Task 1

This software system includes 6 packages and additional class Launcher which contains the main method. Each package has up to 10 classes. The first step is identifying connections of classes between packages. The second step – within the packages.

Step 1:

i) and ii) If we were to draw all the connections between all the classes, things would get messy and not so informative. The optimal level of abstraction seems to be on the package level with showing some precise classes when it seems to be necessary (highlighting special features of the system). Reasons: first, we make use of the fact that the designer of the software took time and effort to design software well, made it readable, organized classes in packages and named the packages and classes meaningfully. Second, we get a big picture of what is happening in the software system without going too much into detail. At the same time, we highlight these special features (e.g., when one class heavily relies on the rest of the system)

The best way to proceed: run a software system (using the main method in Launcher class) and see what it does. Once it's clear, relate the packages to the app. Find an entry point, frequently it is a good idea to start with the class containing the main method – "Launcher" in our case. To draw the diagram between packages we use import statements.

iii) what we understood:

1. In the package "sprite" only class "PacManSpirites" relies on the external classes, hence we highlight this class. As regards other packages, they rely on PacManSpirites, Sprite and AnimatedSprite only (mainly on the first two). We show this feature to underline that any changes made in these classes would lead to consequences in the rest of the system. The rest of the classes in this package are not imported anywhere. Hence, we do not show them in the diagram and abstract away from these details.

2. All packages are dependent on the package "board". All the classes of this package are imported in some other packages. Hence there is no need to show the exact dependencies and list all the classes, here we abstract on the package level. On the contrary – package "board" depends only on two classes – "Sprite" and "PacManSpirites" – both members of the package "sprite". This feature is also shown. Conclusion: any changes made in package "board" would have consequences in other parts of the system, whereas the package "board" itself does not depend much on the external classes.

3. Only package "ui" and class "Launcher" rely on the package "game", specifically only on 2 classes: "Game" and "GameFactory". We highlight this feature in the diagram as well. Similarly, only these two classes import stuff, hence we show them in the diagram.

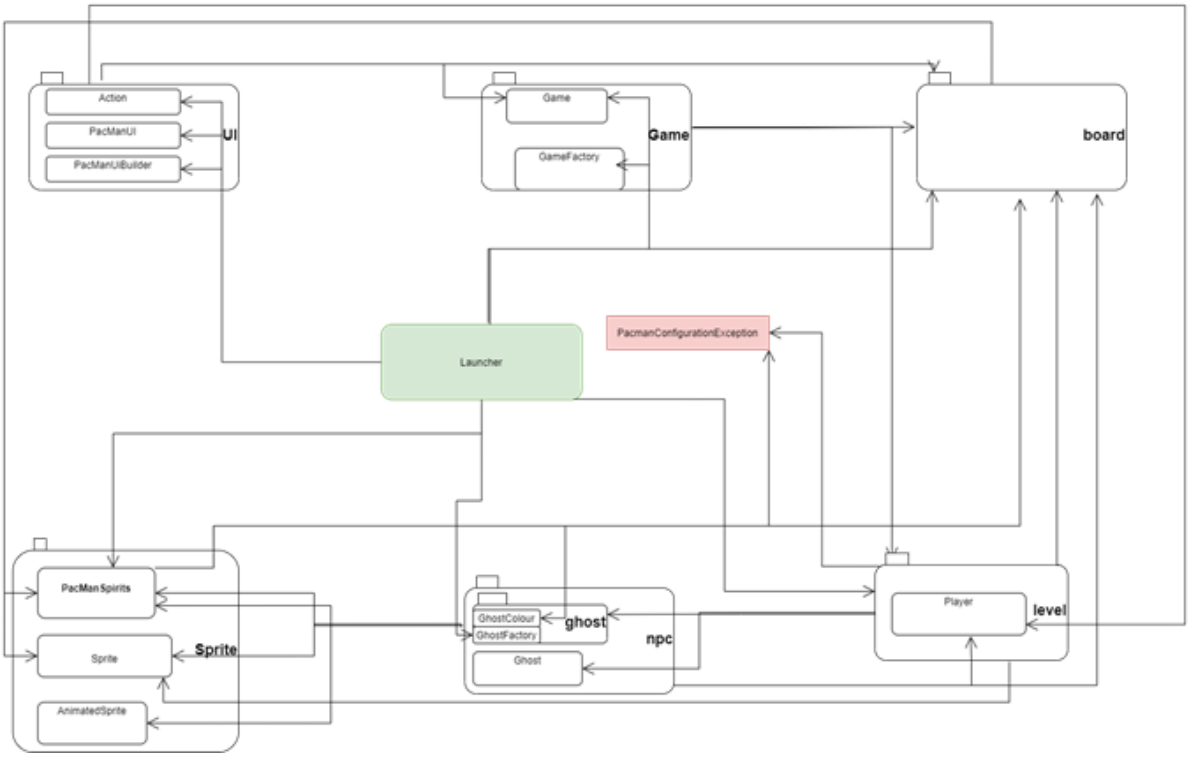
4. As regards package "ui", only class "Launcher" depends on it, because it uses ui to build the interface of the game, build and start the game. Specifically, it imports interface "Action" and classes PacManUi and PacManBuilder. Similarly, mostly only these classes

participate in importing stuff to ui. Therefore, they are shown in the diagram. Conclusion: this package is not much coupled with the rest of the system.

5. As regards package “npc”, it contains subpackage – “ghost” and another class “Ghost”. Whenever something is imported from the package “ghost” it is either “GhostColor” or “GhostFactory”,

6. Finally, many packages rely on the classes from the package “level”. Class “Player” is imported in all cases. Hence, we show it in the diagram, and abstract away from other classes.

Summing up, the diagram shows that 1. package on which everyone relies – “board” – all the classes there are pretty frequently used outside of the package, 2. in other packages such classes as “PacManSpirites”, “Sprite”, “Player”, “GhostColour” “GhostFactory”, “Game” are also used by at least 2 external packages. 3. Class “Launcher” which includes the main method relies on each part of the system. 4. In the package “sprite” everything except for “PacManSpirites” does not import anything.



Step 2:

Looking at packages closely. We comment out classes and see what breaks in that package. Adding the inter-package relations to the diagram above would make it unreadable. Therefore, we just list the most prominent features identified while analyzing inter-package relations:

1. Normally (except for sprite), in each package there is a class which depends on almost all or all remaining classes in the package. These are either classes with “-Factory” in name (such as GameFactory, GhostFactory, BoardFactory, GameFactory) or class PacManUI. -Factory classes would be responsible for creating objects and therefore need to import stuff from the rest of the package. PacManUI probably brings all the parts of ui together. This is quite intuitive from the names without even looking at code.

2. Other classes within the package almost never rely on -Factory.

3. Other classes communicate between each other to a limited extent. E.g. in the package npc Binky, Clyde, Inky and Pinky do not communicate, as they represent the various types of ghosts and do not need to rely on each other. Similarly in the package ui there are a bunch of classes responsible for panels and Keylistener. They, as it would be expected, do not communicate with each other, as they are independent parts of the UI.

1.2

Launcher.main()

-> Launcher.launch()

-> Launcher.makeGame()

-> Launcher.makeLevel()

->getMapParser().parseMap(levelMap)

->BoardFactory.createBoard()

->LevelFactory.createLevel()
->LevelFactory.createGhost()
->ghostFactory.createBlinky() /Inky()/Pinky()/Clyde()

-> getGameFactory ().GameFactory.createSinglePlayerGame (level)

->playerFactory.createPackman()

-> Launcher.addSinglePlayerKeys (builder)

-> PacManUiBuilder.addKey (Builder)

-> Launcher.moveTowardDirection(Direction)

-> PacManUiBuilder.withDefaultButtons().build(game)

-> new PacManUI (game, buttons, keyMappings, scoreFormatter)

-> PacManUI.start()

-> service.scheduleAtFixedRate(PacManUI.nextFrame ,Frame_Interval)

i) - ii) As entry point the main method in Launcher is the most obvious, since it is what one calls when running the program. It does only one thing, calling the launch method which is also in the Launcher class. The launch method then continues to launch the game, which makes the structure more visible: 1. Make game 2. Add keys 3. Build the game 4. Start the game. This gives a big picture of what is happening.

1. Make Game - the depth was chosen such that reach calls of Factories were reached (where all the game elements are constructed): so it is clear that after calling MakeGame, first, level and map are constructed, which implies construction of Ghosts and Board, then the level is passed as an argument to GameFactory which create single player game and this implies building Packman as well.

2. Once Game is ready, keys are added which correspond to move directions. Here we do not go deep, as its technical implementation details, which we do not need to understand the system right now.

3-4. After that the game is built and started.

iii) The packages game, player, level, board and ghost all contain a "-Factory" class. External methods are accessing the packages via this class. The "-Factory" class is responsible for creating and returning the corresponding object. All the "-Factory" classes are accessing the SpriteStore class which is accessing the resources package. This is where it comes back together, all the information is stored in one central place (graphics, board layout etc.). The ui package is responsible for building the interface of the game and starting it.

Task 2

Step 1: underline all the nouns in the requirement.

In this game, two users play Checkers using the same computer and input their moves at each round via the terminal. When the game starts, the program outputs—on the terminal—the board with the pieces in their initial position. The board should be displayed as shown in the following example:..

The first character ('R' or 'W') represents the piece's color (Red or White) and the second one the piece's type (i.e., 'P' = Pawn, 'K' = King[3]). Each row of the board is indexed by a number (starting from the bottom row with '1') and each column is indexed by a character (starting from the left with 'a').

The game then asks the first player to enter their move on the terminal. Users declare their moves as in the following: [current piece position]X[future piece position]; for example by typing [a3]X[b4] they want to move a piece from position a3 into position b4. The validity of the move must be checked; if not valid the user has to enter another move until they enter a valid one. After the player inserted a valid move, the program prints an updated version of the board based on the new positions of the pieces. Afterwards, the program asks the next user to move, unless the game is finished, in which case it prints out who the winner is.

Step 2: exclude physical objects, attributes, etc. Form a list of conceptual entities which will constitute candidate classes:

Physical objects: computer -> do not need

Interfaces to the system terminal, program -> do not need to model them.

- Other than classes:

    - Position or Column & row (probably something else: attribute of Move e.g.)

    - Number and character (input type: int, char)

Grouping:
Version of the board and board -> Board.
User, the first player, the next user, winner -> Player (initially not clear whether we will need this class, hence we kept it in the initial candidate classes set).
Validity - class for checking the move validity / rules (should be called differently).

These steps helped to refine the list down to 6 classes, motivated by a very broad idea of each class's responsibilities:

- Game
- Move
- Piece
- Board
- Player
- Validity

Step 3: Made CRC cards with only classes' names

- Looked through requirements specification for verbs. Based on the relevant verbs assigned the responsibilities to the previously chosen classes and further refined the list of the classes in the process. Based on responsibilities the collaborators were added to each class.

Major decisions which were made while playing the scenarios using CRC cards:

1. The class Piece was controversial, because we have a simple setup and could model the board as an array of strings. However, this would make the code not reusable. Assume, we would like to build Chess and would like to reuse Board. If it contains strings representing pieces of checkers, it would imply rewriting the whole Board code. On the contrary, when we model Piece explicitly and make Board an array of Pieces, this allows us to reuse Board in other set-ups. It follows the responsibility driven design. Piece stores its color and type, can change it and knows how to print itself. Board takes care of the board state only, without knowing how Pieces are stored.

2. Class Validity is renamed to RuleEvaluator, since it will handle the logic of the game and this name would better reflect its responsibility.
3. The only function of the class Player is keeping track of the current player. Because the game is simple ((player does not have name / age / score history, etc) we made the decision not to model Player explicitly. (Creating 2 instances of 2 players - does not make sense, storing it as a static class - and to access public static fields - also not a good option). In a more complex set-up, where one would expect such responsibilities as setName, changeName, updateScore, etc Player would be necessary, but not in our case. Hence, the function of keeping track of the current player was delegated to the class RuleEvaluator.

   The resulting cards:

   - **Game:**
     Responsibilities:
     - main (initialize the game, create instance of Board)
     - nextMove (askforInput, readInput save as coordinates and create Move object once valid input is received) - iteratively

     Collaborators:
     - RuleEvaluator
     - Board
     - Move

   - **RuleEvaluator:**
     Responsibilities:
     - checkValidity
     - checkForJumpMoves (-ForSimpleMoves)
     - checkWinner
     - keep track of currentPlayer and updateTurn

     Collaborators:
     - Board

   - **Move:**
     Responsibilities:
     - move (which calls jump- / simpleMove)
     - simpleMove
     - jumpMove

     Collaborators:
     - Board
     - RuleEvaluator
   - **Board:**
     Responsibilities:
     Providing information about the current state of the board:
     - printBoard
     - isKing / -Red / -White / -Empty.

Updating the current state of the board:
- resetBoard
- movePiece
- removePiece
- changeType

Collaborators: Piece

- **<u>Piece:</u>**
  Responsibilities:
  Storing and providing information about its type and colour
  Returning label for printing it (getLabel)
  - Change its type

  Collaborators: none

2. As pointed out earlier the 5 main classes are: Game, RuleEvaluator, Move, Board, Piece.

- The class **Game** is responsible for general coordination of the game: starting the game (main method with instantiation of Board), repeatedly initiating the next move by asking for input, reading it, asking for validation and instantiating Move whenever valid input is received.

- The class **RuleEvaluator** as the name suggests would make sure that the rules of the game are followed. Its responsibility - containing the logic of the game. 1. It would ensure that the move that was put in is indeed valid, check for simple or jump moves. 2. In case of entering a jump move it would check if further jumps are possible. 3. After each round it would check if a winner is determined and update the turn.

- The class **Move** is responsible for actual implementation of the move depending on whether it is a simple or a jump move. It stores coordinates of a valid move. The decision was made not to implement simple and multiple jump move methods separately, but rather to have jumpMove only. It asks to check if further jumpMoves are possible, if not -> it asks to update turn and initiates next move, if yes - > it initiates next move without updating the turn. Whenever a jump move is possible, a simple move is not accepted. Hence the same player will be forced to continue the jump move until the move is finished, and only after that the turn will be changed to the other player. The main reason for this: if we ask the player to input start and end coordinates of a multiple jump move, there are situations when you can make the move in different ways and therefore one would need to reask the player which way he would take. This would complicate the design of the game. Hence we chose this approach as it is the most efficient.

- The class **Board** is responsible for the current state of the board. 1. It stores the current state of the Board with Pieces and can provide information whether the cell is empty or not, and when not can request the relevant information from the Piece. It is

not aware of how Piece is stored, which provides encapsulation advantage and generally its not its responsibility. 2. It makes changes to the state of the board when moving / removing Pieces (when it is requested by the class Move).

- The class **Piece** stores information about color and type of the piece on the board, can provide this information (getType / -Colour), knows how to return the label of it (for printing out) and can change its type.

Collaborators:
The system is decoupled in a way that not every class has to collaborate with one another.

- **Game** collaborates with 3 classes as it coordinates the whole Game. First, it creates an instance of Board when initializing a new game. Second, whenever input of move is received, it passes the coordinates to RuleEvaluator to check the validity. Third, if the move coordinates pass the validity test, it creates the Move instance, calls the function move from the class Move and passes the board. Hence, it has 3 collaborators: Move, RuleEvaluator and Board.

- **Move** collaborates with Board and RuleEvaluator. First, whenever a move is executed it has to be checked whether it is a simple or a jump move, whether multiple moves are possible in the case of jump moves, and turn has to be updated. Hence, Move calls the relevant functions of the class RuleEvaluator. Second, when pieces are required to be moved / removed / changeTypes Move calls the relevant functions of the class Board to update the board state.

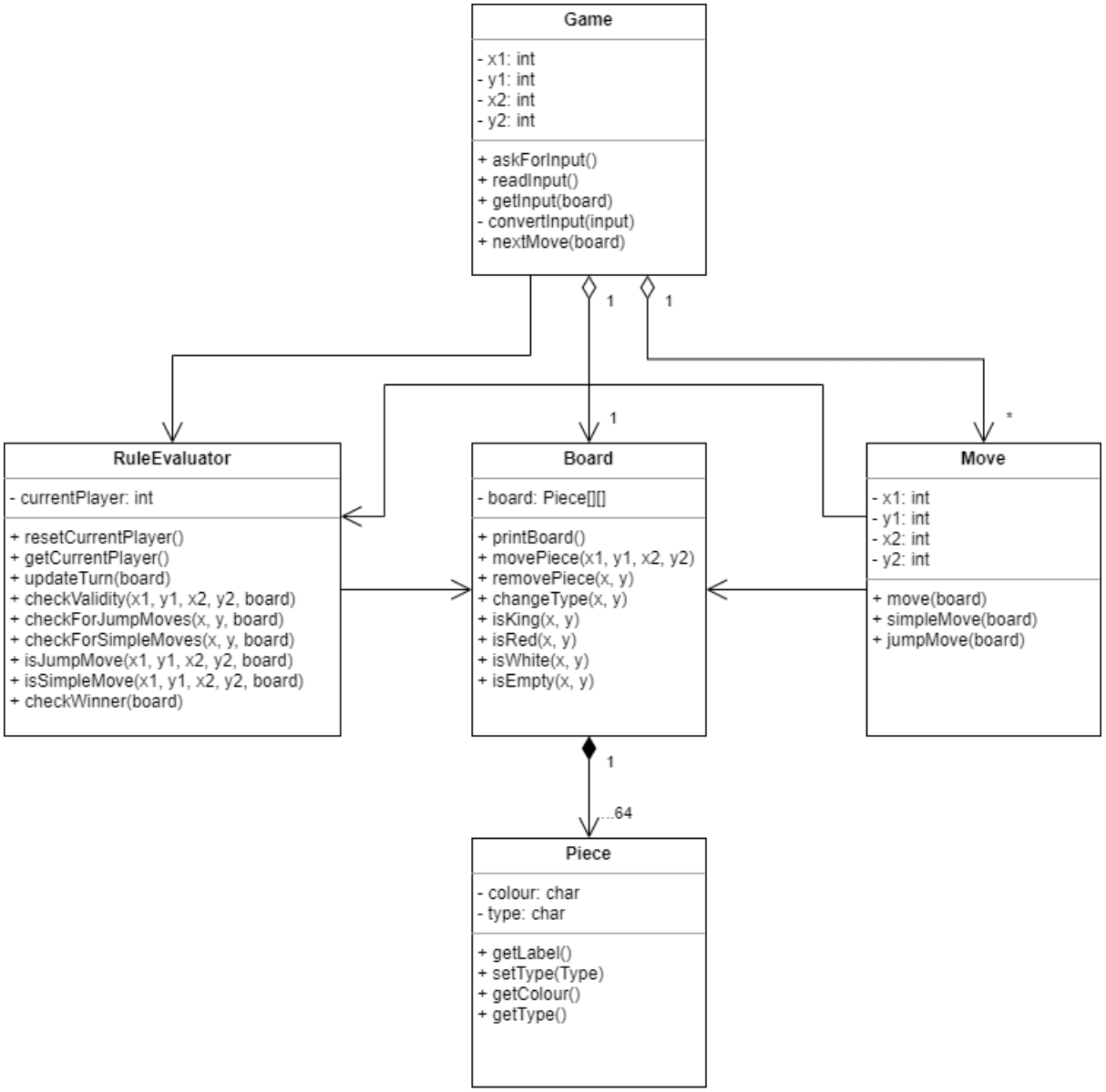
- **RuleEvaluator** collaborates with the Board only whenever it needs to get information about the current state of the board.

- **Board** collaborates with Piece when it needs to ask about its type or colour, or label to print and update the type.

3. Class Player was considered less important as explained earlier. In such a simple game, it doesn't get enough responsibility. Therefore, class Player was reduced to the variable currentPlayer in RuleEvaluator and RuleEvaluator then received the necessary functions to work with the variable currentPlayer.

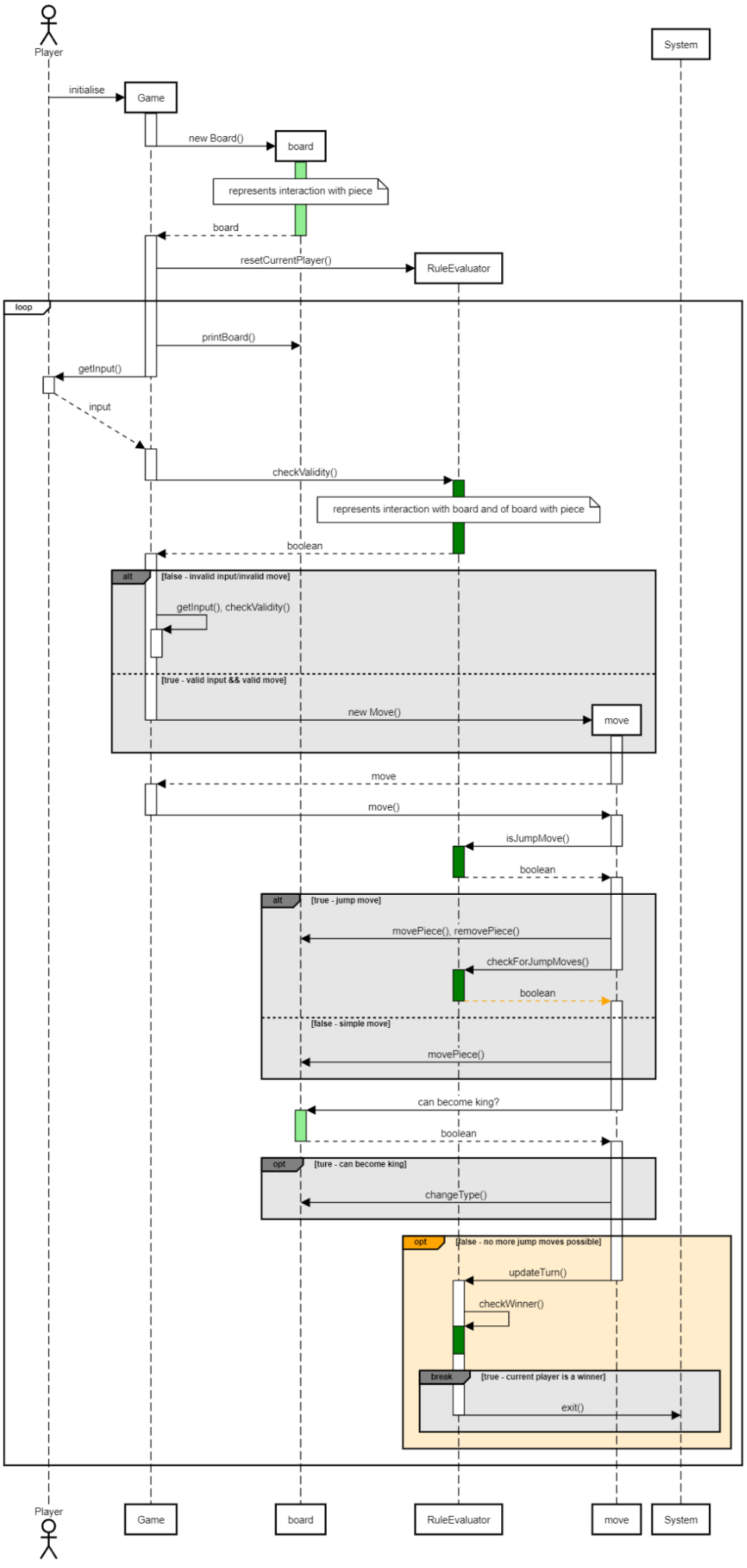
Step 4: Class diagram

1. Following the guides and examples given in the lecture, it was decided to use composition between Piece and Board, as the object Piece, vaguely speaking, cannot exist without the object Board. On the contrary, Move and Board can exist without Game, e.g. RuleEvaluator can interact with Board. Thus the use of different arrows is intended to represent the difference in those relations.



Step 5: Sequence diagram

1. As Piece is only interacted with Board and as these interactions are, in a sense, almost trivial, we decided to leave it out of the diagram to make it more readable. Instead these interactions are represented with light green.
2. The dark green colour represents interaction of RuleEvaluator with Board, and then of Board with Piece again to keep the diagram readable.
3. The orange colour highlights the returned boolean from checkForJumpMoves(), as it is used much later in the diagram to see if the turn needs to be updated.


Player
System
initialise
Game
new Board()
board
represents interaction with piece
board
resetCurrentPlayer()
RuleEvaluator
loop
printBoard()
getInput()
input
checkValidity()
represents interaction with board and of board with piece
boolean
alt
[false - invalid input/invalid move]
getInput(), checkValidity()
[true - valid input && valid move]
new Move()
move
move
move()
isJumpMove()
boolean
alt
[true - jump move]
movePiece(), removePiece()
checkForJumpMoves()
boolean
[false - simple move]
movePiece()
can become king?
boolean
opt
[ture - can become king]
changeType()
opt
[false - no more jump moves possible]
updateTurn()
checkWinner()
break
[true - current player is a winner]
exit()
Player
Game
board
RuleEvaluator
move
System