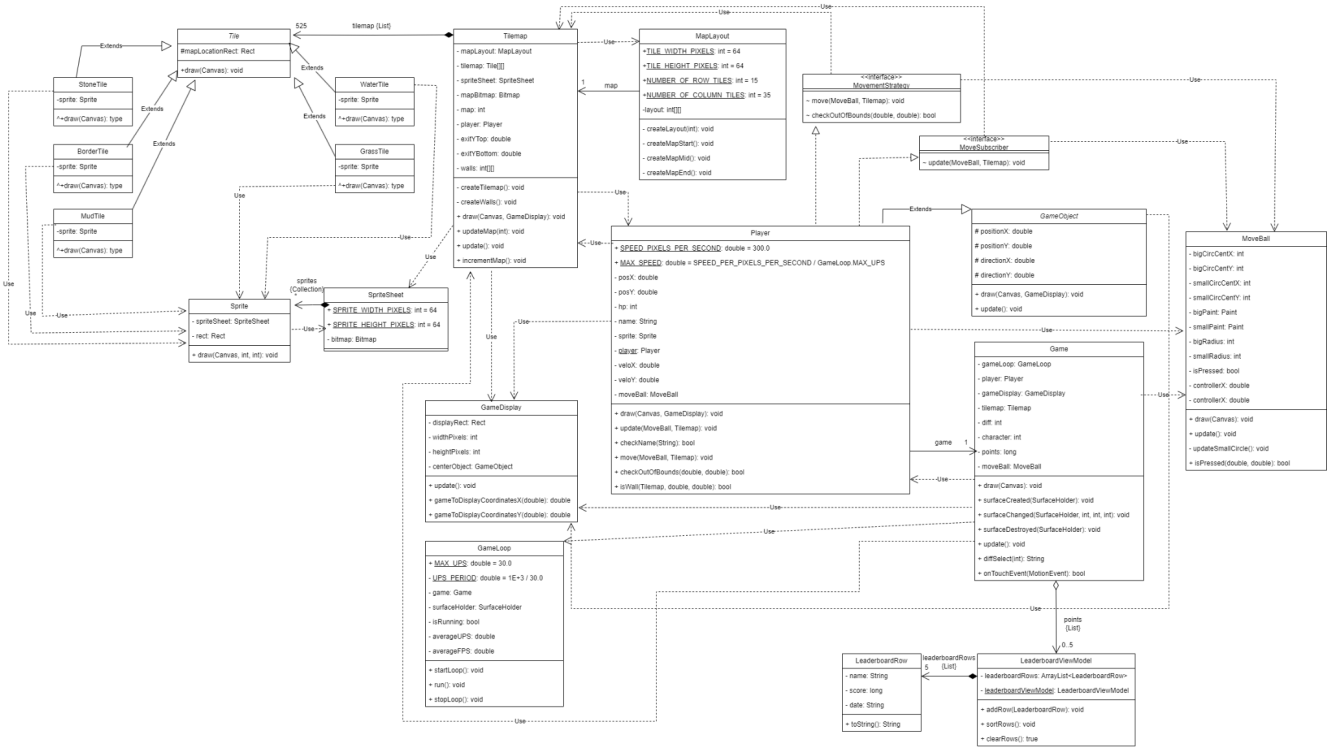
cs2340 team 51

sprint 3 design deliverables

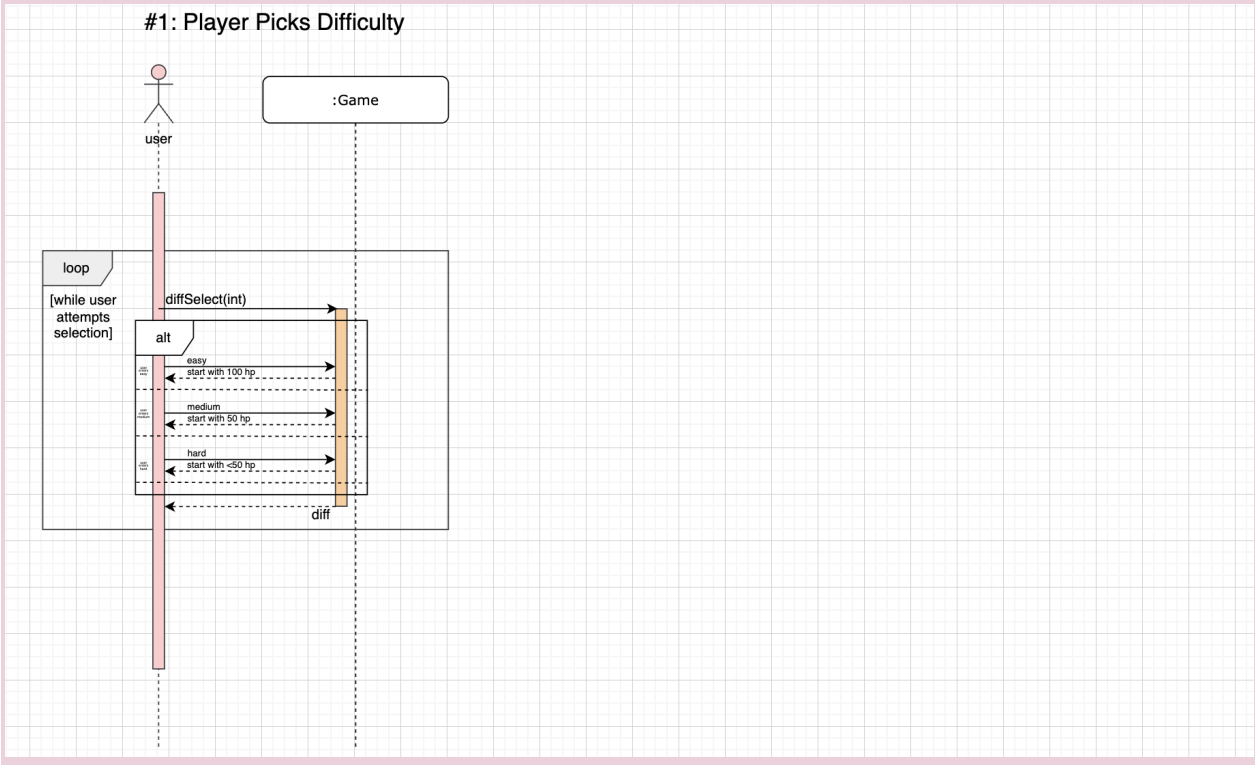
Design Class Diagram



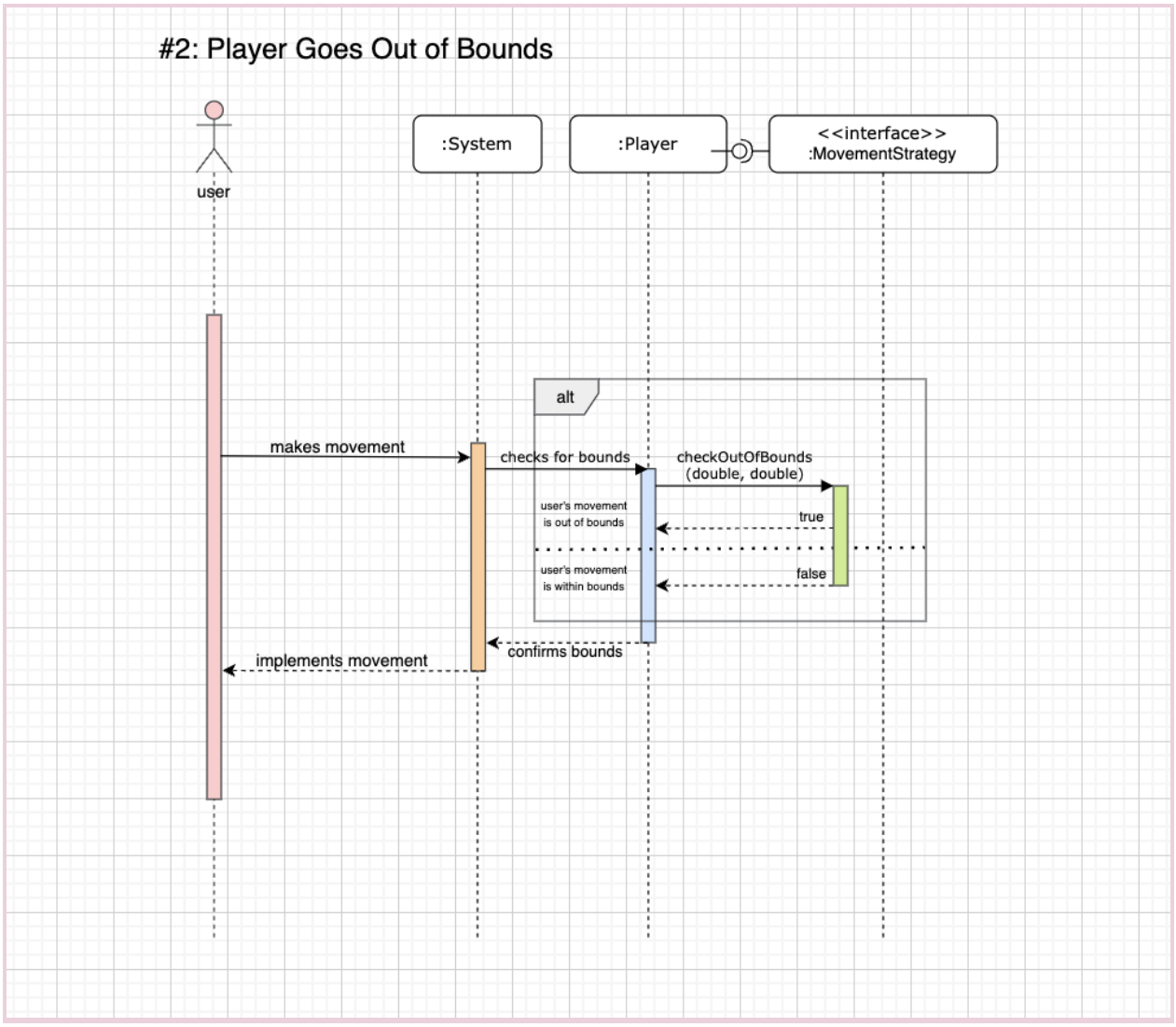
Link to bigger image

Sequence Diagrams

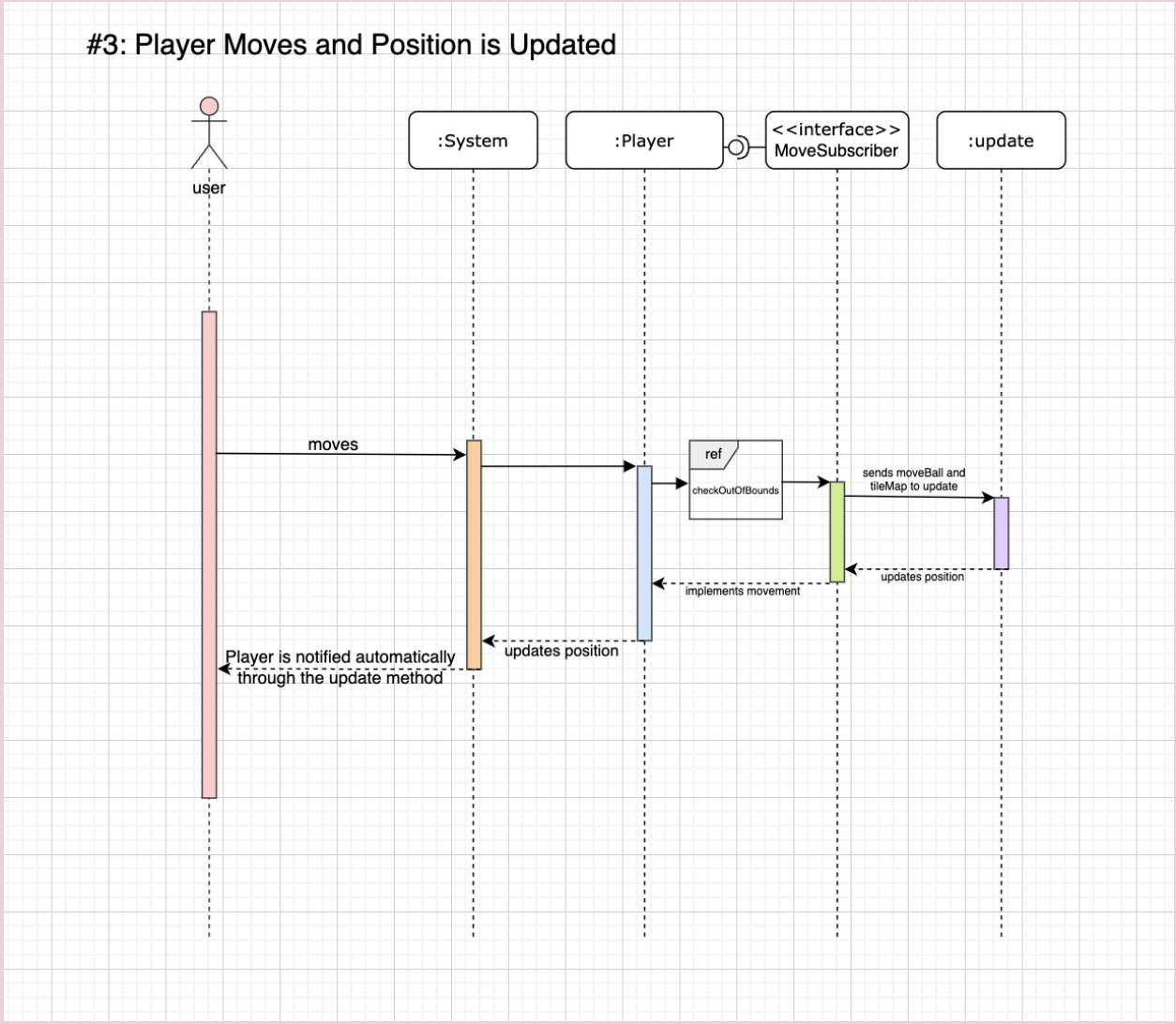
1. Player picks the difficulty level in the game



2. *Player's movements lead to the player going out of the bounds of the game*



3. Player makes a movement and is notified as their position is updated



Design Pattern Evidence

1. Strategy Pattern

```
package com.example.team_51.model;

1 usage  1 implementation  Caljorb
public interface MovementStrategy {
    1 usage  1 implementation  Caljorb
    abstract void move(MoveBall moveBall);

    1 usage  1 implementation  Caljorb
    abstract boolean checkOutOfBounds(double posX, double posY);
}
```

```
25 usages  Caljorb +1
public class Player extends Circle implements MovementStrategy
```

```
1 usage  Caljorb
@Override
public void move(MoveBall moveBall) {
    veloX = moveBall.getControllerX() * MAX_SPEED; // moveBall.getController is always 0
    veloY = moveBall.getControllerY() * MAX_SPEED;

    System.out.println(veloX); // velocity not updated after first retry
    System.out.println(veloY);

    double tempX = posX + veloX;
    double tempY = posY + veloY;

    if (!checkOutOfBounds(tempX, tempY)) {
        posX = tempX;
        posY = tempY;
    }
```

```
1 usage  Caljorb
@Override
public boolean checkOutOfBounds(double posX, double posY) {
    boolean xIn = posX > 1110 && posX < 3300;
    boolean yIn = posY > 500 && posY < 1400;

    return !(xIn && yIn);
}
```

2. Observer Pattern

```
1 usage  1 implementation  Caljorb
public interface MoveSubscriber {
    1 implementation  Caljorb
    abstract void update(MoveBall moveBall, Tilemap tilemap);
```

```
25 usages  Caljorb +1
public class Player extends Circle implements MovementStrategy, MoveSubscriber {
```

```
Caljorb
@Override
public void update(MoveBall moveBall, Tilemap tilemap) {
    move(moveBall, tilemap);
}
```

3. Strategy Pattern Design Paragraph

The strategy class I created was called MovementStrategy. It has two abstract methods that classes using it will have to implement. Those methods are move and checkOutOfBounds. These methods allow for Player to be dynamically updated, and in the future when implementing enemies, we can use this MovementStrategy interface to create enemy movement. Because MovementStrategy is an interface, this allows us to implement the methods uniquely in each

class that implements the interface, so while each class that does this has the same behavior, the way in which we achieve this is slightly different. In player, the move method lets the moveBall be used to update the player's position, while the checkOutOfBounds ensures that this position will not result in the player going out of the screen. In the future, it's possible that we include more Strategy pattern designs, such as an AttackStrategy class. This would allow for an easier way to dynamically update things such as player health and enemy health.

4. Observer Pattern Design Paragraph
    a. The observer class I created was called MoveSubscriber. For this sprint, we only have one abstract method to be implemented, called update. Player implements MoveSubscriber, so it uses this update method in order to update the player's position. Because there are no enemies yet, Player is the only class that implements MoveSubscriber as it is the only asset that moves. The reason this fulfills the Observer pattern is because MoveSubscriber has its dependents (Player in this case), and whenever there is a state change, Player is notified automatically through the update method. In the future enemies will also implement this interface so they can also move.