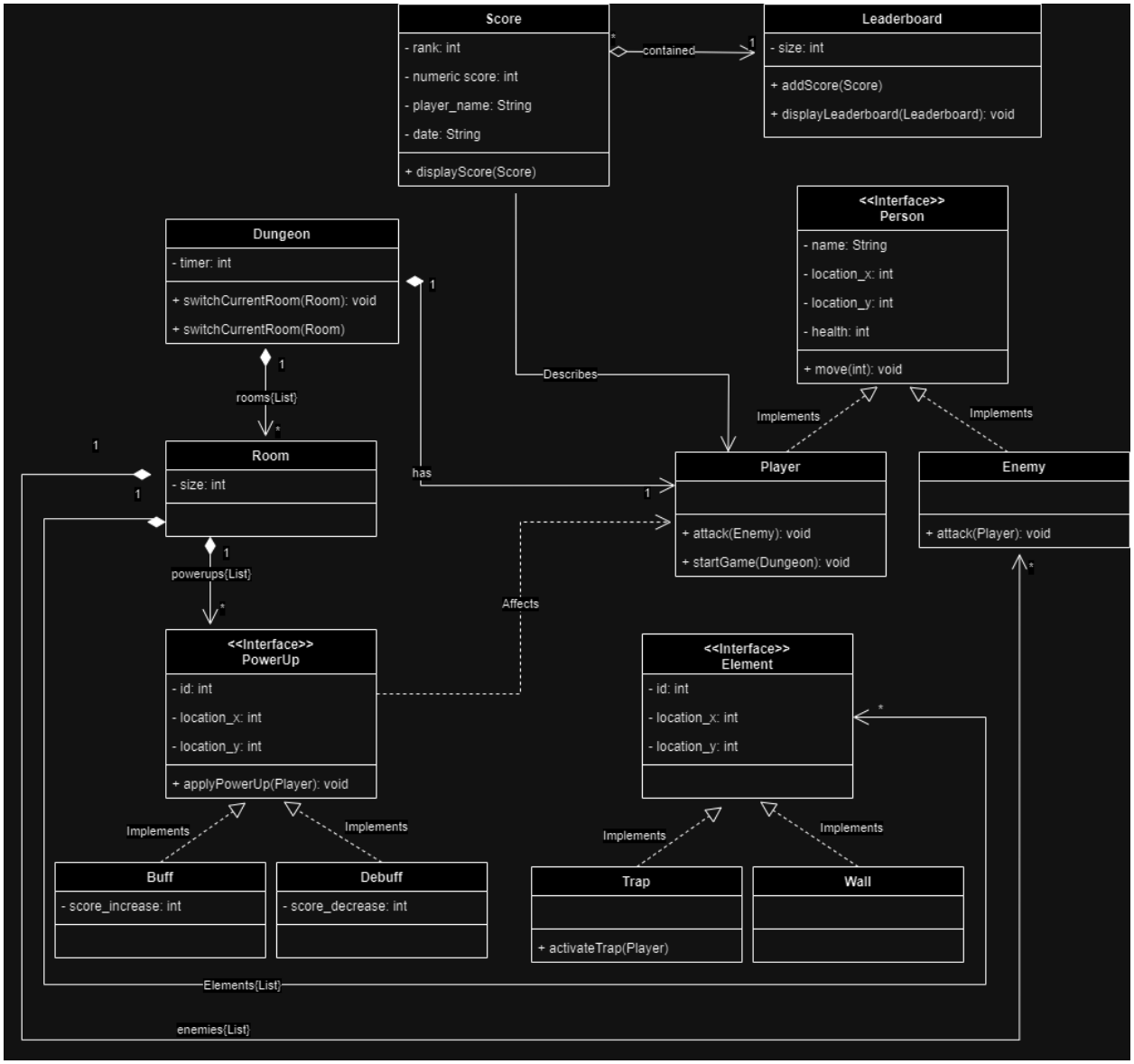
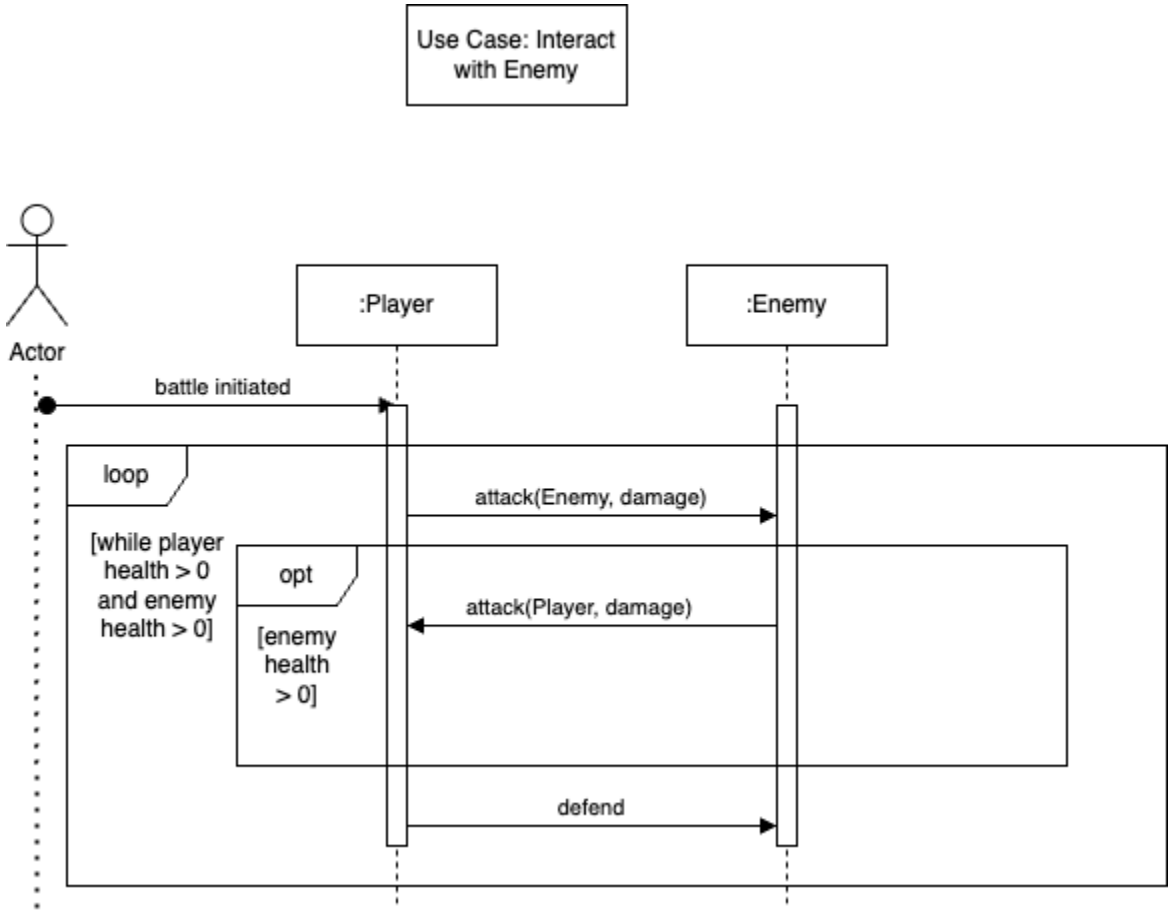
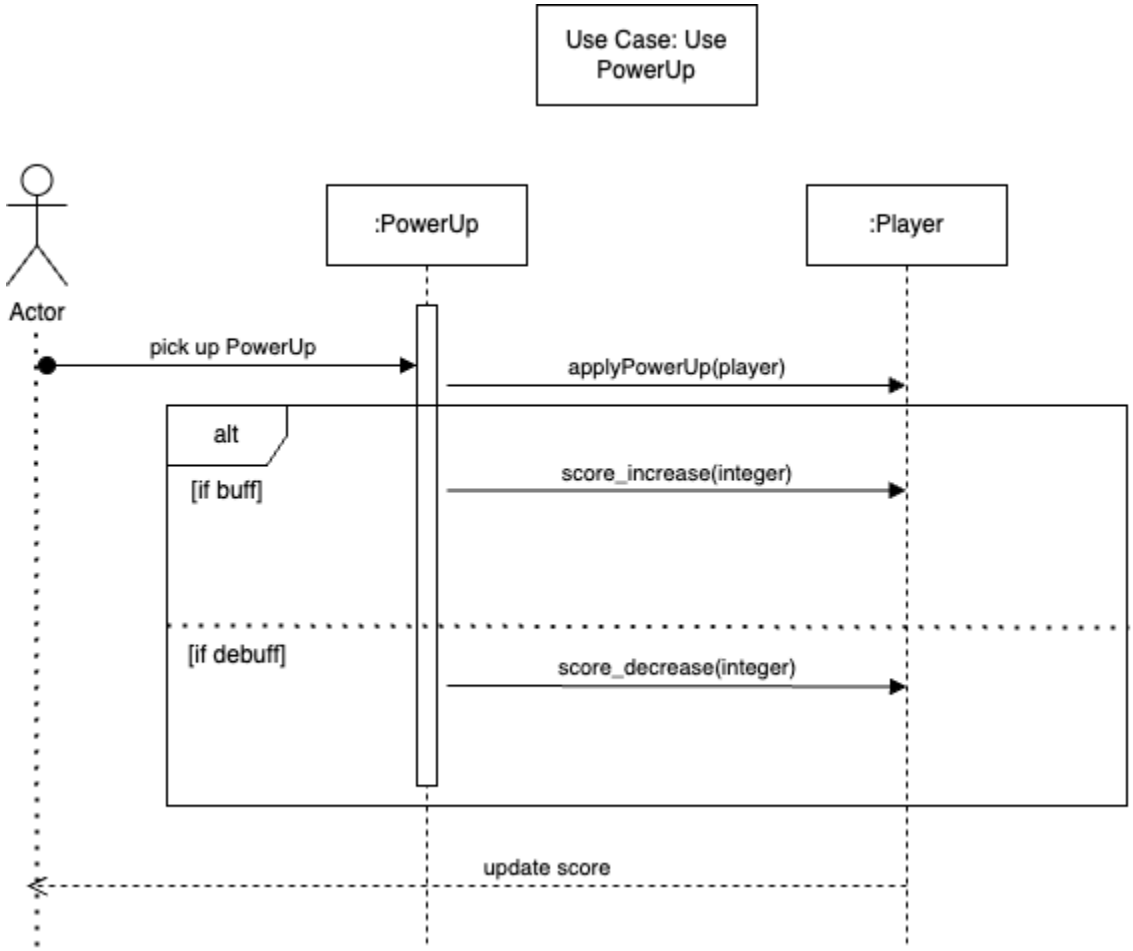
DCD Diagram:


Score
- rank: int
- numeric score: int
- player_name: String
- date: String
+ displayScore(Score)
contained
Leaderboard
- size: int
+ addScore(Score)
+ displayLeaderboard(Leaderboard): void
<<Interface>>
Person
- name: String
- location_x: int
- location_y: int
- health: int
+ move(int): void
Dungeon
- timer: int
+ switchCurrentRoom(Room): void
+ switchCurrentRoom(Room)
rooms{List}
Describes
Implements
Implements
Room
- size: int
has
Player
+ attack(Enemy): void
+ startGame(Dungeon): void
Enemy
+ attack(Player): void
powerups{List}
Affects
<<Interface>>
PowerUp
- id: int
- location_x: int
- location_y: int
+ applyPowerUp(Player): void
<<Interface>>
Element
- id: int
- location_x: int
- location_y: int
Implements
Implements
Implements
Implements
Buff
- score_increase: int
Debuff
- score_decrease: int
Trap
+ activateTrap(Player)
Wall
Elements{List}
enemies{List}

Use Case Diagram 1:

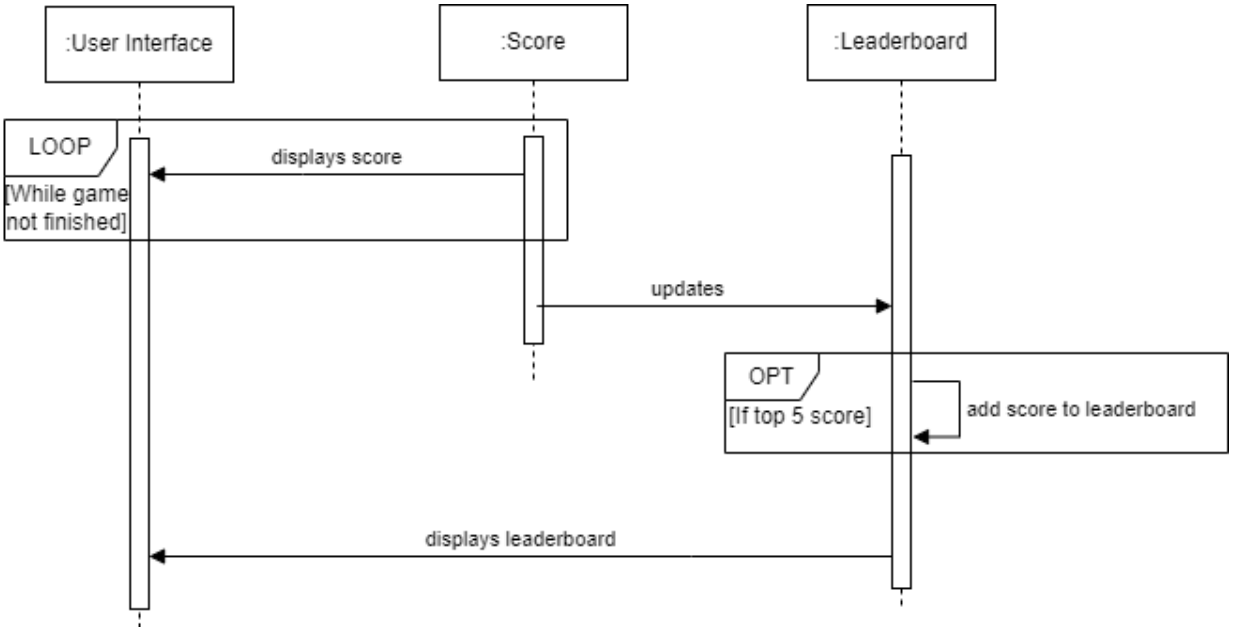


Use Case Diagram 2:

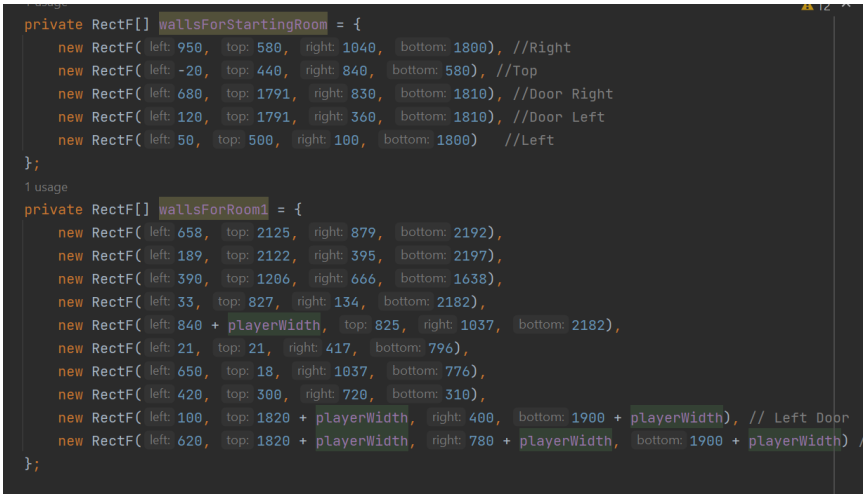


Use Case Diagram 3:

**Use Case: Leaderboard Sequence**



## Strategy Pattern



```
private RectF[] wallsForStartingRoom = {
    new RectF( left: 950,  top: 580,  right: 1040,  bottom: 1800), //Right
    new RectF( left: -20,  top: 440,  right: 840,  bottom: 580), //Top
    new RectF( left: 680,  top: 1791,  right: 830,  bottom: 1810), //Door Right
    new RectF( left: 120,  top: 1791,  right: 360,  bottom: 1810), //Door Left
    new RectF( left: 50,  top: 500,  right: 100,  bottom: 1800)   //Left
};
1 usage
private RectF[] wallsForRoom1 = {
    new RectF( left: 658,  top: 2125,  right: 879,  bottom: 2192),
    new RectF( left: 189,  top: 2122,  right: 395,  bottom: 2197),
    new RectF( left: 390,  top: 1206,  right: 666,  bottom: 1638),
    new RectF( left: 33,  top: 827,  right: 134,  bottom: 2182),
    new RectF( left: 840 + playerWidth,  top: 825,  right: 1037,  bottom: 2182),
    new RectF( left: 21,  top: 21,  right: 417,  bottom: 796),
    new RectF( left: 650,  top: 18,  right: 1037,  bottom: 776),
    new RectF( left: 420,  top: 300,  right: 720,  bottom: 310),
    new RectF( left: 100,  top: 1820 + playerWidth,  right: 400,  bottom: 1900 + playerWidth), // Left Door
    new RectF( left: 620,  top: 1820 + playerWidth,  right: 780 + playerWidth,  bottom: 1900 + playerWidth)
};
```

```
switch (roomNum) {
//Log.d("PlayerViewModel", "The room num is " + roomNum);
case 0:
    wallsForCurrentRoom = wallsForStartingRoom;
    break;
case 1:
    wallsForCurrentRoom = wallsForRoom1;
    break;
case 2:
    wallsForCurrentRoom = wallsForRoom2;
    break;
default:
    return false;
}
```

We followed the Strategy Pattern by allowing for abstraction with the walls for the room. What this means is that we essentially had an area where all of the walls are defined. Then, we can choose which walls will go to which room depending on where the players are. Essentially, we had different versions for the walls for different rooms.

**Observer Pattern**

```
if (wallsForCurrentRoom != null) {
    for (RectF wall : wallsForCurrentRoom) {
        if (RectF.intersects(newPlayerRect, wall)) {
            return false;
        }
    }
}
```

In the program for the player

Our code properly utilizes the Observer pattern for movement by using the character and its interactions with walls. Essentially what our program does is have the wall rectangles act as a publisher which broadcasts its location. The walls notifies the player about its locations. When the player intersects a wall, it will be notified. This will then make the wall not allow the player to keep moving in that direction.