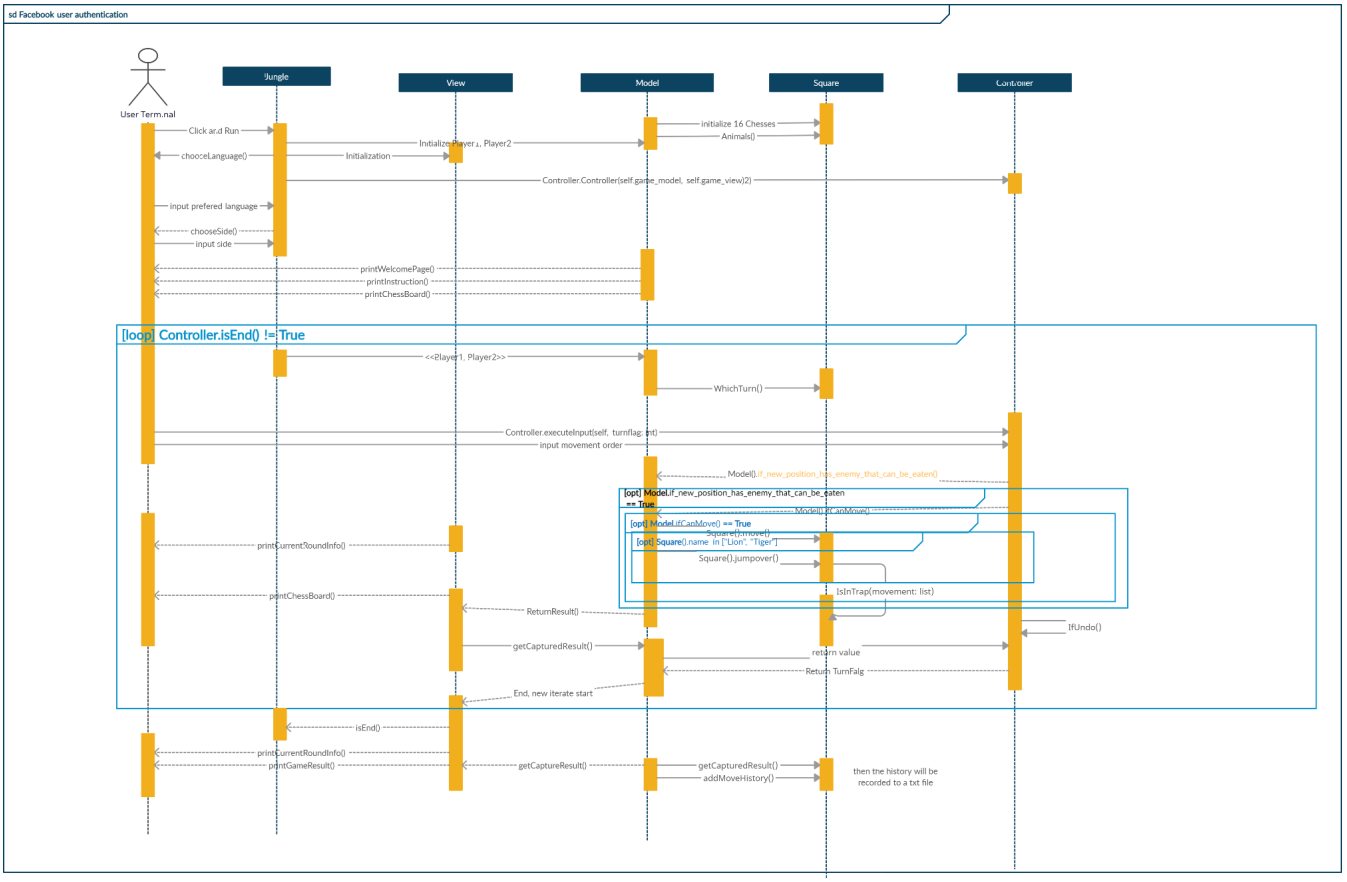
At this time, the user will frequently interact with the program, especially the initialization class, Junge.py. Chess and user will be initiated and stored in cached for preloading. This stage contains view.View() and model.Model(). The main contents in this section are inner initialization and deploying the necessary info to buffer before the user starts inputting the order.

- **In-game stage:**

At this stage, interaction is mainly among MVC structural codes. It contains Controller.excuteInput() and View.presentChessBoard(). The main contents in this phase are the inner computing program situation.

- **End-game stage:**

In this phase, the system will list a sequence of game information and store the options in history. Mainly function contain View.printMoveHistory. The system will compute the performance of each user and record the historical choices and orders. The program will compute more instead of I/O operations to keep the result properly and intuitively represent the users. Lastly, the system will write the historical record to a specific path for the user to review later.



For the sequence diagram, now the updated system mainly focuses on the interaction between Square.py and model.py. After getting input from the command line, the controller only responded to passing variables into the model. The model will invoke the function ifCanMove() first to check the validation of movement. After that, the system will start to judge whether the next position stands for the enemy or the same side of chess. So below is a brief example to help you understand our programming process better.

Moving Chess: a Lion on the upside, we assume its next step has an enemy elephant. After the order passes from the controller, the lion is going to move down one step from (3, 5) -> (3, 4)

self.ifCanMove(Lion, nextPos)

self.if_move_out_of_range(nextPos) will return False

self.if_next_step_in_land(nextPos) will return True

self.if_position_has_same_side_animal(Lion, nextpos) will return False

self.if_position_has_higher_rank_enemy(Lion, nextPos) will return Ture

This step at this time is illegal. So it will warn the user: “you can not type in such kind of order”

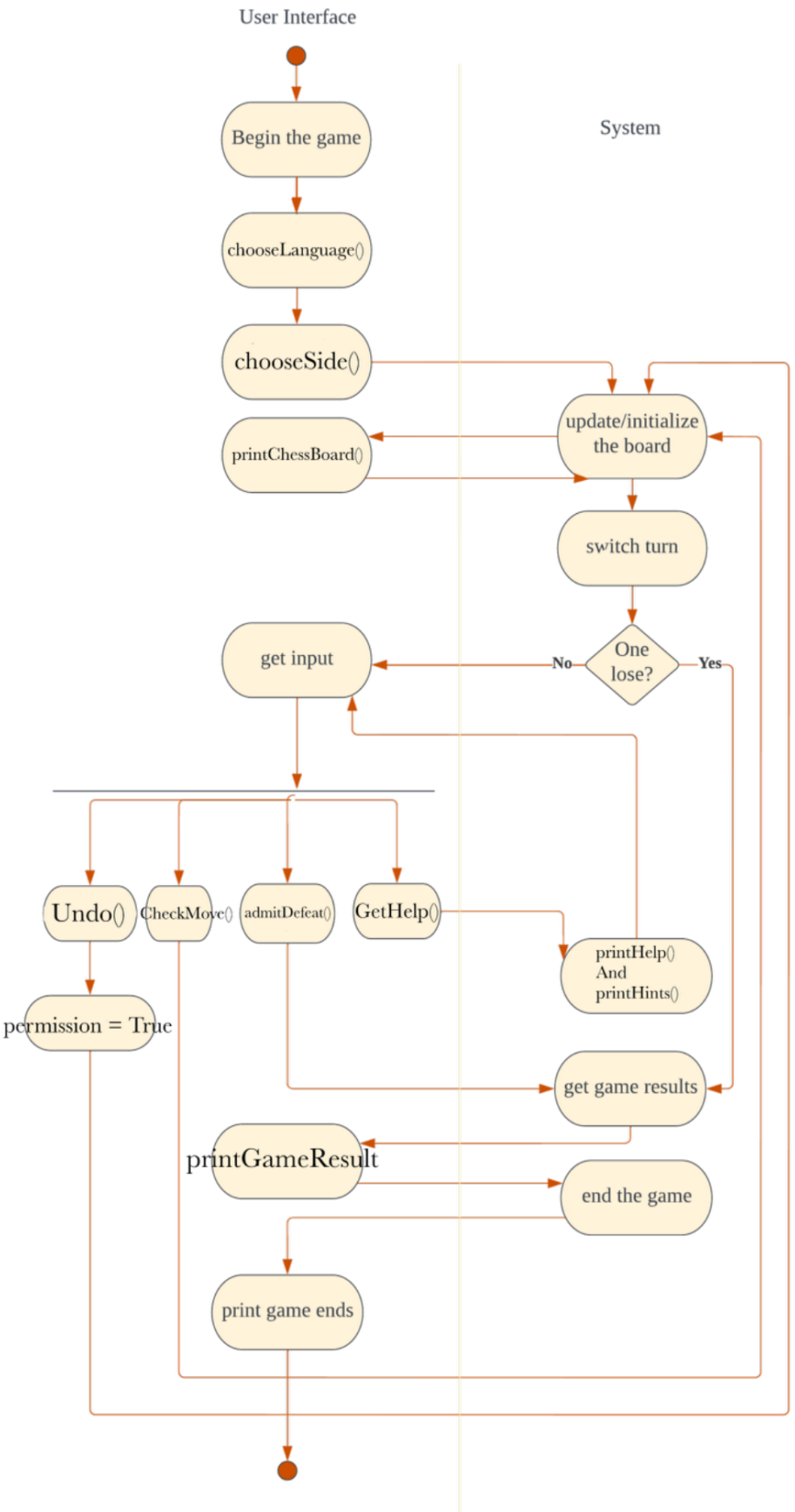
2.3 Activity diagram

An activity diagram is a graphical representation of a workflow of step-by-step activities and actions, supporting selection, iteration, and concurrency, UML activity diagrams help reveal the flow of the system being developed.

Drawn here is the activity diagram of the entire game containing:

- actions(ellipses)
- decision(diamond)
- bars(split and join)
- two circles(start and end)

revealing the control logic of a round of the game, as well as the relationship and interaction between the user interface and the system.

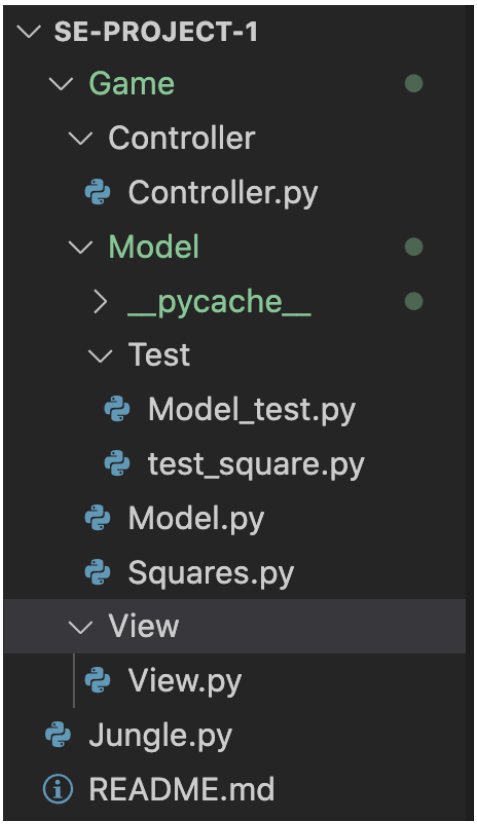

User Interface
System
Begin the game
chooseLanguage()
chooseSide()
printChessBoard()
update/initialize the board
switch turn
One lose?
No
Yes
get input
Undo()
CheckMove()
admitDefeat()
GetHelp()
printHelp() And printHints()
permission = True
get game results
printGameResult
end the game
print game ends

In a round of games, the core control unit is the decision part to determine whether the game has been won by one player or not according to the current chess status on the board. For sure, when either player chooses to surrender, the game ends and the other player wins. Each cycle of conditional judgment needs to keep track of and update the latest status of the chessboard in the system data and visualize it in the user interface in real-time.

By using two columns to distinguish whether the process belongs to the user interface operation or the system operation, the control logic of the user input to the system is more intuitively displayed.

3 A Brief Description of design

3.1 A brief description of the whole design



Above is the file structure of this project. In this project, our structure follows the MVC model, dividing the component into four parts.

- **The first part: Jungle.py.**

Jungle.py is designed as the main function to start this project. Since we use an MVC model, we import other parts to support the game start. This way, the code design will be clearer and the structure more flexible. In this part, the players are required to choose their chess color and who goes first. The jungle imports other components to support players' needs and start the game.

- **The second part: Model**

In the model part, we have employed chess.py, players.py, and model.py.

For chess.py, we use a chess class with all the basic attributes that chess should have. And since some chess has different functions, I separate to deal with them and use an instance method to reduce code redundancy.

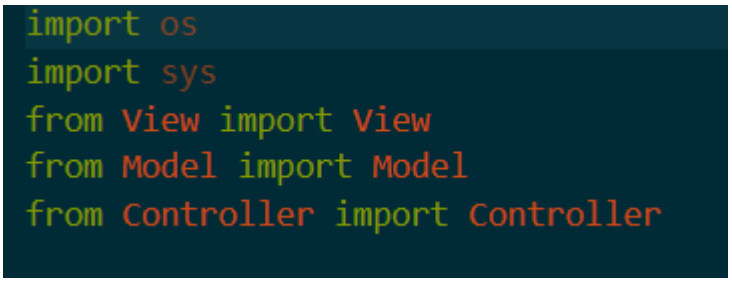
The player.py is defined for two players.

The model.py is defined to better print the situation of the game.

- **The third part: View**

In the view part, we print the message which we use to interact with the players. Such as print the game board, the user tips, game rules and so on.

- **The fourth part: Controller**



The controller part deals with the whole system's non-functional requirements and different user inputs.

For example, this part will help to control which player goes first, the rest time of the game, and the player's turn. Also, if the users input different commands, the controller will help deal with their information, which may depend on the view and model parts.

3.2 Model component

- Players.py
  - Constructor:

```
def __init__(self) -> None:
    pass
```

  - Attributes:

    the instance of the two players

- model.py
  - Attributes:

    the instance of the game model

  - Methods:

    `ifCanMove(self, moving_animal: Animals, direction) -> tuple:` This function is used to check whether the animal can move to next step.

    `move(self, moving_animal: Animals, direction) -> None`
    This function is used for change the moving animal's position.

    `jumpOver(self, moving_animal: Animals, direction) -> None`
    This function is used for change the moving Tiger/Lion position if they want to jump over a river.

    `if_new_position_has_enemy_that_can_be_eaten(self, moving_animal: Animals) -> bool`
    This function is used to check that if the new position where the animal move to has an enemy that it can eat.

```
get_same_position_enemy(self, moving_animal: Animals)
-> 'Animals'
```

This function is used only when if_new_position_has_enemy_that_can_be_eaten() is True. Because it's confirmed that the animal has moved to the square that has enemy pieces that it can eat, this func will search the exact enemy piece that will be eaten

```
die(self, dying_animal: Animals) -> None
```

This function is used after found the enemy to be eaten. The enemy piece's attribute of "status" will change from True(live) to False(dead).

```
get_estimated_new_position(self, moving_animal:
Animals, direction) -> tuple
```

get the estimated position of an animal's move

```
if_move_out_of_range(self, position: tuple) -> bool
```

check if the position is out of range of a chessboard

```
if_next_step_in_land(self, new_position: tuple) -> bool
```

check if the new position is in a land square

```
if_position_has_same_side_animal(self, moving_animal:
Animals, position: tuple) -> bool
```

check if the new position is occupied by a same side animal

```
if_position_has_same_rank_enemy(self, moving_animal:
Animals, position: tuple) -> bool
```

check if the new position is occupied by a same rank animal

```
if_position_has_lower_rank_enemy(self, moving_animal:
Animals, position: tuple) -> bool
```

check if the new position is occupied by a lower rank enemies

`if_position_has_higher_rank_enemy(self, moving_animal: Animals, position: tuple) -> bool`

check if the new position is occupied by a higher rank enemies (Rat is an excluding example here)

`if_next_step_in_river(self, new_position: tuple) -> bool`

check if the new position is in a river square

`if_position_has_enemy_Elephant(self, moving_animal: Animals, position: tuple) -> bool`

check if the new position is already occupied by an enemy elephant

`if_position_has_enemy_Rat(self, moving_animal: Animals, position: tuple) -> bool`

check if the new position is already occupied by an enemy rat

`if_position_has_lower_rank_enemy_except_Rat(moving_animal: Animals, position: tuple) -> bool`

This function is only used when moving_animal is Elephant. To check if the new position is occupied by a lower rank enemies except Rat.

`if_rat_in_that_river(self, side_of_river: str) -> bool`

This function is to find whether there is a rat on the specific river "side_of_river"

❖ Squares.py

➢ Constructor

```
def __init__(self):
    self.den_position = [(3, 0), (3, 8)]
    self.trap_position = [(2, 0), (4, 0), (3, 1), (2, 8),
    (3, 7), (4, 8)]
    self.river_position = [(1, 3), (1, 4), (1, 5), (2, 3),
    (2, 4),
                           (2, 5), (4, 3), (4, 4), (4, 5),
                           (5, 3), (5, 4), (5, 5)]
```

```
def __init__(self, name, rank, position: tuple,  status):
    self.name = name
    self.rank = rank
    self.position = position
    status = True
    self.status = status
```

➢ Parameter:

chess name, chess rank, chess position, chess status, whether in a trap

➢ Attributes:

the chess instance of the game

➢ functions:

`getRank(self) -> int:` get the animal's rank

`getPosition(self) -> tuple` : get the animal's position

`ifInTrap(self) -> bool:` check if the animal is in trap

`ifInLand(self) -> bool`:check if the animal is in land

`ifInDen(self) -> bool:` check if the animal is in den

`ifInRiver(self) -> bool:` check if the animal is in river

`upMove(self) -> None:` move up 1 step, its attribute of the position will change from (x,y) to (x,y+1)

`downMove(self) -> None:` move down 1 step, its attribute of the position will change from (x,y) to (x,y-1)

`leftMove(self) -> None:` move left 1 step, its attribute of the position will change from (x,y) to (x-1,y)

`rightMove(self) -> None:` move right 1 step, its attribute of the position will change from (x,y) to (x+1,y)

`jumpOverRiverUp(self, direction)-> None:` jump forward three steps,its attribute of the position will change from (x,y) to (x,y+1)

`jumpOverRiverDown(self, direction)-> None:` jump back three steps,its attribute of the position will change from (x,y) to (x,y-1)

`jumpOverRiverLeft(self, direction)-> None`: jump towards left two steps,its attribute of the position will change from (x,y) to (x-1,y)

`jumpOverRiverRight(self, direction)-> None`: jump towards right two steps,its attribute of the position will change from (x,y) to (x+1,y)

`ifInTrap(self)-> bool:` decide whether the chess in a trap

`changeStatus(self)-> None:` when a chess dead, change its status

3.3 View component

- Methods

`printWelcomePage()-str:`

This function displays a welcome message in the interface before the game starts. The welcome message includes a welcome sentence, a system brief introduction, a game rules description, and a brief user manual.

`askPreference()-str:`

This function displays a preference-asking sentence in the interface, including language preference and color preference. To attract more users from different language backgrounds, we design 2 language versions: Chinese and English. The color preference is chosen to distinguish between the player's cheese.

return: None

`printChessboard(player1: Model.Players, player2: Model.Players):`

This function takes 2 Players' objects as parameters, and returns a well-organized and clearly visualized chessboard. Each chess in a Players object will be well-displayed according to their coordinate positions. The river area, traps area, and den area will be laid in the correct positions.

`printHelp()-str:`

This function displays a list of helpful information in the interface, including a system brief introduction, game rules description, and a brief user manual. The user can search for help once they forget any regulation and operation skills.

`printHints()-str:`

This function displays different kinds of hints according to different invalid movements made by players.

- **Hint1**

  "illegal syntax": the player is only allowed to input a sentence in a form like "animal name + move direction (u, d, l, r)."

- **Hint2**

  "Moving out of board range"

- **Hint3**

  "Can't move into the river": Except for mice, no other animals can enter the river area

- **Hint4**

  "Lion/tiger is not allowed to jump over the river, because there is a rat on one of the rivers"

- **Hint5**

  "The rat is entering the river, it's only allowed to capture the opponent rat when they're both in the river"

- **Hint6**

  "The rat is entering the land; it's allowed to capture the elephants and rats(if the opponent rat is also on the land)"

- **Hint7**

  "A piece may not move to its own den"

- **Hint8**

  "You're going into a trap. Any opponent animal regardless of rank can capture you"

return: None

`printTiming()-str:`

This function displays a real-time countdown second during a player's movement decision period. The game is designed to be time-limited, as this enhances player focus and improves the sense of involvement in the game.

return: None

`printMoveHistory()-str:`

This function displays a player's chess move history, including which chess is chosen to move, how it moves,s and its captured results (if it has). The function is used once the player wants to review his/her movement strategy during the game.

return: None

`printCurrentRoundinfo()-str:`

This function displays the current round number in each round of the game. It's an additional function that aims to provide a better view of chess competitions.

return: None

`printCapturedResult()-str:`

This function displays a list of captured results in the current round when a player is making a movement decision. It provides a review so that players can know what his/her has captured instead of spending time observing the chessboard.

return: None

`printGameResult()-str:`

This function displays the game result about which player wins the game at the end of the game.

return: None

3.4 Controller component

❖ Constructor:

```
def __init__(self, game_model: Model, game_view: View):
    self.game_model = game_model
    self.game_view = game_view
```

❖ Parameters:

Game_model: model

Game_view: view

Player: player

❖ Attributes: game_model: Model, game_view: View

❖ Raise: Exception()

❖ Methods:

`chooseLanguage(self)-> None:`

Allow user to choose their preferred language to go through a better game experience. This will be called by Jungle.py during the execution of the game.

Return: None

`chooseSide(self)-> None:`

Require the user to choose one side, the upper side chess will be red color, and the downside will be green. Major related functions should get from Model.Model.

Return: None

```
executeInput(self, turnFlag: int)-> None:
```

Receive and parser the user input from the command line. Input will be parser to list for order identification. Parameter turnFlag is used to record which user's turn.

A boolean flag of a player's turn will be received to judge whether it was the player's round.
if move is selected, the function will check whether that was a legal move calling checkMove() and process the command;
if help is selected, printHelp() from view.py will be called, printing out the rules and instructions for the game;
if defeat is selected, admitDefeat() will be called, ending the game as the played claimed;
if exit is selected, print and call Exit() to exit Jungle.py.

Return: None

```
checkMove()-> bool:
```

Used to check whether the movement order is validated or not. If a user consistently inputs the wrong order, the system will regard this as malicious input and the user will suffer from punishment from the system.

The parameter cmd stores the player's move command with 4 possible directions: l, r, down, up. The size of the board is 9*7. Thus the vertical range should be [0, 8] and the horizontal should be [0, 6].

```
getHelp()-> None:
```

It will show game instructions again on the command line when the user inputs "help".

Return str: help information

```
Undo():
```

After both side permit one user could undo one round of chess, the system will ignore the user’s operation. Kindly remind this operation won't be recorded.

If an undo request is confirmed by input "y", check the number of previous undos. If already done 3 times then the request will not be granted.
If less than 3 times, ask for the other player's permission. If agreed, withdraw the player's move.

Return: None

`AdmitDefeat()-> None:`

The game will be terminated when one user admits his/her is defeated and have surrendered.

Return: None

`Exit()-> None:`

The game will over immediately after calling the function.

Return: None

`ifEnd()-> None`:

The system will monitor the whole process to check whether the game requires end conditions. Return: bool: True/False

`whichTurn(turnFlag)-> None:`

Used to validate the turn. Return: None

`time_spire()-> None`:

The total consideration time is limited to 10 minutes. Once the user's total time runs out, he/she will automatically be considered a failure. For the first five rounds, the time for each turn is limited to 30s, and starting from the sixth round, the time for each turn is limited to 90s. At the same time, the system will remind the user at the last 10 seconds of each round.

Return: None

3.5 Jungle.py

❖ constructor

`main():`
Functions that start the entire program and game, it calls the start function in class Jungle to start the game.

return:None

❖ parameters

`sys = Jungle()`
a variable with type of Jungle created in main function, working as the game operator.

The class Jungle has the following parameters:

`initPlayer1`, **`initPlayer2`**
Parameters which store the chess status for both players, which's class was imported from model.

Return: None

`game_view`
The parameter to load and store the chess display for interface.

`game_model`
The parameter to store the general model for the game, containing all the attributes for one game

`game_controller`
The parameter used for game controlling.

`inturn`
The boolean function judging the turn for players.

❖ methods

`start()`

defined in the class Jungle, start is to start the game which is stored in Jungle class.

Return: None

Jungle.py has imported View, Model in order to call their methods below:

`View.view()`

Initialize the game_view parameter defined in class Jungle.

Return: None

`Model.Model()`

Initialize the game_model parameter defined in class Jungle.

Return: None

`Controller.Controller()`

Initialize the game_controller parameter defined in class Jungle.

return: None

4 Important Design Decisions

4.1 Parts that are difficult to design

- **How to set a timer countdown for the user.**

Since we want to use the time to control which player goes next step and to make our game more formal, the step for each time Refers to the chess rules, for example, the length of each step is shorter at the beginning, and then the duration is appropriately longer. Therefore, the timing controller Involves a lot of logic, therefore becoming more complex to design.

- **How to fulfill the redo function**

Now our group's idea is that we use a temporary data structure to record the last whole chess state and if the redo is done, we then move back to the former state.

- **How to write each chess move into the txt file while redo is used**

Since when players could redo, therefore the txt file which record all the chess move will be much more difficult to record, we are still thinking about how to deal with it-put the redo move in the txt file or just remove it from the txt file, which needs to be solved in later study.

- **How to import other parts and use function calls and use the MVC model to construct the system**

Since we just learned the MVC model and this is the first time we use it as practice, therefore our group does not have a clear structure about how to combine the controller, view and model part. But we have the confidence to sort out the relations between them and work well.

4.2 parts that spent the most time coming up with

The part that spent the most time considering is the Players class. In the beginning, we didn't design a Player class until considering how to display a chessboard using the Object-Oriented Programming method. In an OOP way and MVC design pattern, displaying a chessboard is divided into the Model component, the View component, and the Controller component. Therefore, an object that can represent both player's side of chess and other fixed areas (river, traps, and den) is needed in Model component. That's what the Player class does. It is constructed by a list of Chess objects, a list of tuples of the river area, a list of tuples of traps area, and a tuple of the den area. The Player class is the parameter of printChessboard() function. In the View component, both sides' chessboards will combine and display in a well-organized way. In the Controller component, the chessboard is designed to display at the beginning of each round. So, the printChessboard() is called at every beginning of the game playing While Loop.

The other part that also spent much time considering is, how to integrate the model, view, and controller component. Once identify the purpose of each component clearly, it's easy to design functionalities in different components.

4.3 Parts that you think are smart and worth mentioning

Shining points of the project

There were many excellent ideas during the project implementation, and we successfully achieved them. Here is an illustration of these brilliant points.

- **User-friendly chessboard interface**

Two colors for two sides will discern the chess color. A user could easily recognize which chess belongs to themselves. At every turn, the user could type in "help" for detailed game rule instructions to facilitate their understanding of the game.

Kindly remind that although the system has a total time limitation, this time won't be counted in while users type in "help" and view the instruction, which means the system is paused.

- **The consummate rule is in a row with international standards**

It now contains both flexibility and delight. The game rules are designed as close to international standards. We also refer to both international and Chinese chess rules for game balance.

- **Exigent and exciting time limitation rules:**

Constrained time will bring more funs and challenge to the user. Also, the exigent time limitation refers to the ladder game principle—initial time will be easier, but with rounds going up and time reduction, how to win efficiently and quickly gradually becomes the priority. At this time, prescient decisions, brilliant brainstorming, and a clever scheme is the key to victory.

- **Will record the history options of users for reviewing**

Users' history options will be recorded for later review. Every step, what chess the user moved is tracked and written into a txt file. This could better help users to go over your history. Based on this, the user could better upgrade their skill by tracking the flaws exposed in such history. Meanwhile, the opponent's options will be also tracked, too. Hence users could find out the checkmate they may miss and excellent choices made by both sides. Reflecting yourselves from history could correct one's behavior.

- **Allow user to undo their choice while gaining their opponent's permission:**

To provide a flexible and impartial game environment, the user is allowed to undo their one-step early option with the opponent's permission. So basically, this could help a user better understand the game and better participate in brain competition.

4.4 parts that are critical for the game

- **The operation of the system needs to adapt to and serve the API**

API is directly oriented to users, so the design idea of API represents the needs of users, that is, functional requirements.

The core part of the game also needs to present the updated board data in each round to the player in real-time during the running process. In addition, when players have needs such as regrets or

checking the rules of the game at all times, the system should also make corresponding adjustments to ensure that all needs are perfected.

- **Simplify API calls to the system**

The storage and function packaging of any information, including the board, the progress of the game, and the state of repentance, should be strictly limited to the system, only provide some global functions for API class calls, and the number of calls should be enough to provide players. Do the minimum on the premise to avoid redundant steps when the API interacts with the system. In addition, the functions of the internal process of the game should not be visible and called by the API, such as functions that determine whether the game is over. They should be encapsulated in the system's class and specifically provide an output function for the API that returns the result variable.