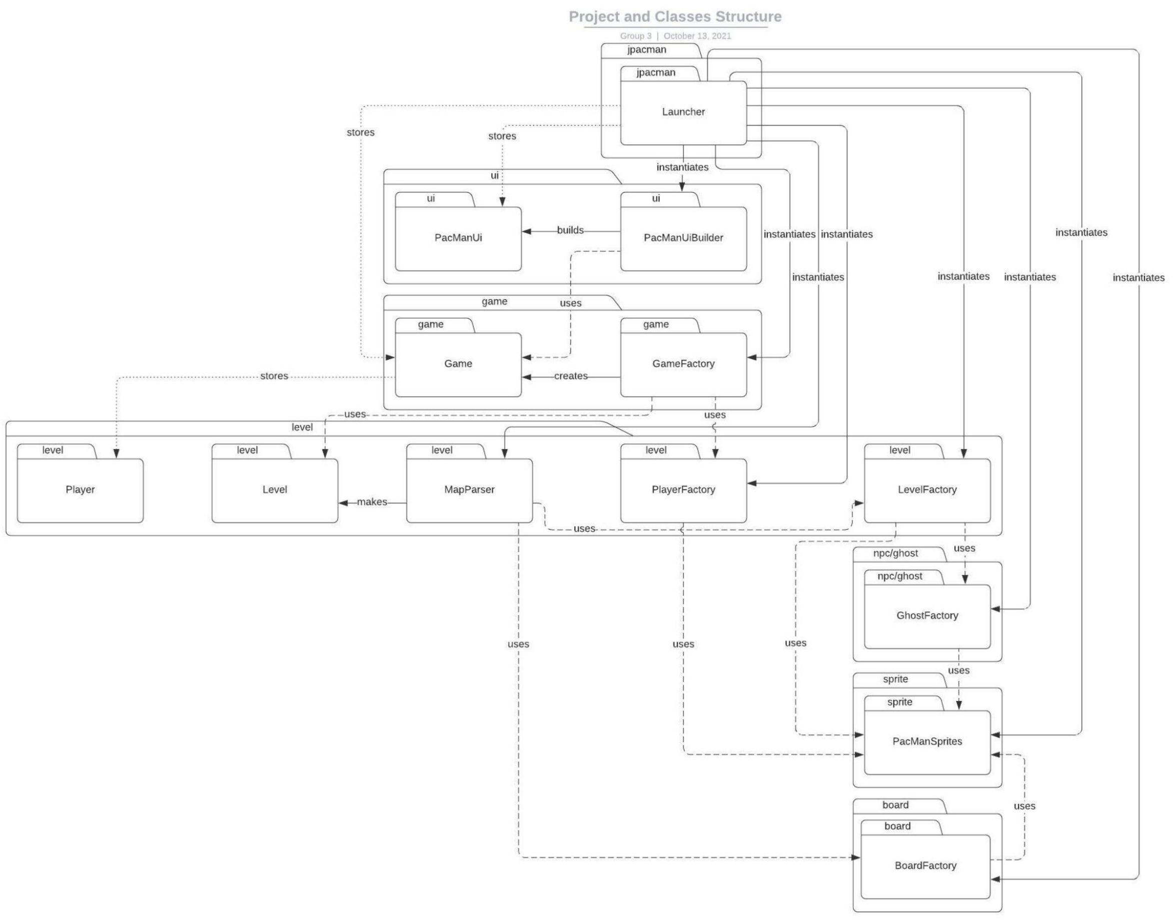
Kai Zinnhardt 19-763-176/ Jia Fu 21-907-662/ Paul Luley 21-741-491/ Joel Meier 16-916-959

Software Construction HS 2021 Group 03: Assignment 1



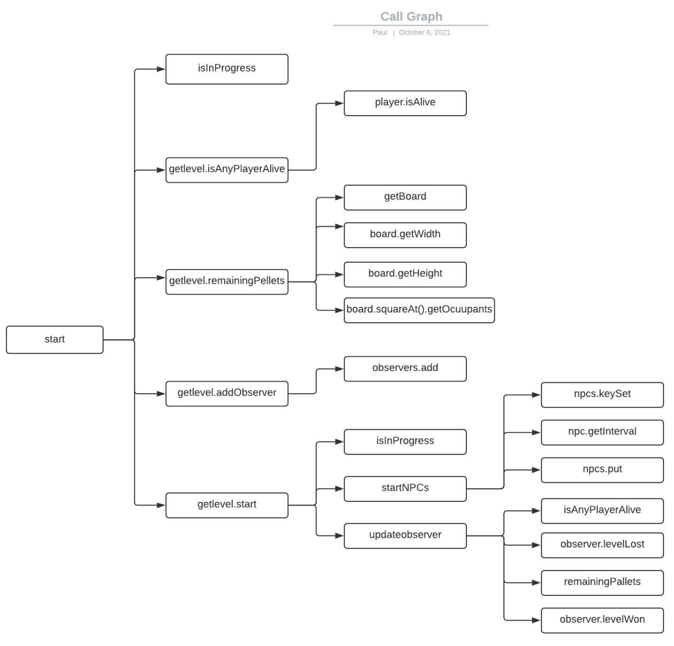
Exercise 1:

<u>1.1</u> In terms of level of abstraction we choose an intermediate one. Whereas a lower one would focus on the language that the system is written in, its semantic, and syntax, and a higher level would abstract the used methods, the chosen level of abstraction addresses the system's component's relations and how they interact with each other. [https://www.geeksforgeeks.org/abstraction-levels-in-reverse-engineering/]

<u>1.2</u> Since the "Launcher" class contains the main method and by that the starting point for the Java Virtual Machine to execute the program, it felt natural to us to select it as an entry point for our bird view of the project and classes structure.

<u>1.3</u> Superficially inspected, the project name suggests that it contains a Java based implementation of the game "Pacman".
After opening the project with an IDE and browsing to the "nl.tudelft.jpacman" package it became clear to us that it contains besides the class "Launcher" six further subpackages: "board", "game", "level", "npc", "sprite", and "ui".
Furthermore, we investigated what the execution of the main method triggers within the scope of the Launcher class. In line 7 - 20 all the project internal imports are listed which

implies that the mentioned classes are at least used within the Launcher class. The revealed namespaces of these classes made clear that all of the above mentioned subpackages are accessed from the “Launcher” class.
Although we didn’t dig deeper into any lower classes it was possible for us just by inspecting the initialisations to detect several inter-package and inter inter-file dependencies.
To begin with an object of type “Launcher” is constructed and its “launch” method is called. At first, this method creates the prerequisites to build an UI, i.e. a “Game”. Therefore, it creates a “GameFactory” that relies on a “PlayerFactory” and needs a “Level” to be able to create a “Game”. However, this “Level” is returned by a “MapParser” that in turn relies on a “LevelFactory” and a “BoardFactory”. To be able to create these factories a “GhostFactory” and a “PacManSprites” are needed. That’s why the Launcher has to interfere with all the above mentioned subpackages and is responsible for initialising the factories and the “MapParser”.
After the “GameFactory” made the “Game” the “PacmanUiBuilder” that is responsible for building the UI is created. It has to be equipped with the information about which key controls a move towards the directions “north”, “south”, “east”, and “west”.
With that information in place and the previously created “Game” the “PacmanUiBuilder” is able to build the UI that starts afterwards.



<u>2.1</u> This call graph starts inside the game class, as shown in the diagram.

<u>2.2</u> The call graph starts at the Start method, which is located inside the game class. This was decided, as it is the place where the program begins to execute the Pac Man game. Therefore, one can see the first step of how the program interacts and where some constraints and checks were implemented for this game.

For this call graph, 3 levels were used, this was done as one can include all the central methods from this starting position.

Furthermore a 4th level was decided not to be taken. As it would be a repetition of some of the methods displayed in the first 3 levels.

2.3 There are several behaviours that can be understood through this call graph. Firstly, as it can be seen through the “isInProgress" method, states are intialized, so that one can notice if the start Button is pressed multiple times. Here it is important to not restart the game every time the button is pressed. Secondly, the call graph includes many different kinds of checks, e.g. “getLevel.isAnyPlayerAlive", this is used to notice the dynamics in the programm. In the example mentioned above, the check refers to whether the player is alive or dead. Lastly, as this is the start of the PacMan game, the Ghosts (NPCs) are instantiated and start to move. As a side note, when looking at the code for the creation of this call graph one noticed how locks were used to prevent the program from misbehaving (e.g. the progressLock). Another note that has to be made, is that neither the moving of the PacMan nor the collision cases are covered in this Call-graph.

Kai Zinnhardt 19-763-176/ Jia Fu 21-907-662/ Paul Luley 21-741-491/ Joel Meier 16-916-959

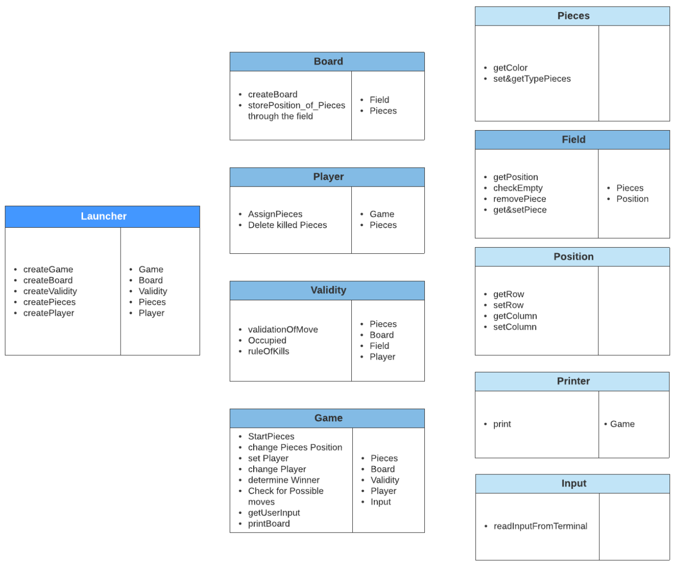
Exercise 2:

<u>1.1</u> We started off by searching for the noun phrases. We found the following nouns: game, users, checkers, round, move, boards, position, piece, character, color, type, row, number, column, validity, player, winner.

<u>1.2</u> After looking at those noun phrases we formulated our candidate clases. We did this by conducting a class rationale selection. Thereby we found conceptual entities (Board,Pieces, Game etc.). Additionally, we found several words for one concept (choosing piece - character, piece, checker). Furthermore, we also found values of attributes (move, number). Lastly, we renamed nouns, to clarify what they should do (round got renamed to turn). After doing this step we got the following list of candidate clases: Turn, move, validity, piece, player, launcher, board, game, king, pawn, row, column, position and player.

<u>1.3</u> Before starting the CRC Session we looked at the verb phrases that are inside the description. We got the following list of verbs: input, start, output, displayed, indexed, aks, enter, declare, typing, move, checked, insert, print, ask

<u>1.4</u> We started the CRC Session with the role play for the different scenarios. The main scenarios that we covered were: initialization and doing a move of checkers (therefore we had to check the validity, execute the move, and knowing when to change the turn). After the CRC session we noticed that we need to add the classes of the launcher, input and the printer. Additionally, we could discard the clases of: move, turn, pawn, king, and color (thereby we changed pawn, king into a pieceType and color into enumerations). This therefore leaves us with the following CRC Cards:



While coding we noticed that we need an additional class “Field”. This class is used, so that the 2D Board array knows what happens at each of its indices (its fields).

Kai Zinnhardt 19-763-176/ Jia Fu 21-907-662/ Paul Luley 21-741-491/ Joel Meier 16-916-959

2. In our checker's project, we have 4 main classes: Firstly, the Game class is identified as the main class. The Game has the responsibility of creating and starting the pieces on the board, thereby it also has the responsibility to move the pieces on the board (changePosition). For those tasks, the game needs to collaborate with the board, pieces and the validity. Furthermore, the game also has the responsibility to set and change the players turns, however it also needs to identify a winner. Therefore the game needs to collaborate with the player and the Validator class. Additionally the Game also takes in the input, so that the game can change the checkers, hence collaborating with the Input.

A second class is the Launcher. The Launcher initializes and creates the objects Game, Board, Printer and Input. To do those steps, it needs to collaborate with those classes and is essential to the game itself since the game.start() method resides in the Launcher class as well.

The third main class is the Input. The Input has the purpose of being ready to check new input commands and make sure length and grammar are correct. Thereby, the Input needs to collaborate with the Game. Additionally, it also has to have access to Position and convert the input .

A final main class is the Validator. Its purpose is to check if the move is valid or not. Thereby, it needs to cover all the corner cases such as: If the player can kill, then it has to do so. Or if the piece is a king, then it has more legal moves. To do so it has to collaborate with the Game, the Piece, Field and Position classes. Together with the Game class the Validator brings the most essential game mechanics to the table.

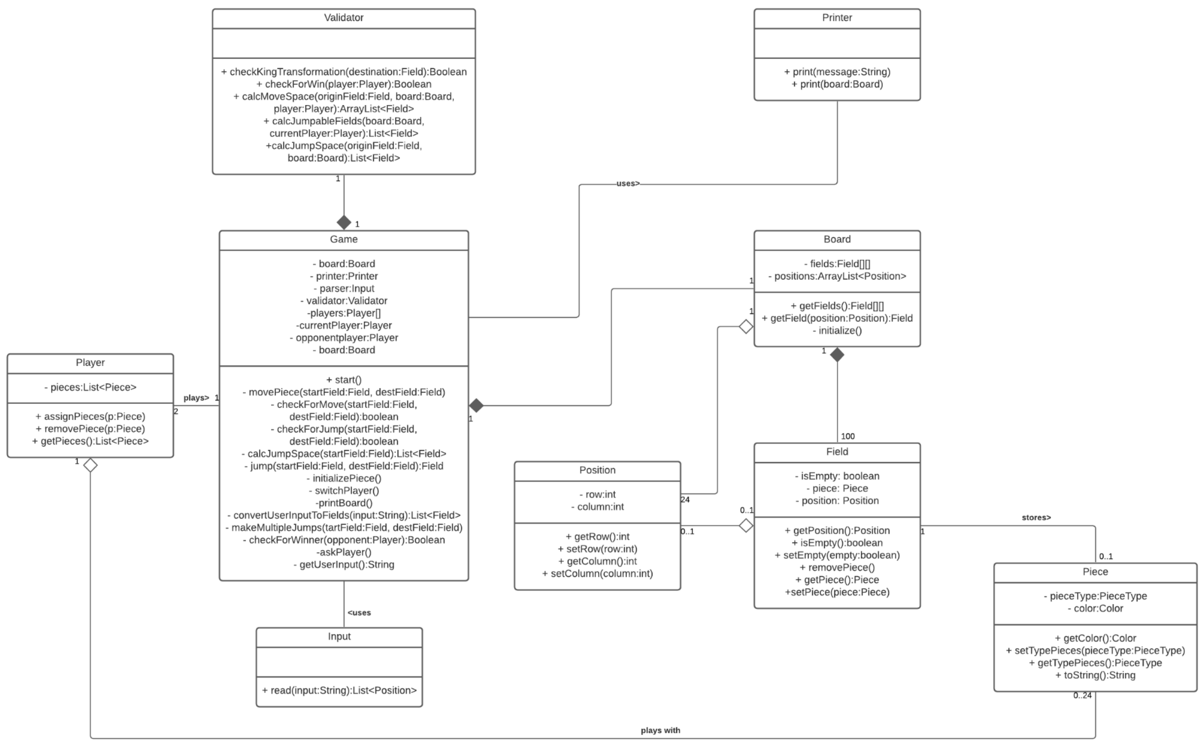
3. The other classes are less important, as they are considered to be the "helper classes". The Printer class only takes the commands (from the Game class) that it should execute and print in a specific way to the terminal. In Contrast, the Input has the responsibility to check and transform the user input for the moves.

The class Piece has the purpose to have PieceType and PieceColor. On top of that each Piece stores a Position and therefore collaborates with that class.

The Fields class mainly helps the Board to know what objects there are at a specific position on the board. The Board is initiated with a 2D Array of Type Field. This class was added during coding in order to simplify coordination and localisation of the pieces.

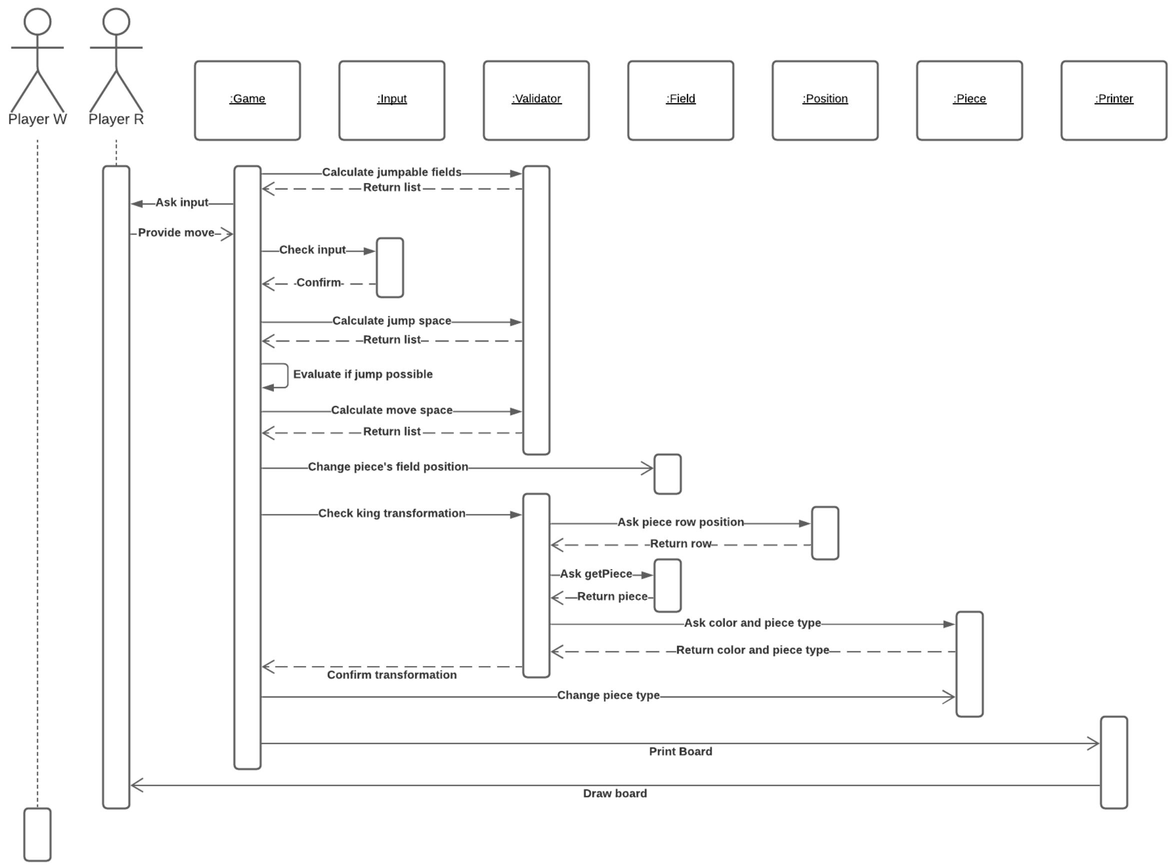
The Position class helps the Field and the Board class to store the information of the pieces on the Board. It therefore collaborates with the Field class.

4. We have constructed the following Class Diagram:


Validator
+ checkKingTransformation(destination:Field):Boolean
+ checkForWin(player:Player):Boolean
+ calcMoveSpace(originField:Field, board:Board, player:Player):ArrayList<Field>
+ calcJumpableFields(board:Board, currentPlayer:Player):List<Field>
+calcJumpSpace(originField:Field, board:Board):List<Field>
Printer
+ print(message:String)
+ print(board:Board)
uses>
1
1
Game
- board:Board
- printer:Printer
- parser:Input
- validator:Validator
-players:Player[]
-currentPlayer:Player
- opponentplayer:Player
- board:Board
+ start()
- movePiece(startField:Field, destField:Field)
- checkForMove(startField:Field, destField:Field):boolean
- checkForJump(startField:Field, destField:Field):boolean
- calcJumpSpace(startField:Field):List<Field>
- jump(startField:Field, destField:Field):Field
- initializePiece()
- switchPlayer()
-printBoard()
- convertUserInputToFields(input:String):List<Field>
- makeMultipleJumps(tartField:Field, destField:Field)
- checkForWinner(opponent:Player):Boolean
-askPlayer()
- getUserInput():String
Board
- fields:Field[][]
- positions:ArrayList<Position>
+ getFields():Field[][]
+ getField(position:Position):Field
- initialize()
Player
- pieces:List<Piece>
+ assignPieces(p:Piece)
+ removePiece(p:Piece)
+ getPieces():List<Piece>
plays>
1
2
1
1
1
1
1
100
Position
- row:int
- column:int
+ getRow():int
+ setRow(row:int)
+ getColumn():int
+ setColumn(column:int)
24
0..1
0..1
Field
- isEmpty: boolean
- piece: Piece
- position: Position
+ getPosition():Position
+ isEmpty():boolean
+ setEmpty(empty:boolean)
+ removePiece()
+ getPiece():Piece
+setPiece(piece:Piece)
1
stores>
0..1
Piece
- pieceType:PieceType
- color:Color
+ getColor():Color
+ setTypePieces(pieceType:PieceType)
+ getTypePieces():PieceType
+ toString():String
0..24
<uses
Input
+ read(input:String):List<Position>
plays with

5. Our sequence diagram can be seen on the next page. The scenario we chose to depict was a move with correct inputs by the user, a simple move (no jump or kill) to the opposite edge of the field and therefore a piece type change from “pawn” to “king”. This best illustrates the way two of the most important classes, Game and Validator, interact with each other. As we can see the Game class coordinates all steps and guides the checkers game overall. The Validator checks whether moves etc. are possible or allowed and covers many different edge cases. Also, both of the main classes make use of various helper classes in order to get information or change attributes. At the end of day the above mentioned classes carry the heavyload of the work and cross-check patterns, possibilities and carry out actions or at least route them to the respective helper classes.

Kai Zinnhardt 19-763-176/ Jia Fu 21-907-662/ Paul Luley 21-741-491/ Joel Meier 16-916-959


Player W
Player R
:Game
:Input
:Validator
:Field
:Position
:Piece
:Printer
Calculate jumpable fields
Return list
Ask input
Provide move
Check input
Confirm
Calculate jump space
Return list
Evaluate if jump possible
Calculate move space
Return list
Change piece's field position
Check king transformation
Ask piece row position
Return row
Ask getPiece
Return piece
Ask color and piece type
Return color and piece type
Confirm transformation
Change piece type
Print Board
Draw board