Homework 5: Adventure Game – Object Oriented Approach

Description:

In the previous assignment, you learned how to use multiple arrays to model a dungeon map. In this assignment we will use an object-oriented approach to implement your dungeon. Objects make your code easier to understand, easier to maintain and easier to build upon. Best of all, an object-oriented approach in designing software complements the iterative design principles we've used throughout the semester. We can design and implement one class at a time, test it, and then reliably use it to buildup the next portion of our application.

Required Concepts:

Modeling with objects
Instance Variables
Getters and Setters
Constructors
Instance methods
Composition (has-a)

Intro to Object Oriented Programming:

If you are familiar with the basic ideas behind object oriented programming then skip this section.

The basic concept that governs object-oriented design is the realization that your target application can be expressed as a sum of smaller, well defined parts. The intention of Object Oriented Programming (OOP) is to define a process of developing software that is synonymous to the process of crafting real-world objects. We'll walk through the basis for OOP using a real world example; let's assume you wanted to construct a Lego model of an airplane.

Lego model of biplane

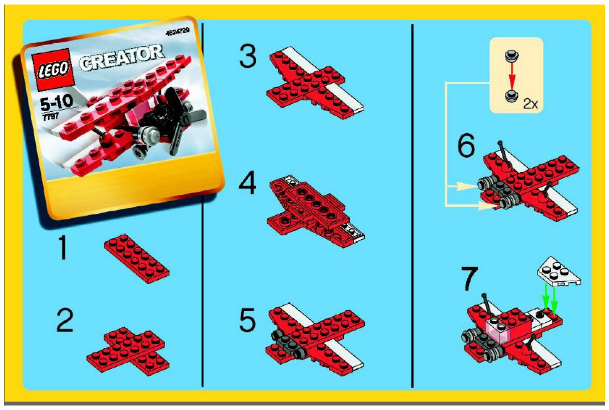
The completed airplane model is analogous to a modeled software object. It represents the end goal of what we'd like to achieve. In real life, we accomplish this by constructing the model airplane from smaller pieces, Legos. In this example, we assume that a single Lego block represents a fundamental type. Notice some parts of the plane may require more complex types, like the propeller, which is an object itself. Before we can build the plane, we would need to have all of its necessary pieces.

Once the plane is created it can then be used as part of something even larger, like part of an airport. This is akin to how we use objects to build up an application.

Parts list for Lego biplane similar to instance variables

In OOP, we create a class. The class is responsible for defining the object. In our plane example, the class would contain the Legos *parts list* and the *instructions* on how to put it together.

The *parts list* represents the properties that the plane needs to have to create a plane. In OOP, the *parts list* represents our **instance variables.**



Instructions for assembling biplane is like constructor

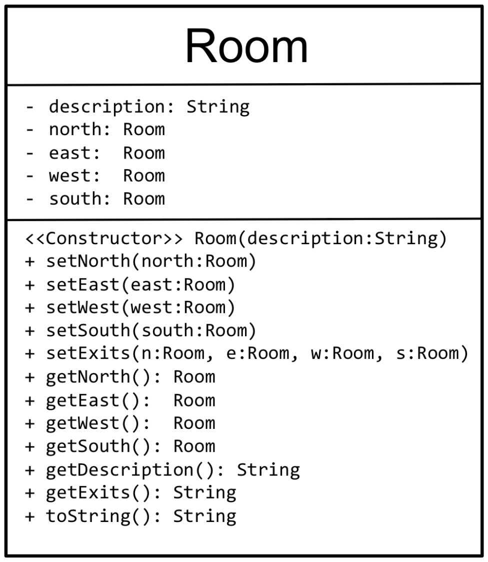
The *instructions* for assembling the biplane detail how the plane model should be created from its constituent set of parts. In OOP, this is the job of the **constructor**. The constructor would build an instance of the plane model. Once created, we can then use it to do things, **instance methods;** which define the behaviors the plane can have. This might include a spinning propeller and rolling wheels.

Fundamentally, the goal of OOP is to create your applications using software building blocks. Each building block has some use that the system needs done. For example, if your application is the board game monopoly then you may want to model as objects: dice, money, game board, player tokens, event cards, house markers, hotel markers, and deeds.

Modeling the Adventure Game using Objects:

The first step to refactoring your adventure game is to identify the objects that are required to build your game. A good rule of thumb is that you should divide the application's responsibilities into objects. You should use UML diagrams to determine the objects' properties and behaviors. In your adventure game the user explores a dungeon, which consists of a series of interconnected rooms. So, you must define a class that represents a room object and a dungeon object. Below are UML diagrams for our own implementation, but feel free to create your own or add more to ours. You may need more classes than we have if you added more game features, suggested additions could include: items, player, monster, traps, doors.

Room



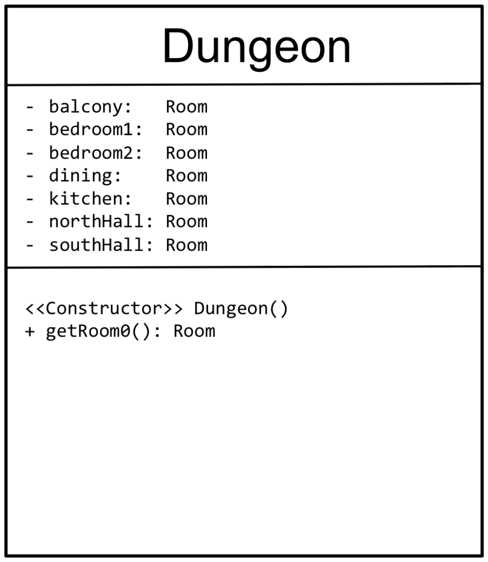
```
Room
- description: String
- north: Room
- east:  Room
- west:  Room
- south: Room
<<Constructor>> Room(description:String)
+ setNorth(north:Room)
+ setEast(east:Room)
+ setWest(west:Room)
+ setSouth(south:Room)
+ setExits(n:Room, e:Room, w:Room, s:Room)
+ getNorth(): Room
+ getEast():  Room
+ getWest():  Room
+ getSouth(): Room
+ getDescription(): String
+ getExits(): String
+ toString(): String
```

UML class diagram for Room

Room Class

Rooms are responsible for representing an area in the dungeon. The properties that define a room are its description and its exits. The room description may be stored using a String. For the exits, this room can hold references to other rooms that connect to it. The room constructor only sets up its description leaving its exits initially null. Client code can then update the room's properties with getter and setter methods. It would be convenient to also have a method that can set all 4 exits at once. The room's description shouldn't say its exits; instead have a method (getExits) that creates a String of exits. The `toString` method should contain the description and the exits.

Dungeon



```
Dungeon
- balcony:   Room
- bedroom1:  Room
- bedroom2:  Room
- dining:    Room
- kitchen:   Room
- northHall: Room
- southHall: Room
<<Constructor>> Dungeon()
+ getRoom0(): Room
```

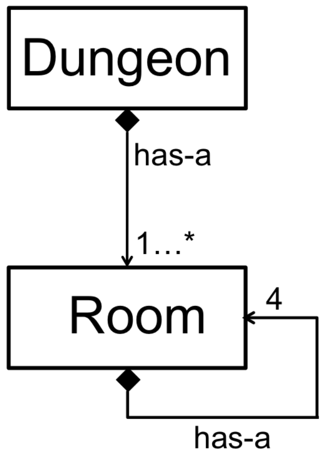
UML class diagram for Dungeon

Dungeon Class

The dungeon is responsible for setting up all the rooms and establishing the connections between them. The dungeon's properties (i.e. instance variables) are the rooms that it contains. The dungeon constructor should initialize all of the rooms' descriptions and its exits. The only method that our dungeon needs is to provide the starting room. Once we have the starting room we can navigate to any other room from using the rooms themselves.

Composition

Both Dungeon and room use composition since they have a *has-a* relationship to room. It's important to understand that an object can hold a reference to its own type so a room can have references to the other rooms that connect to it. The ability to link objects directly to one another is a critical aspect in understanding data structures. Below is UML relationship diagram showing the relationships that our two classes have with one another.



UML Class relationship Diagram

Dungeon has 1 or more instance of class *Room.*

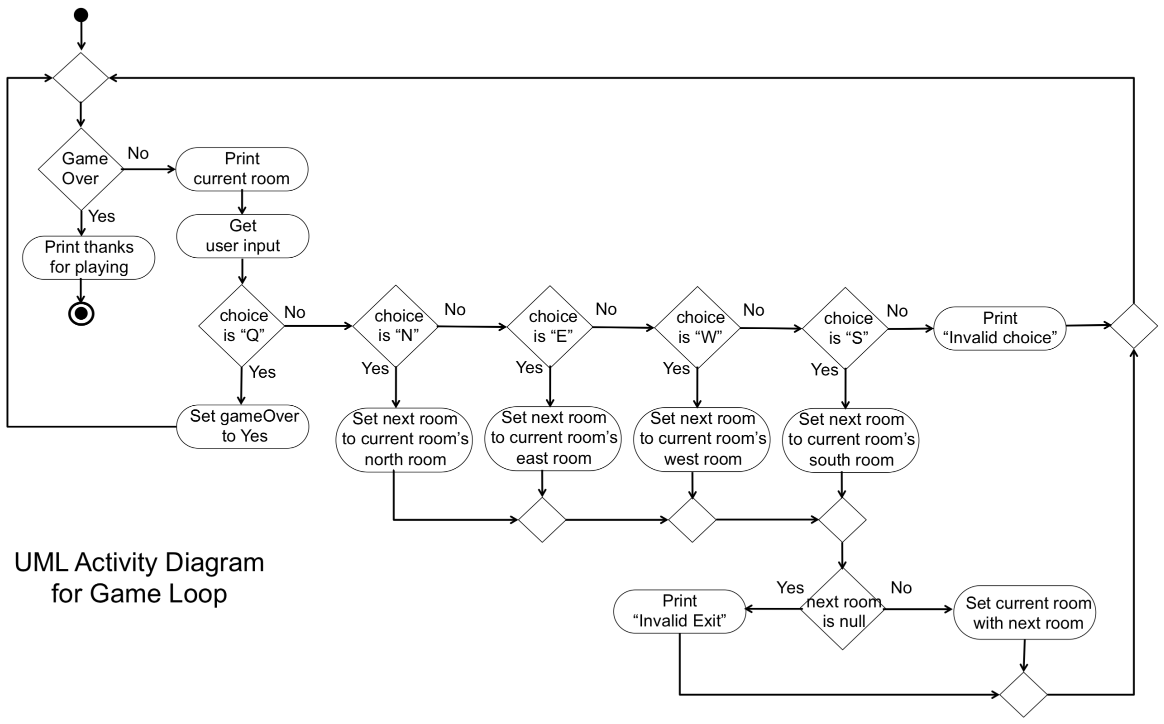
Room has exactly 4 instances of class *Room*, one for each direction.

Game Application

The application contains the main method and the game logic. The base implementation needs only three pieces of data to setup for the game:

- The starting room, which we get from the dungeon class
- The loop control variable, gameOver, user for the game loop
- User input variable

Below is the UML activity diagram used to illustrate the application's flow. You can break the tasks within the activity diagram into multiple methods if you'd like. You are also free to add additional options within the game. This is simply an example of how you may do this.



UML Activity Diagram
for Game Loop

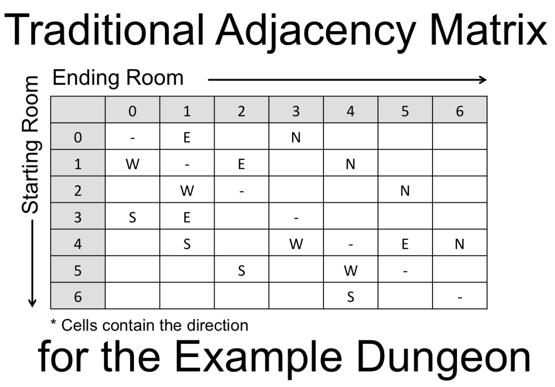
The game loop should repeat until the player decides to quit the game.

You may want to split up your game logic into different methods when appropriate. Remember that a good method should perform a single specific task. For instance, you might have a method `executeDirection(Room current, String choice)` that returns the destination room (or null) given a current room and user input.

Concluding Note: Two Different Approaches to Modeling Exits.

Adjacency Matrices

In the previous assignment we had used a 2d integer array to model a dungeon's exits. The underlining mathematical premise for this approach comes by way of discrete mathematics called an Adjacency Matrix. In a traditional adjacency matrix the rows are the starting nodes (i.e. rooms) and the columns are the ending nodes (i.e. rooms). We had slightly modified this concept similar to the table on the right so that we could easily implement it in Java.



Traditional Adjacency Matrix

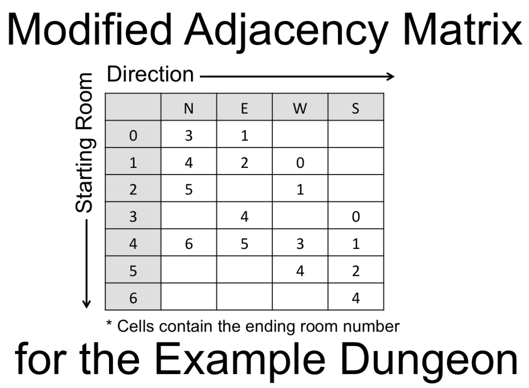
Ending Room →

Starting Room ↓

	0	1	2	3	4	5	6
0	-	E		N			
1	W	-	E		N		
2		W	-			N	
3	S	E		-			
4		S		W	-	E	N
5			S		W	-	
6					S		-

* Cells contain the direction

for the Example Dungeon



Modified Adjacency Matrix

Direction →

Starting Room ↓

	N	E	W	S
0	3	1		
1	4	2	0	
2	5		1	
3		4		0
4	6	5	3	1
5			4	2
6				4

* Cells contain the ending room number

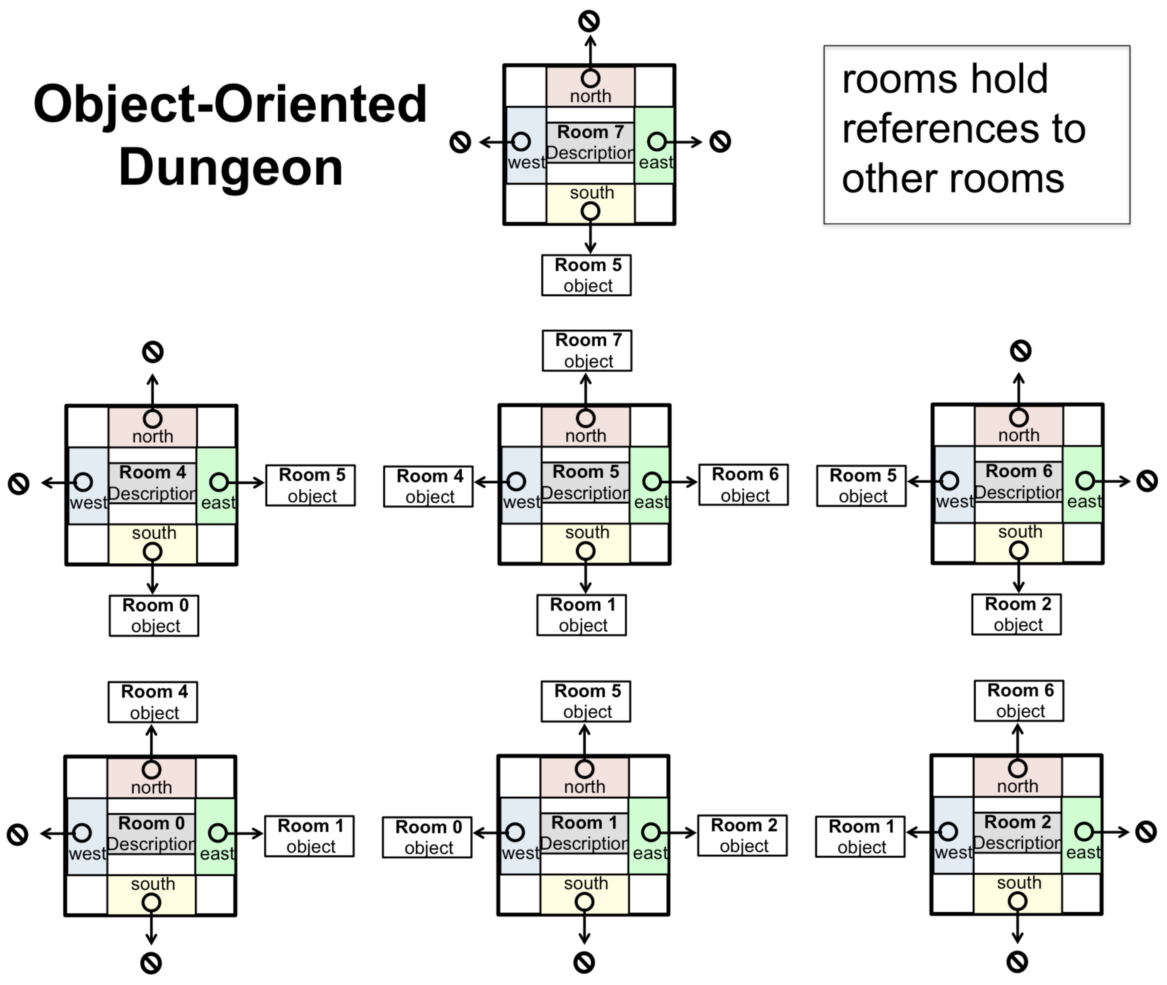
for the Example Dungeon

One disadvantage of using adjacency matrices is that they require a global view of the dungeon to construct it. This means that the matrix requires that you know how every room interconnects with one another. The constraint of having a global view of the system is limiting. If you wanted to add a new room you'd have to rebuild the entire adjacency matrix. If you wanted to combine dungeons with a classmate, you would have to make a whole new adjacency matrix and re-label all of the rooms. In other words, it's not easy to make additions or changes to adjacency matrices.

Linked Nodes

In this assignment we instead link rooms to one another. This method is similar to techniques you will learn further in Data Structures with Linked List. The idea is that each room contains the information about itself, its description, and then also contains references to the other rooms that it connects to. The advantage of this approach is that we never require a global view of the dungeon. Each room only needs a local knowledge of the dungeon; it only needs to know its neighboring rooms. This allows us to start building a dungeon without needing to know every room that it will contain. Because of this we could potentially randomly generate rooms, or distribute the work of making rooms to multiple people, or even easily combine or connect multiple dungeons together. A graphical illustration of how this linked node approach holds the data for our dungeon is given below.

Object-Oriented Dungeon



Bonus:

Add additional objects beyond these specifications. For example, you might create player and monster classes that allow for combat encounters, picking up treasures, or have interesting victory or defeat conditions. Be imaginative and have fun with this project.