Project 3 - Dungeon Model

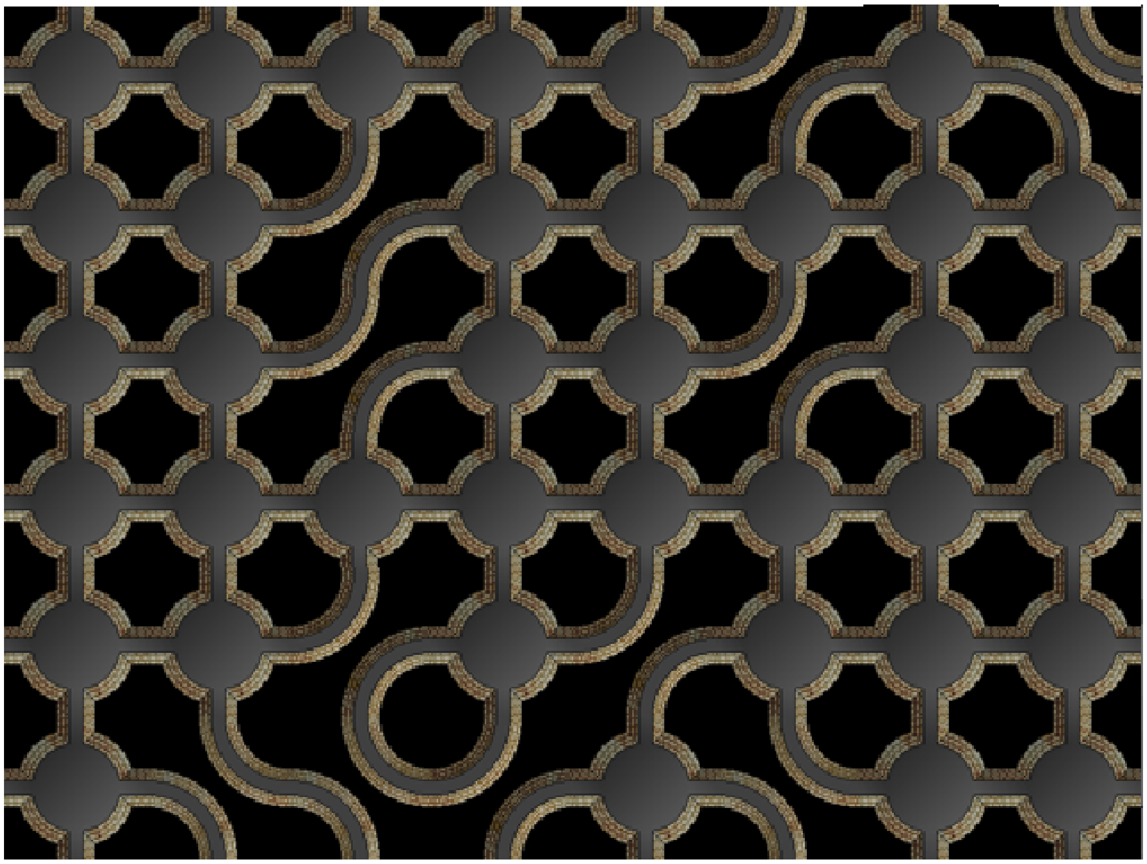
Objective

The goal of this assignment is to design and implement a model that we will use for a full MVC project.

For the remaining three (3) projects this semester, we will be implementing an adventure game using the MVC design pattern. This project consists of the model for our game and represents the first part of this process. Students are encouraged to keep this in mind during the design process but to be cautious with doing more than each parts asks.

Dungeons

The world for our game consists of a *dungeon*, a network of tunnels and caves that are interconnected so that player can explore the entire world by traveling from cave to cave through the tunnels that connect them. Consider the following example:



This example dungeon is represented by a 6 x 8 two-dimensional grid. Each *location* in the grid represents a location in the dungeon where a player can explore and can be connected to at most four (4) other locations: one to the north, one to the east, one to the south, and one to the west. Notice that in this dungeon locations "wrap" to the one on the *other side* of the grid. For example, moving to the north from row 0 (at the top) in the grid moves the player to the location in the same column in row 5 (at the bottom). A location can further be classified as *tunnel* (which has exactly 2 entrances) or a *cave* (which has 1, 3 or 4 entrances). In the image above, we are representing caves as circular spaces.

In many games, these dungeons are generated at random following some set of constraints resulting in a different network each time the game begins. While the details on algorithms for generating a dungeon for our game are given in Part 2 - Implementation, we consider that an implementation detail. We provide the constraints that should be used for the design process:

- The dungeon should be able to be represented on a 2-D grid.
- There should be a path from every cave in the dungeon to every other cave in the dungeon.
- Each dungeon can be constructed with a *degree of interconnectivity*. We define an *interconnectivity = 0* when there is exactly one path from every cave in the dungeon to every other cave in the dungeon. Increasing the degree of interconnectivity increases the number of paths between caves.
- Not all dungeons "wrap" from one side to the other (as defined above).
- One cave is randomly selected as the *start* and one cave is randomly selected to be the *end*. The path between the start and the end locations should be at least of length 5.

Model Requirements

The model should create and support a player moving through the world. To do this, it should support operations that allow:

- both wrapping and non-wrapping dungeons to be created with different degrees of interconnectivity
- provide support for at least three types of treasure: diamonds, rubies, and sapphires
- treasure to be added to a specified percentage of caves. For example, the client should be able to ask the model to add treasure to 20% of the caves and the model should add a random treasure to at least 20% of the caves in the dungeon. A cave can have more than one treasure.
- a player to enter the dungeon at the *start*
- provide a description of the player that, at a minimum, includes a description of what treasure the player has collected
- provide a description of the player's location that at the minimum includes a description of treasure in the room and the possible moves (north, east, south, west) that the player can make from their current location
- a player to move from their current location
- a player to pick up treasure that is located in their same location

It's not required, but it is a **good idea** to provide a way to dump the dungeon to the screen for debugging and testing purposes.

Part 1 - Design

Before you start to write code, it is a good idea to design your solution. To do this, you need to understand what your program needs to do, decide what classes you will need, and what methods each class will need. It is a really good idea to think about how each of these methods and classes could be tested to ensure the correctness of your implementation. Thinking about this early will make the coding process much easier.

What to do

Design the model for the dungeon in a way that captures the similarities and accurately represents the relevant data. Create interfaces/classes as you see fit and specify appropriate constructors. You should also be sure to capture what methods and fields those classes have, the visibility of the methods and fields, and the relationships between them.

Write a testing plan (see Module 1.4) that could thoroughly test your design. How do your tests convince someone else that your code works correctly? For each test in your design, you should specify what condition you are testing, what example data you will use to test that, and what values you might expect a method to produce (known as the expected value) when appropriate.

What to submit

A single PDF containing a UML class diagram that captures all aspects of your design and a testing plan for your model. Submit this to Handins by Friday, October 22nd.

Part 2 - Development

What to do

Implement the classes/interfaces that you specified in Part 1. Since this is a model, we will again create a **driver class** (as opposed to a full MVC application) that demonstrates how to use your model. Your driver class should:

- Use command-line arguments to specify size of the dungeon, its interconnectivity, whether it is wrapping or not, and the percentage of the caves that have treasure.
- Demonstrate the player navigating the dungeon, collecting treasure, and reaching the end location (see the *What to submit* for specific scenarios that you should be able to demonstrate).

Hint: Use the same strategy for random number generation as you did for the last project to get deterministic behavior.

Another Hint: You should avoid using `System.out` in the model. Remember that it is the controller and view, not the model, that are responsible for input/output.

Algorithm

We can build our dungeon using one of several algorithms for generating mazes. These include **depth first search** along each branch before backtracking), **Kruskal's algorithm** , or **Prim's algorithm** . Each of these algorithms easily generates a maze that can meet the criteria for our dungeon that there is a path from every cave in the dungeon to every other cave in the dungeon. However, we also want to generate a degree of interconnectivity between the locations in our dungeon, that is most easily generated using **Kruskal's algorithm** .

To build our dungeon, we will need to use the command-line arguments that were provided to build our dungeon. In this video, we demonstrate how to modify Kruskal's algorithm to build these mazes:

Documentation

We expect your code to be well-commented using well-formed English sentences. The expectations are:

- Each interface and class contains a comment above it explaining specifically what it represents. This should be in plain language, understandable by anybody wishing to use it. Comment above a class should be specific: it should not merely state that it is an implementation of a particular interface.
- Each public method should have information about what this method accomplishes (purpose), the nature and explanation of any arguments, return values and exceptions thrown by it and whether it changes the calling object in any way (contract).
- If a class implements a method declared in an interface that it implements, **and** the comments in the interface describe this implementation completely and accurately, there is no need to replicate that documentation in the class.
- All comments should be in `Javadoc`-style.

Create a JAR file of your program

In order to make your application easier to run, you are required to create and submit an executable `JAR` file:

- Directions for doing this in Eclipse can be found at **this link** .
- Directions for doing this in IntelliJ can be found at **this link** .

What to submit

Log on to the **Handins submission server** and upload a ZIP file of your assignment. Your ZIP file should contain three folders: src/, test/ and res/ (even if empty).

- All your code should be in src/.
- All your tests should be test/.
- Your design documents (original and revised) should be in res/.
- Submit a correct `JAR` file in the res/ folder. We should be able to run your program from this jar file using Java 11 from a command line using the command `java -jar NameOfYourFile.jar`
- In the res/ folder, also submit at least two example runs of your driver program in a ***simple text file*** that can be used to verify that your model meets all of the above specifications. Your runs should include:
  - One run that shows a wrapping dungeon
  - One run that shows a non-wrapping dungeon
  - One run that shows the player visiting every location in the dungeon
  - One run that shows the player starting at the *start* and reaching the *end*
  - One run that shows that the player's location and description at each step
- Submit a README.md file that documents your program [following these guidelines].
- Your zip file should also be free of IDE configuration files, compiled .class files, and other non-essential files that are automatically generated. The META-INF/MAINFEST.MF file is required for your JAR file to be executable and should be included.

Criteria for grading

Your projects will be assessed in three different ways:

1. Completeness, correctness, design, and testing will be assessed through the completion of a *self-evaluation* similar to the ones you have been doing for lab assignments. In the self-evaluation, you will be asked to answer a series of questions about your code, provide pointers (tags) into your code, and provide explanations of your answers and/or implementation. The pointers into your code will allow us to confirm your answer since they will show us where to look in your implementation. Self-evaluations are worth ~50% of the project grade.
2. Style and whether you are following good programming principles will be assessed on the Handins server via your submission. Style is assessed automatically by Handins and good programming principles will be assessed manually. To see details of what will be manually assessed, refer to the [Manual Grading Checklist]. The submission and the manual grading of said submission is worth ~20% of the project grade.
3. Finally, you are asked to defend your design and/or implementation choices in a meeting with your instructors for the last ~30% of the project grade including submitting a reflection of your meeting experience.