SEM - ASSIGNMENT 4

Group 42

Exercise 1 - Code improvements

Exercise 1.1 - *if* or *case* statements

Case 1 - PickupFactory

The *PickupFactory* class contained a big *switch* statement:

```
public static PowerUp createPowerUp(int randomNumber) {
      switch (randomNumber) {
            case 0:
                  return new ExtraLife();
            case 1:
                  return new ExtraRope();
            case 2:
                  return new ExtraTime();
            case 3:
                  return new SpeedBoost();
            case 4:
                  return new Shield();
            default:
                  return new ExtraTime();
      }
}
```

This statement is refactored using a static list of pairs of probabilities and classNames:

```
private static List<Pair<Integer, Class>> powerUpProbabilities =
new ArrayList<>();
static {
      powerUpProbabilities.add(new Pair<>(1, ExtraLife.class));
      powerUpProbabilities.add(new Pair<>(2, ExtraHarpoon.class));
      powerUpProbabilities.add(new Pair<>(2, ExtraTime.class));
      powerUpProbabilities.add(new Pair<>(2, SpeedBoost.class));
      powerUpProbabilities.add(new Pair<>(2, Shield.class));
}
```

and:

```
return (AbstractPowerUp)
powerUpProbabilities.get(i).getR().newInstance();
```

Case 2 - Character sprite

The draw method of the *Character* class looked like this:

```
if (direction == 0) {
      getSprite().draw(getPosition());
} else {
      getSprite().draw(getPosition(), direction, 1);
}
```

The method now just inherits from the *AbstractEntity* class. This is done by adding a field "scale" to *AbstractEntity* which determines the scale of the sprite and the collision shape of that entity.

Case 3 - Sprite.setFrames

The setFrames method of the *Sprite* class used to be like this:

```
if (frames > 0) {
      this.frames = frames;
} else {
      this.frames = 1;
}
```

This method is refactored to:

```
this.frames = Math.max(1, frames);
```

This removes an if case and is sufficient.

Case 4 - KeyboardInputManager

The *KeyboardInputManager* has a field "input" which tracks which keys are pressed and which are not. This field used to be a *List*, but is now transformed into a *Set*. This removes the need to check if a key is already pressed.

Case 5 - Character sprites

The *Character* class used to have a switch statement which choose which sprite the character has. This is moved into a separate class containing arrays with the sprites of characters. This class now just returns the nth element of such an array instead of choosing a sprite according to the id in a switch case.

Case 6 - Rectangle setter

In the setter for the size we used two if statements to verify that the size of the rectangle could not be negative. We can replace this check using the Math.max()

```
public void setSize(Vec2d size) {
   if (size.x < 0) {
       size.x = 0;
   }

   if (size.y < 0) {
       size.y = 0;
   }
   this.size = size;
}
```

This could be refactored to:

```
public void setSize(Vec2d size) {
   size.x = Math.max(size.x, 0);
   size.y = Math.max(size.y, 0);

   this.size = size;
}
```

Case 7 - EntityFactory

We could not find a way to refactor this particular case statement, because we need a way to compare the strings in the JSON file to the name of the entity.

```
public static AbstractEntity createEntity(JSONObject entity) {
   String type = entity.getString("type");
   double x = entity.getDouble("x");
   double y = entity.getDouble("y");
   Vec2d position = new Vec2d(x, y);

   switch (type) {
       case "Player":
           return createCharacter(position);
       case "Ball":
           return createBall(position,
entity.getJSONObject("attributes"));
       case "Wall":
           return createWall(position);
       case "Floor":
           return createFloor(position);
```

```
        default:
            return null;
    }
```

Other cases

We could not find other cases with if or switch statements that could be refactored, because the these are mostly single if statements like `if(player.isAlive)` .

Exercise 2 - Teaming up

Exercise 2.1 - new game features

For the new feature, we choose two things that work together very well. Sound effects and a settings menu. Sound effects can be separated in background music and sound effects. The settings control the sound and music volume and other settings.

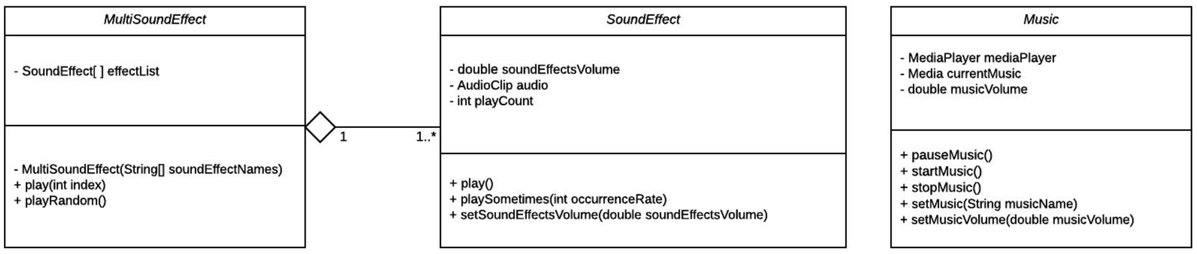
Requirements for both the sounds and the settings view can be found at:

https://github.com/MatthijsKok/TI2206/wiki/Requirement-documents#setting-menu

https://github.com/MatthijsKok/TI2206/wiki/Requirement-documents#sound-effects

Exercise 2.2 - responsibility driven design and UML

<u>Sound effects</u>


MultiSoundEffect
- SoundEffect[ ] effectList
- MultiSoundEffect(String[] soundEffectNames)
+ play(int index)
+ playRandom()
1
1..*
SoundEffect
- double soundEffectsVolume
- AudioClip audio
- int playCount
+ play()
+ playSometimes(int occurrenceRate)
+ setSoundEffectsVolume(double soundEffectsVolume)
Music
- MediaPlayer mediaPlayer
- Media currentMusic
- double musicVolume
+ pauseMusic()
+ startMusic()
+ stopMusic()
+ setMusic(String musicName)
+ setMusicVolume(double musicVolume)