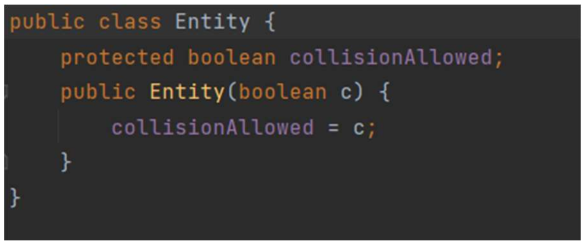
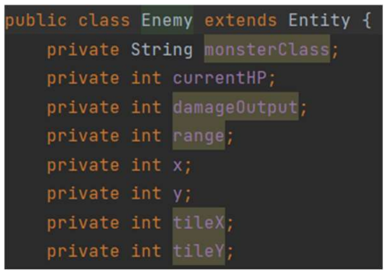
SOLID

Open Close Principle



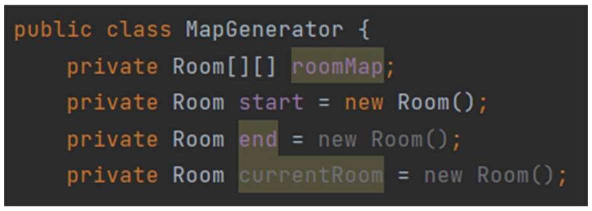
```java
public class Entity {
    protected boolean collisionAllowed;
    public Entity(boolean c) {
        collisionAllowed = c;
    }
}
```



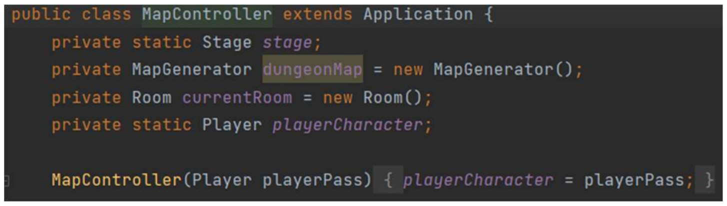
```java
public class Enemy extends Entity {
    private String monsterClass;
    private int currentHP;
    private int damageOutput;
    private int range;
    private int x;
    private int y;
    private int tileX;
    private int tileY;
```

This example came from our source code to introduce entities into our rooms. We will extend the Entity class to create classes that will have a collision relationship with the player. We add functionalities by writing it into the new subclass. As seen in the images, the Enemy class will utilize the Entity principle, but it will also modify it to include new attributes such as health, damage, location, etc. This is a good example of the Open Close Principle since the Entity class is being extended, but it is not being modified.

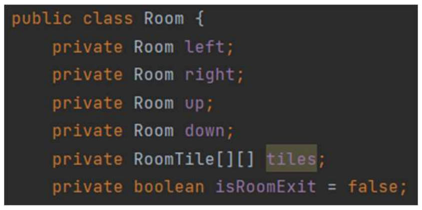
Single Responsibility Principle



```java
public class MapGenerator {
    private Room[][] roomMap;
    private Room start = new Room();
    private Room end = new Room();
    private Room currentRoom = new Room();
```



```java
public class MapController extends Application {
    private static Stage stage;
    private MapGenerator dungeonMap = new MapGenerator();
    private Room currentRoom = new Room();
    private static Player playerCharacter;

    MapController(Player playerPass) { playerCharacter = playerPass; }
```

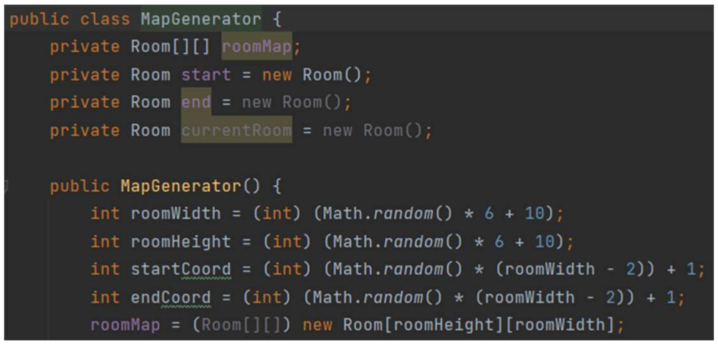


```java
public class Room {
    private Room left;
    private Room right;
    private Room up;
    private Room down;
    private RoomTile[][] tiles;
    private boolean isRoomExit = false;
```

The map is based on different types of rooms. For the player to have the ability to move, search, and attack, many different aspects need to be created and implemented. Due to this fact, we created many classes that will be responsible for one single aspect. For example, MapGenerator was made for the sole purpose of generating a map with rooms, based on the Room class, while the MapController is used to set the player's movement and actions in the map. The Single Responsibility Principle supports this motion. We built our game with numerous small classes with specific names to be responsible for one single responsibility.

GRASP

Creator



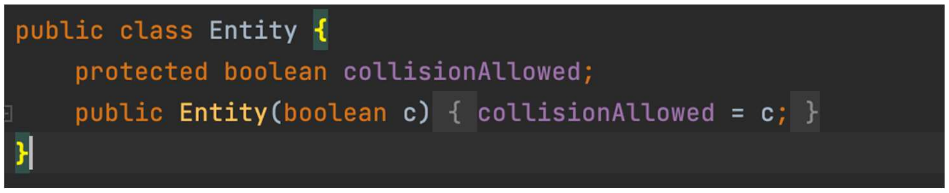
```java
public class MapGenerator {
    private Room[][] roomMap;
    private Room start = new Room();
    private Room end = new Room();
    private Room currentRoom = new Room();

    public MapGenerator() {
        int roomWidth = (int) (Math.random() * 6 + 10);
        int roomHeight = (int) (Math.random() * 6 + 10);
        int startCoord = (int) (Math.random() * (roomWidth - 2)) + 1;
        int endCoord = (int) (Math.random() * (roomWidth - 2)) + 1;
        roomMap = (Room[][]) new Room[roomHeight][roomWidth];
```

The creator principle discusses who should be responsible for creating an instance of an objects. It states that the object should:

- Contains or aggregates the new object
- Records the new object
- Closely uses the new object.

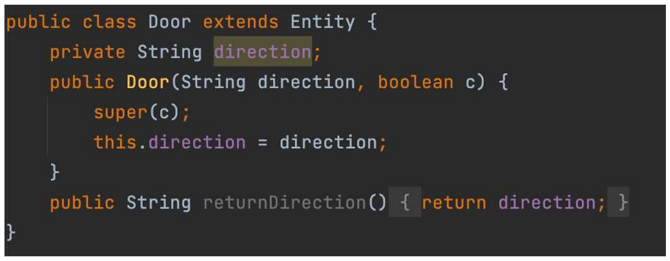
With this definition, the MapGenerator class should be a good example of the Creator principle. The MapGenerator must create new Room objects and record them to allow certain actions in the game such as creating the exit room and walking through different rooms.

Polymorphism



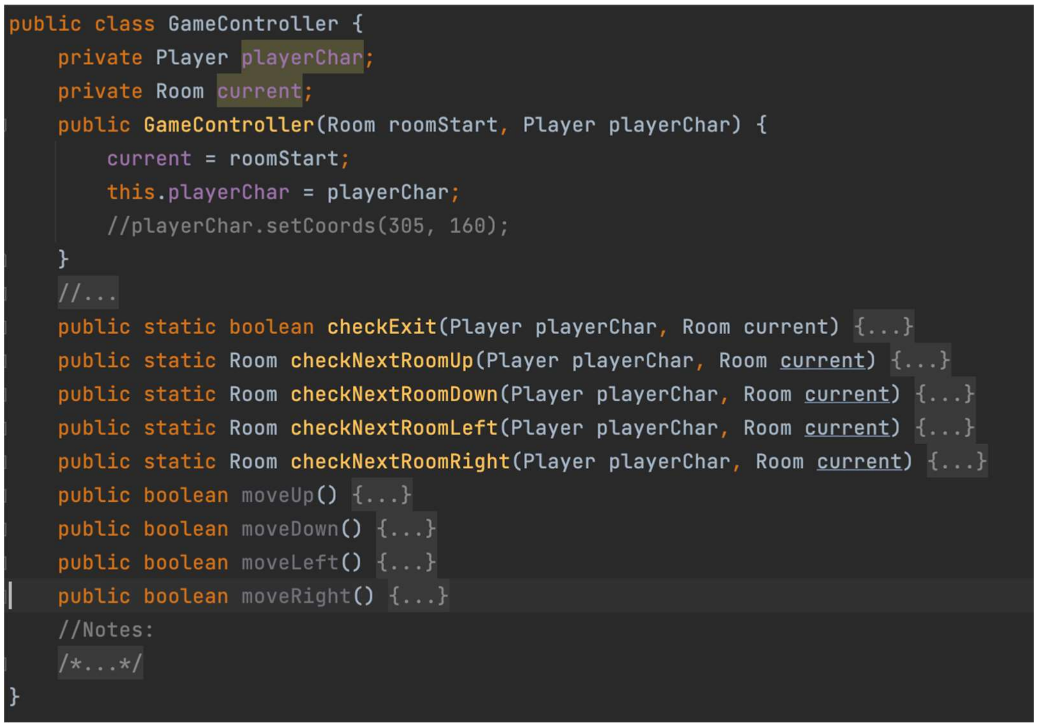
```java
public class Entity {
    protected boolean collisionAllowed;
    public Entity(boolean c) { collisionAllowed = c; }
}
```

In our source code we use Entity class to define different objects in our game such as Door, Wall, Player, Monster. Each object extends the entity class and inherits the properties / characteristics of the entity and extends it with its own set of functions and features. This is a perfect example of polymorphism (feature that allows us to perform a single action in different ways).



```java
public class Door extends Entity {
    private String direction;
    public Door(String direction, boolean c) {
        super(c);
        this.direction = direction;
    }
    public String returnDirection() { return direction; }
}
```

Controller

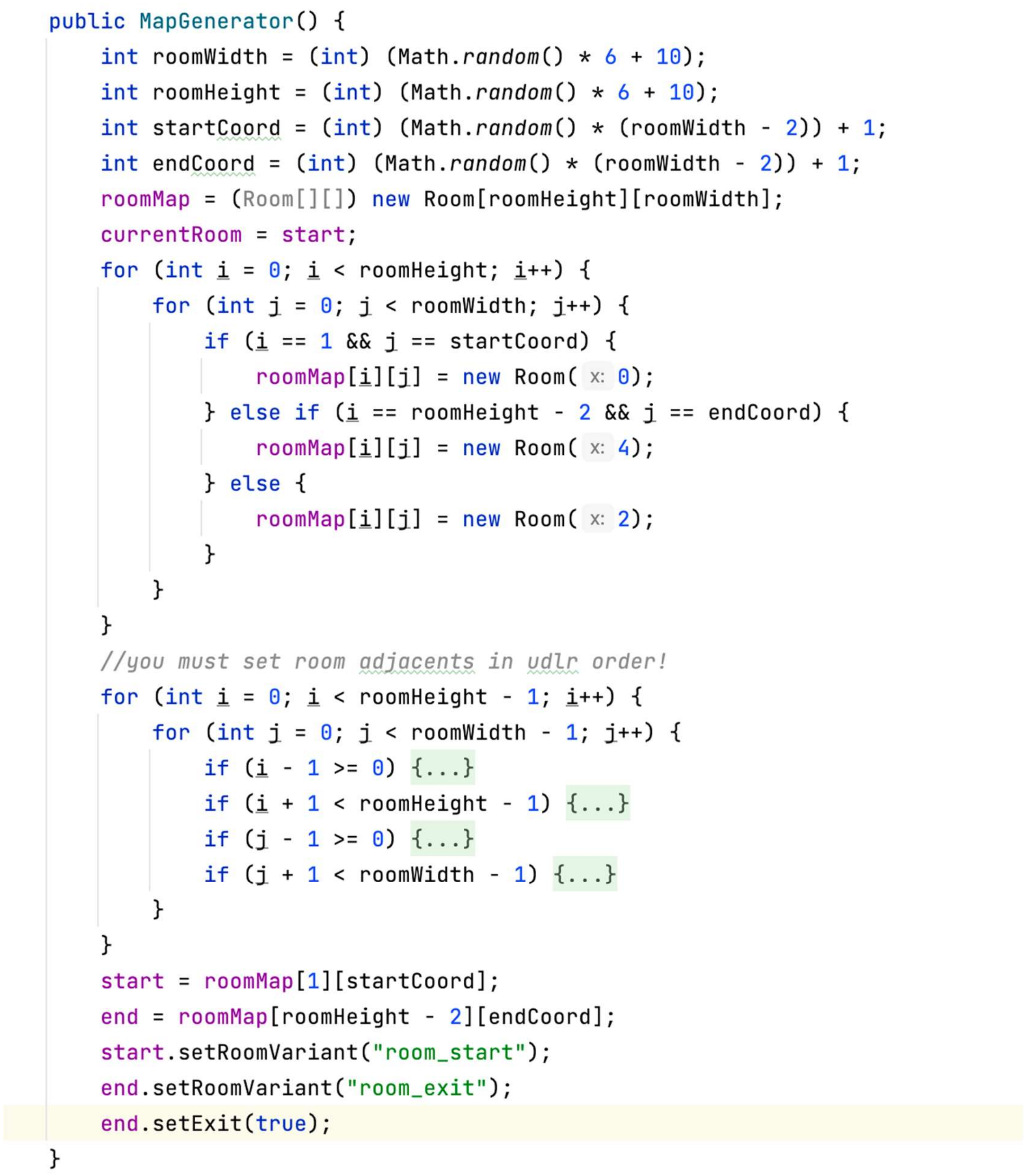


```
public class GameController {
    private Player playerChar;
    private Room current;
    public GameController(Room roomStart, Player playerChar) {
        current = roomStart;
        this.playerChar = playerChar;
        //playerChar.setCoords(305, 160);
    }
    //...
    public static boolean checkExit(Player playerChar, Room current) {...}
    public static Room checkNextRoomUp(Player playerChar, Room current) {...}
    public static Room checkNextRoomDown(Player playerChar, Room current) {...}
    public static Room checkNextRoomLeft(Player playerChar, Room current) {...}
    public static Room checkNextRoomRight(Player playerChar, Room current) {...}
    public boolean moveUp() {...}
    public boolean moveDown() {...}
    public boolean moveLeft() {...}
    public boolean moveRight() {...}
    //Notes:
    /*...*/
}
```

The GameController from our source code is an example of GRASP Controller. It handles the keyboard inputs from the user and performs the tasks according to different use case models. This class works as a director commanding other object. It does not hold information about the different objects, but it rather explains what will happen when the user pushes certain keys.

Code Smells

Bloaters - Long Method



```
public MapGenerator() {
    int roomWidth = (int) (Math.random() * 6 + 10);
    int roomHeight = (int) (Math.random() * 6 + 10);
    int startCoord = (int) (Math.random() * (roomWidth - 2)) + 1;
    int endCoord = (int) (Math.random() * (roomWidth - 2)) + 1;
    roomMap = (Room[][]) new Room[roomHeight][roomWidth];
    currentRoom = start;
    for (int i = 0; i < roomHeight; i++) {
        for (int j = 0; j < roomWidth; j++) {
            if (i == 1 && j == startCoord) {
                roomMap[i][j] = new Room( x: 0);
            } else if (i == roomHeight - 2 && j == endCoord) {
                roomMap[i][j] = new Room( x: 4);
            } else {
                roomMap[i][j] = new Room( x: 2);
            }
        }
    }
    //you must set room adjacents in udlr order!
    for (int i = 0; i < roomHeight - 1; i++) {
        for (int j = 0; j < roomWidth - 1; j++) {
            if (i - 1 >= 0) {...}
            if (i + 1 < roomHeight - 1) {...}
            if (j - 1 >= 0) {...}
            if (j + 1 < roomWidth - 1) {...}
        }
    }
    start = roomMap[1][startCoord];
    end = roomMap[roomHeight - 2][endCoord];
    start.setRoomVariant("room_start");
    end.setRoomVariant("room_exit");
    end.setExit(true);
}
```

MapGenerator is a method inside the public class Map that's responsible for generating a game map as the name suggests. Although necessary and vital for the game this method is bloated with tons of lines of code and if conditionals. A possible solution is to create a method to hold all the for loops.