Documentation

Having only little programming experience, I decided to try the basic version of squash for a start. You find the task description here: https://open.hpi.de/courses/java-capstone-1/items/3MyQM463Thg6UX9HxoqXCZ

There is a ball which moves automatically according to the common rules of squash. The paddle can be controlled by the user pressing ‘r’ to move it to the right and ‘l’ for movement to the left. Tip: Keep the keys pressed to make the paddle move fast. If the ball misses the paddle, the program decides at random whether to restart or exit the game.

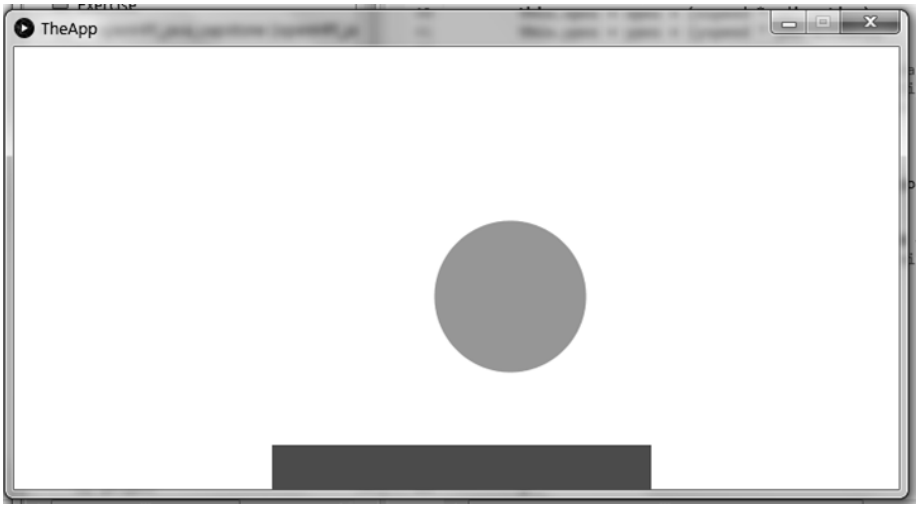

Fig. I Screenshot of the running program

The UML diagram and the Java code are enclosed and, as far as the code is concerned, explained below. For further information about the UML diagram, see the Lab Report.

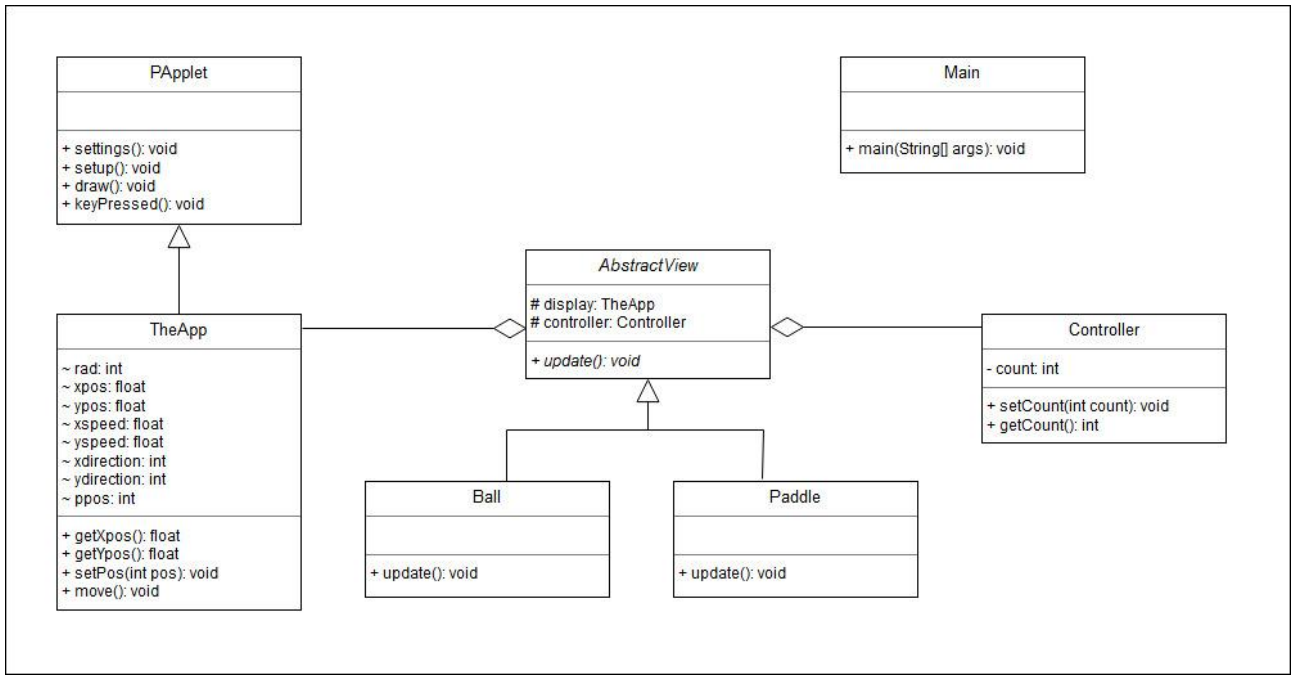

Fig. II UML class diagram

The program starts in class Main.

Main.java

```
1  package de.openhpi.java.capstone.model;
2
3  import processing.core.PApplet;
4
5  public class Main {
6
7      public static void main(String[] args) {
8          PApplet.main(new String[]{TheApp.class.getName()});
9      }
10
11 }
```

Fig. III Main.java

From here, TheApp is executed. It serves the purpose of the model where the data is stored.

TheApp.java

```
1  package de.openhpi.java.capstone.model;
2
3  import processing.core.PApplet;
4  import de.openhpi.java.capstone.view.*;
5  import de.openhpi.java.capstone.controller.*;
6
7  public class TheApp extends PApplet {
8
9      Controller controller = new Controller();
10
11     Ball ball = new Ball(this, controller);
12     Paddle paddle = new Paddle(this, controller);
13
14     final int rad = 60; // Radius of the ball
15     float xpos;     // Starting position of the ball
16     float ypos;
17
18     float xspeed = (float) 2.5; // Speed of the ball
19     float yspeed = (float) 2.5;
20
21     int xdirection = 1; // Left to Right
22     int ydirection = 1; // Top to Bottom
23
24     int ppos;   // Tracks changes in paddle position
25
26     public float getXpos() {
27         return this.xpos;
28     }
29
30     public float getYpos() {
31         return this.ypos;
32     }
33
34     public void setPos(int pos) {
35         this.ppos = pos;
36     }
37
```

```
    public void move() {
        // Update the position of the ball
        this.xpos = xpos + (xspeed * xdirection);
        this.ypos = ypos + (yspeed * ydirection);

        // Test to see if the ball exceeds the boundaries of the screen
        // If it does, reverse its direction by multiplying by -1
        // Test whether it hits or misses the paddle
        if (xpos > width-rad || xpos < rad) {
            this.xdirection *= -1;
        }
        if ((ypos > height-rad-35 && xpos > 200 + ppos && xpos < 500 + ppos) || ypos < rad) {
            this.ydirection *= -1;
        }
        else if (ypos > height-rad-35 && (xpos < 200 + ppos || xpos > 500 + ppos)) {
            double random = Math.random(); // Determine randomly whether to exit or not
            if (random <= 0.5) {
                // Reset the game
                xpos = width/2;
                ypos = height/2;
                xdirection = 1;
                ydirection = 1;
                controller.setCount(0);
            }
            else {
                exit();
            }
        }
    }

    @Override
    public void settings() {
        // Set the screen
        size(700, 350);
    }

    @Override
    public void setup() {
        noStroke();
        frameRate(30);
        ellipseMode(RADIUS);
        // Set the starting position of the ball
        xpos = width/2;
        ypos = height/2;
    }

    @Override
    public void draw() {
        // Draw the screen
        background(255);
        ball.update();
        paddle.update();
    }

    @Override
    public void keyPressed() {
        // User moves paddle to the right
        if (key == 'r') {
            controller.setCount(ppos + 1);
        }
        // User moves paddle to the left
        if (key == 'l') {
            controller.setCount(ppos - 1);
        }
    }

}
```

Fig. IV TheApp.java

The ball and the paddle are derived from the class AbstractView. All the views come with an instance of TheApp and a controller.

AbstractView.java

```
package de.openhpi.java.capstone.view;

import de.openhpi.java.capstone.model.*;
import de.openhpi.java.capstone.controller.*;

public abstract class AbstractView {

    protected TheApp display;
    protected Controller controller;

    public AbstractView(TheApp display, Controller controller) {
        this.display = display;
        this.controller = controller;
    }

    public abstract void update();
}
```

Fig. V AbstractView.java

Ball.java

```java
package de.openhpi.java.capstone.view;

import de.openhpi.java.capstone.model.*;
import de.openhpi.java.capstone.controller.*;

public class Ball extends AbstractView{

    public Ball(TheApp display, Controller controller) {
        super(display, controller);
    }

    @Override
    public void update() {
        // Move the ball
        display.setPos(controller.getCount());
        display.move();

        // Draw the ball
        float xpos = display.getXpos();
        float ypos = display.getYpos();
        display.fill(150);
        display.ellipse(xpos, ypos, 60, 60);
    }

}
```

Fig. VI Ball.java

Paddle.java

```java
package de.openhpi.java.capstone.view;

import de.openhpi.java.capstone.model.*;
import de.openhpi.java.capstone.controller.*;

public class Paddle extends AbstractView{

    public Paddle(TheApp display, Controller controller) {
        super(display, controller);
    }

    @Override
    public void update() {
        // Draw paddle in current position
        display.fill(75);
        display.rect(200 + controller.getCount(), 315, 300, 35);
    }

}
```

Fig. VII Paddle.java

The controller keeps count of the user interaction via keyboard. This information is pulled and used to update model and view.

Controller.java

```java
package de.openhpi.java.capstone.controller;

public class Controller {
    // Changes are pulled
    private int count = 0;

    public void setCount(int count) {
        this.count = count;
    }

    public int getCount() {
        return this.count;
    }

}
```

Fig. VIII Controller.java

Lab Report

Thinking about the task given, I decided to approach it in two steps.

1. First, I would need a general idea of how to translate it into code.
2. Second, I could use the advantages of object-oriented programming in Java to enhance the code.

> Step One

Since the first step did not necessarily require object orientation, I wrote a simple Python program, which allowed me to field-test my ideas in a quick and easy-to-do way.

So, how to program a squash game? I have chosen a (6x6) board which, on every field, contains information about what happens to the ball there, whether it bounces and in which direction. For that purpose, the ball needs an x- and a y-component of both its position and speed. In each step of the iteration, these components are changed according to the information on the board. In addition to that, the user can decide whether to move the paddle or not, for which, in that early version of the game, there are three possible positions (left, middle, right). The game is over when the (only) ball misses the paddle.

In the following, you can see a screenshot of the code and the program as it is running.

```
# -*- coding: utf-8 -*-
import os

# sub-programs

def setBoard(): # set board
    size[0][0] = "xy" # corners
    size[0][5] = "xy"
    size[5][0] = "xy"
    size[5][5] = "xy"
    size[5][1] = " " # paddle (starting position)
    size[5][2] = "y"
    size[5][3] = "y"
    size[5][4] = " "
    for i in range(1,5): # centre of board
        for j in range(1,5):
            size[i][j] = "."
    for i in range(1,5): # left/right wall
        size[i][0] = "x"
        size[i][5] = "x"
    for i in range(1,5): # ceiling
        size[0][i] = "y"

def printBoard(): # print board
    for j in range(5):
        for i in range(6):
            if i == positionX and j == positionY: # show ball
                print "o",
            else:
                print ".",
        print
    for i in range(6):
        if i == positionX and 5 == positionY: # show ball
            print "o",
        elif size[5][i] == "y": # show paddle
            print "_",
        else:
            print ".",
    print
    print
```

```
def move(): # move ball
    if speedX > 0:
        i = min(5,positionX + speedX)
    else:
        i = max(0,positionX + speedX)
    if speedY > 0:
        j = min(5,positionY + speedY)
    else:
        j = max(0,positionY + speedY)
    if size[j][i] == ".": # free movement
        x = speedX
        y = speedY
        return i,j,x,y
    elif size[j][i] == "x": # hits left/right wall
        x = reflectX()
        y = speedY
        i = positionX + x
        return i,j,x,y
    elif size[j][i] == "y": # hits ceiling/paddle
        x = speedX
        y = reflectY()
        j = positionY + y
        return i,j,x,y
    elif size[j][i] == "xy": # hits a corner
        x = reflectX()
        y = reflectY()
        i = positionX + x
        j = positionY + y
        return i,j,x,y
    else:
        print "Game over." # leaves board -> game over
        return -1,-1,0,0

def reflectX(): # reflect ball on left/right wall
    return (-1)*speedX

def reflectY(): # reflect ball on ceiling/paddle
    if speedY > 0:
        return -1
    else:
        return 2 # falling ball
def changePosition(n):
    if n == -1: # left postition
        size[5][1] = "y"
        size[5][2] = "y"
        size[5][3] = " "
        size[5][4] = " "
    elif n == 0: # middle position
        size[5][1] = " "
        size[5][2] = "y"
        size[5][3] = "y"
        size[5][4] = " "
    elif n == 1: # right position
        size[5][1] = " "
        size[5][2] = " "
        size[5][3] = "y"
        size[5][4] = "y"

# main program

speedX = 1
speedY = 2
positionX = 0
positionY = 0

size = range(6) # create board
for i in range(6):
    size[i] = range(6)

setBoard()

while positionX + positionY > -1: # play as long as ball stays on board
    printBoard()
    n = input("Paddle position? Middle (0), left (-1), right(1): ")
    os.system("cls")
    changePosition(n)
    positionX, positionY, speedX, speedY = move()
```

Fig. IX Code of the Python program

Fig. X Screenshot of the running Python program

>> Step Two

With my program running, I was able to venture a step further.

How to improve the code with the help of object-oriented programming? First of all, object orientation allows dividing responsibilities between the classes. For now, only three classes were needed, the board, the ball and the paddle. They are shown in the UML diagram below.

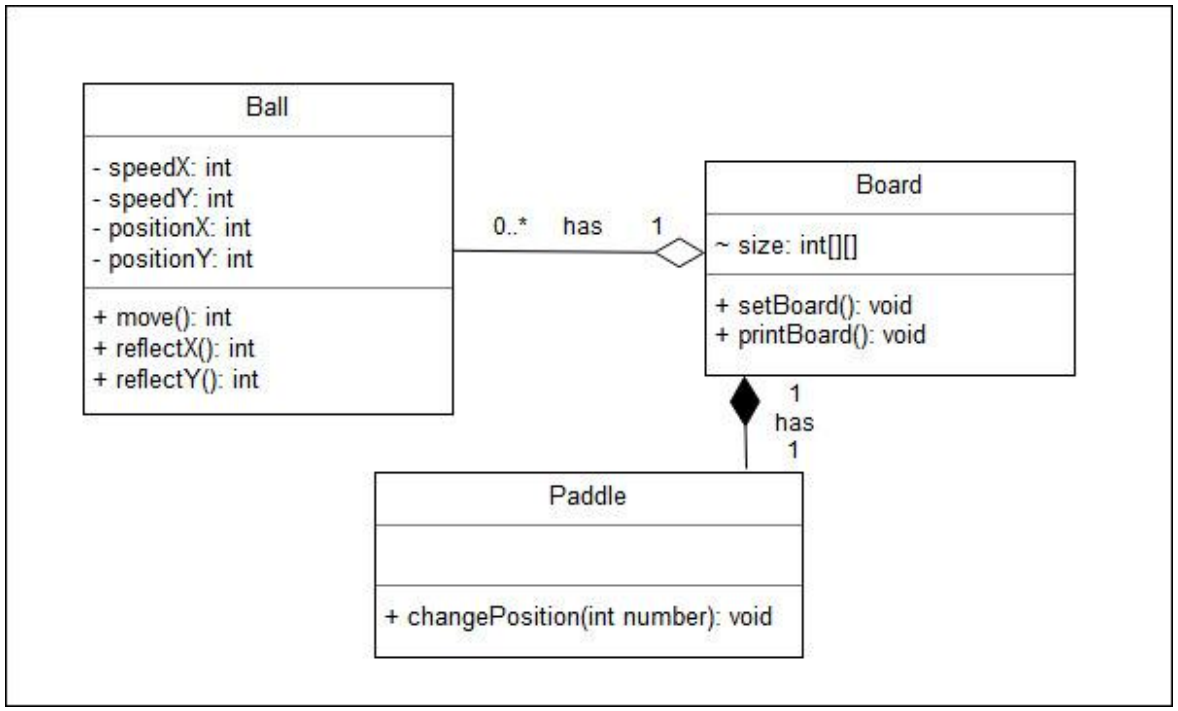

Fig. XI UML diagram

Furthermore, no graphical user interface had yet been included. The Model-View-Controller pattern had not been applied either. Both required some changes in the UML diagram.

Similarities are evident, such as the move() method which appears in both diagrams or the setPos() method which corresponds to the changePosition() method. However, the class structure is now better organised, though more complex.

The board has been replaced by the graphical user interface, and the ball moves smoothly now thanks to animation. The user interaction is more advanced and allows to key-control the paddle. The game is not restricted to one ball either.

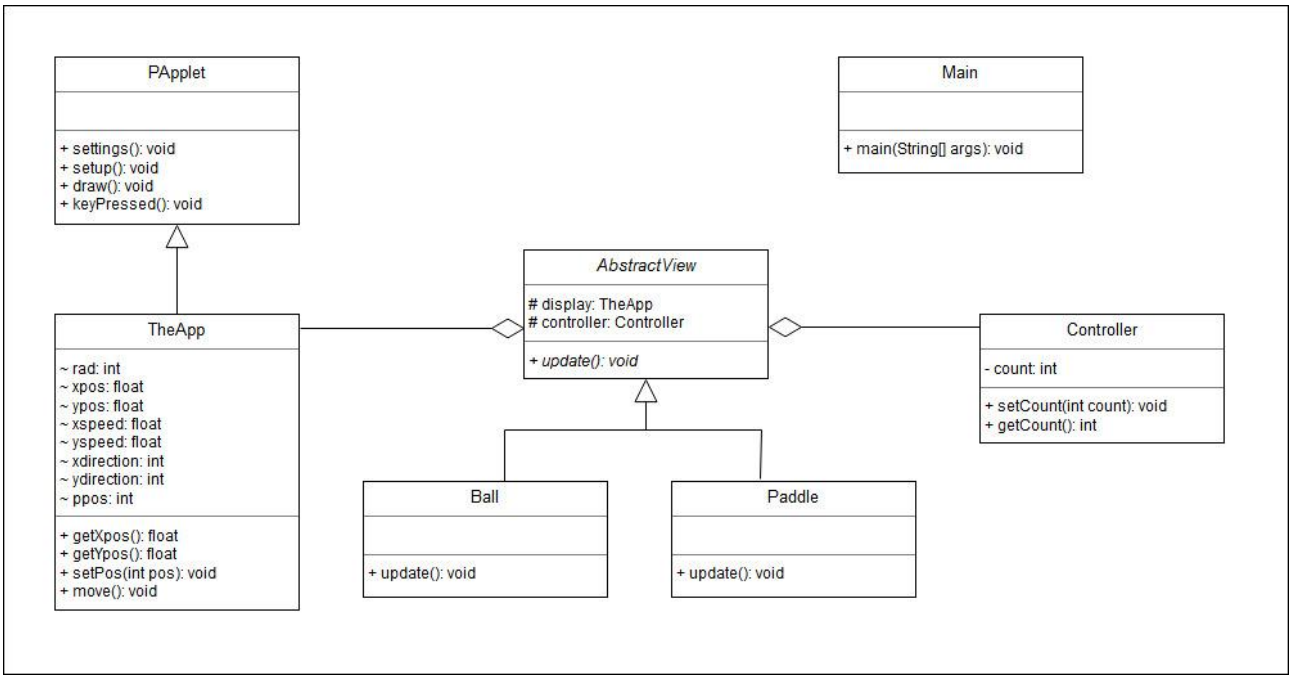

Fig. XII Adapted UML diagram

After these two preparation steps, implementing the Java code was no longer a problem and only little refactoring (which is already mirrored in the UML diagram above) needed to be done afterwards.

>>>

All in all, I am pleased with how the project evolved, though I did not foresee how essentially introducing object orientation, a GUI and the MVC pattern would change the way the program works.

The next time, I will try the more advanced Pong version of the game. In order to do this, I will need a second paddle and another controller which controls the movement of the second paddle independently from that of the first. Thus, I should consider adding a class AbstractController from which the controllers inherit. Probably, further design patterns should be taken into account as well.

Appendix: Learning Journal

peer assessment:
user1 user2
-----submission-------->
self-assessment
(optional)
points <----assessment---
+ review
-----rating-----> bonus points
- double-blind
- rougue results are filtered out automatically
- reported submissions or reviews are checked manually and may be regraded
- team peer assessment: aditional evaluation of your fellow team members
- only active members (who write reviews) receive the points awarded to the group

IDE: integrated development environment
- combines programs which support software development (create, test, fix, optimize, manage)
- basically consists of a code editor and tools for working with the code
- e.g. Eclipse

Eclipse:
1. Create workspace
2. Create project and save in workspace
3. The project folder contains
  - a .settings folder
  - a "bin" folder with the bytecode
  - a "src" folder containing the source code
  - a .classpath file
  - a .project file

export -> archive file
import -> existing project into workspace
debugger: set breakpoints, inspect variables, watch expressions
*: unsaved changes

GUI: graphical user interface
- AWT: Abstract Window Toolkit, outdated, basic libraries still exist
- Swing: GUIs are programmed in Java
- JavaFX: GUIs are programmed in Java or FXML, can be combined with Swing

Processing:
- easy-to-use tool for simple animations
- Eclipse -> Java Build Path -> Add External JARs -> core.jar
- https://processing.org/reference/

git:
- version control, github: online repository
  - history of every change made
    - commit messages: who changed what for what reason
    - allows to go back to previous versions
    - source code backups
  - branching and merging
    - independently work on a copy and merge it into the master branch with the permission of the "owner"
    - conflict resolution
- Git Cheat Sheet

design patterns:
- originate from architecture
- provide a solution to a problem in a context
- e.g. Abstract Factory Pattern
  - creational pattern
  - problem: depending on a given parameter we need either one representation of an object or another
  - solution: dynamically decide on the implementation instead of hard-coding it
- e.g. Composite
  - structural pattern
  - problem: we need to represent part-whole hierarchies
  - solution: compose tree structures which treat individual objects and object composites the same
- e.g. Observer
  - behavioral pattern
  - problem: many objects need to be updated when another object changes its state
  - solution: store reference to dependent objects in list with subject and notify all observers when subjects changes

Model-View-Controller:
- model: stores the data, passes on changes in the data to the view via an observer pattern
- view: presents the data, often uses a composite pattern, informs the controller of user ineraction via the observer pattern
- controller: controls and updates model and view