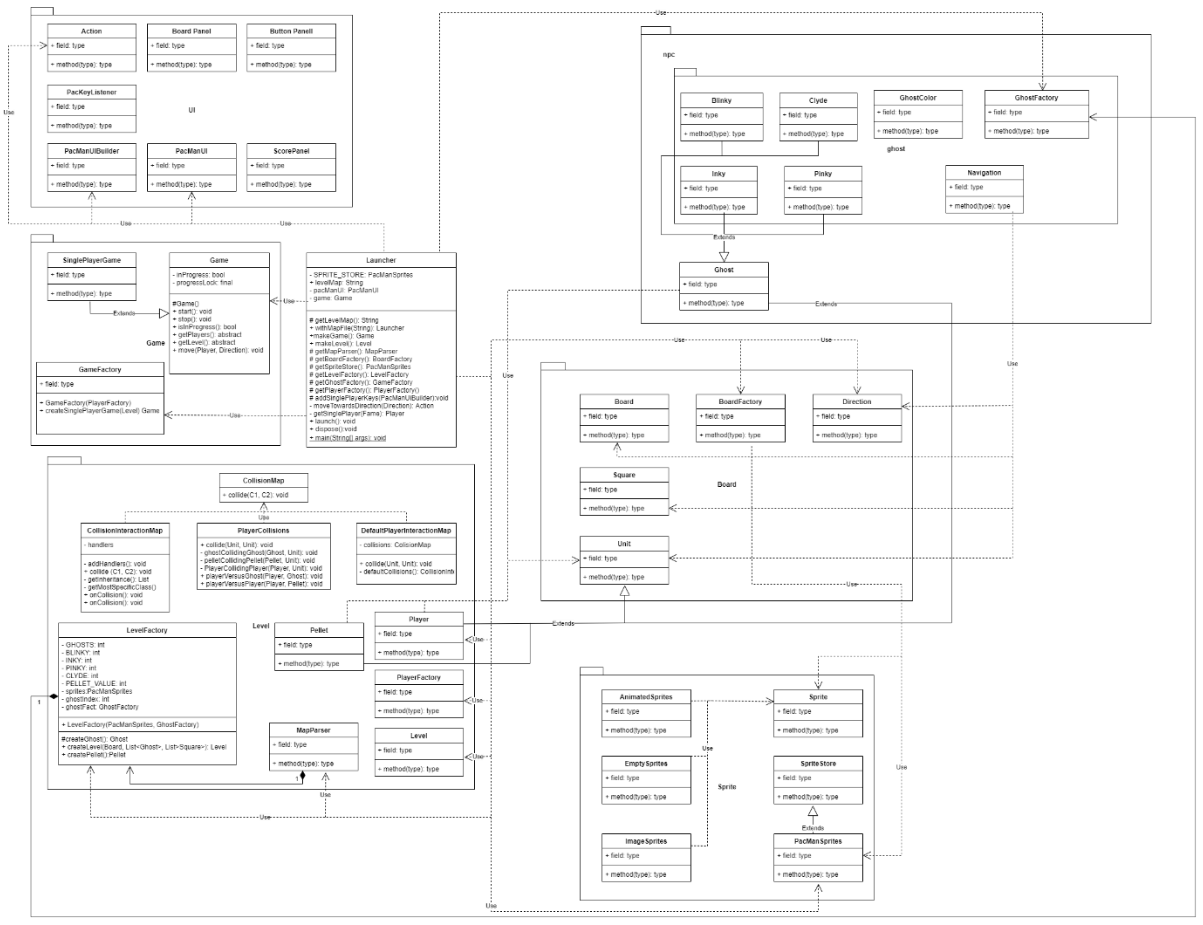
Exercise 1 - A Bird's-eye View of Pacman

1.1



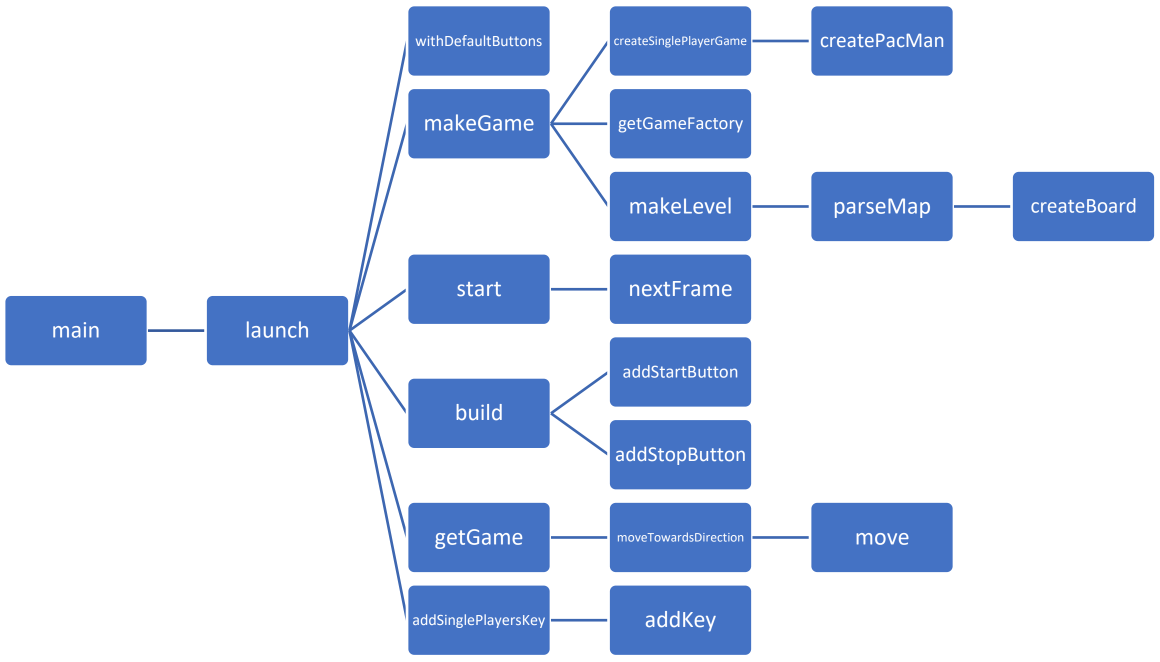
https://app.diagrams.net/#Htheraffael%2FSoftware-Construction%2Fmaster%2FAssignment1%2FAssignment%201

We decided to model the packages as well as the classes and some (but not all) relationships between them. Like this we can see the bigger picture of how the game functions as well as a reasonable level of details. We omitted the resources root to not make it too complex.
In general, we can see that the **Launcher** class has a central role which makes sense as it is the main class.
From the different packages we can get an idea of how the game is structured. First of all, there is the **Board package** which gets called upon by the Launcher. In the Board package there is a class "Board" which says how the board is to be set up. Then there is the class "BoardFactory" which actually creates the Board from a grid of cells and populates it with some sprites. The sprites come from the **Sprite package**. The sprites can either be "animated", "empty", "image" or "PacMan" sprites.
The Launcher also uses classes of the **Level package**. In this package the Player ("operator of the game"), the Level (board with players and AI on it) and CollisionMap (map/table of possible collisions) are made. The pellets are also implemented in the Level package. The Pellet and Player Class extend the Ghost class of the **npc package** (npc stands for Non-Player

Characters). The Ghost class in turn extends the classes of Blinky, Clyde, Inky and Pinky which are part of the **ghost package** (inside the npc package).
There is also the Game package which gets called upon by the Launcher and actually creates, starts and stops a game.
Finally, there is also the **UI package** which creates the Interface for the User to play the game.

1.2 Call graph from the most prominent entry point

In our call graph we consider methods up to sixth level. Our proposition allows one to get a quick overview to understand how the calling relationships of the program's methods are organized. The most prominent entry point is in the Launcher class where we find the main method. This method triggers the launch method from where the important parts of the game is based on. On the 3rd level we can see all methods while on following levels we only provide the ones to be useful to display. The makeGame method seems to be the foundation of the game itself. On the one hand the board is created and on the other hand the Pacman game pieces are created. With the start method the game becomes dynamic. For moving the game piece there is a method move which is called via moveTowardsDirection and via getGame. Using common sense, we can assume that the move method will call many other methods just for the moving parts in the game. Other features like start and stop buttons are called by the build method.



Exercise 2 - A Checkers Game – Design

2.1

Steps taken:

1. Identify candidate classes (nouns): Users (=Players), Board, Pieces (=Checkers), Position, Color (R/W), Type (K/P), Move, Validity, Game
2. Requirements/Rules:
   - two players need to be able to input their moves into the same terminal
   - Board must be safed, displayed and updated
   - Checkers can either move or jump. If one of the checkers can jump, it has to
   - Checkers can only move and jump diagonally
   - Checkers can either be red or white
   - Chekcers can either be a pawn or a king
   - Checker gets converted to king when it reaches the king-row (this also ends the players turn)
   - Pawns can only move forward, kings can move forward and backwards
   - Player must be able to enter its move in the format of [current piece position]x[future piece position]
   - Validity of move must be checked
   - Player must move each turn
   - Each turn a player can make a single move, a single jump or a multiple jump
   - Player loses if all pieces got captured or if player is unable to move.
3. CRC-Cards:

Class	model/Coordinate
Responsibility	*Collaboration*
Model Coordinates during a move/knows its coordinates -private int fromX -private int fromY -private int toX -private int toY -private string jumpOrSingle	

Class	model/ Board
Responsibility	*Collaboration*
-Keep track of all Checkers -display the Board -add and remove pieces	Checker

Class	model/Player
Responsibility	*Collaboration*
-is either red or white -keep track of all checkers belonging to player -check if a move still possible	Checker

Class	logic/Game
Responsibility	*Collaboration*
Contains most of the logic: -Calc all possible moves Check if move possible for: -single moves & jumps -Check if crowning possible -check if game finished -get active player	- Board - Player - Checker - Coordinate

Class	model/Checker
Responsibility	*Collaboration*
Contains a list of possible moves for checker -keeps track of coordinates -color -if is king -if is captured	

Class	CheckerGame
Responsibility	*Collaboration*
-Main() -Run game -Set the board -Check if input is valid	- Checker - Game - Board - Player

2.2

CheckerGame

CheckerGame contains the Main() method. and sets up all Model Classes. Here we are checking if the inputs and moves are correct. Which leads us to Game Class. The Responsibilities of the CheckerGame class is to essentially run the game, initially set up the board and to check if the given input of a player is valid. To do all that it directly collaborates with all other classes except the Coordinate class.

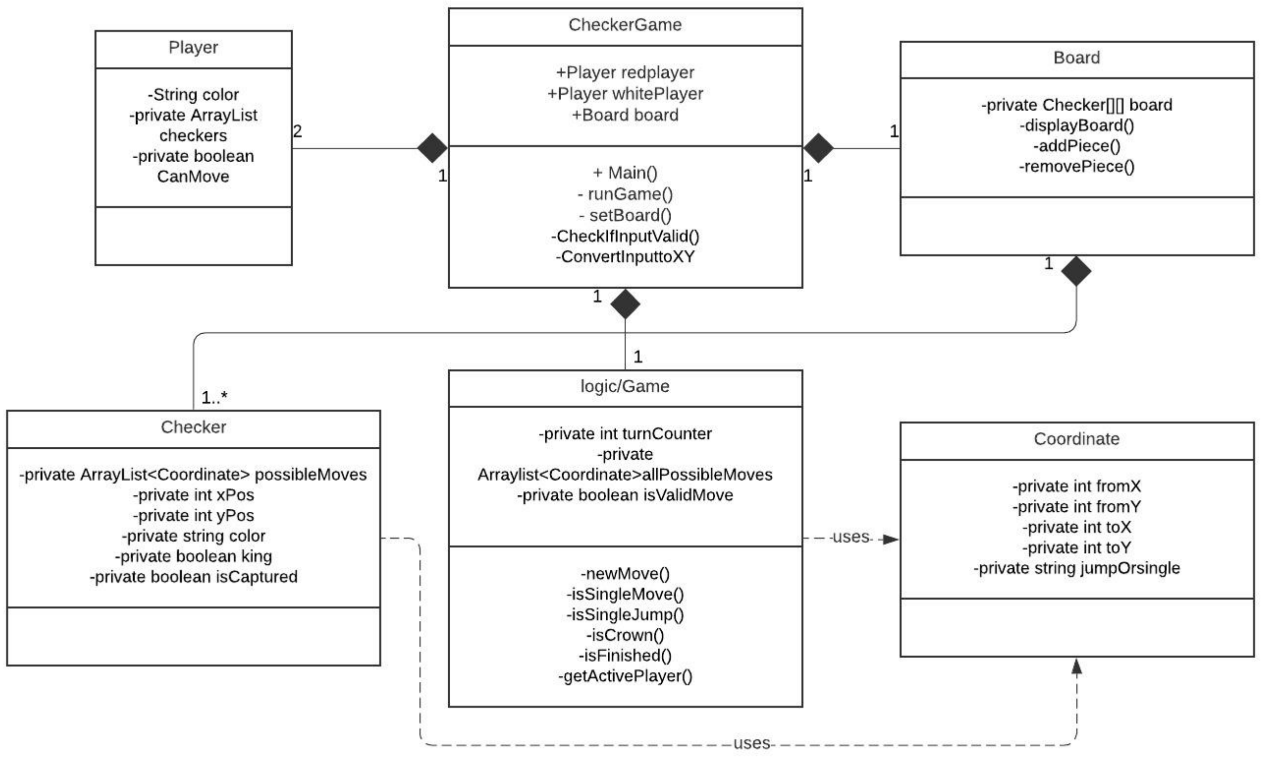
Game

Our Game class could also be regarded as a main class, as the main logic of the game resides there. We are checking every move if that move is valid, if the game is finished etc. The Game class collaborates with all classes of the model package.

2.3

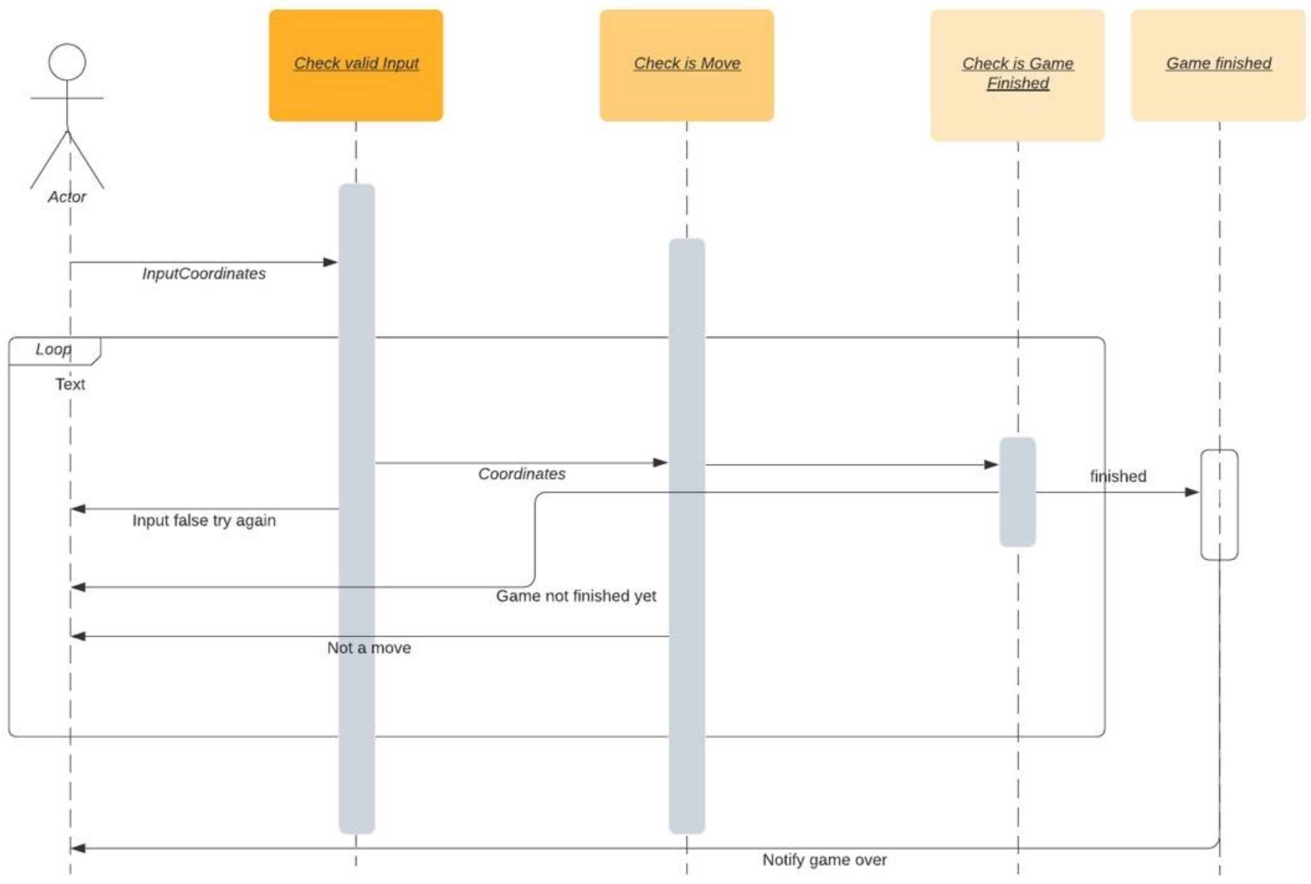
We do not consider our other Model classes to be less important. They just have a function which is lower on the hierarchy of the modelling of the game. We could not remove one of the classes without compromising the whole. What would be possible, is to merge for instance the Player class into the CheckerGame class. However, we do not consider this to be a very smart decision.

2.4



D. Baumgartner, L. Fischer, Y. Higashigaito, R. Kummer

2.5


Check valid Input
Check is Move
Check is Game Finished
Game finished
Actor
InputCoordinates
Loop
Text
Coordinates
finished
Input false try again
Game not finished yet
Not a move
Notify game over