Module 3 Lab -- Building Dungeons

Due Sep 26 by 11:59pm **Points** 100 **Available** after Sep 19 at 12am

> The goal of this lab is practice the design and implementation of a complex object using the Builder pattern.

Building Dungeons

One of the tasks that game developers face is the creation of complex game levels using a relatively small number of simple classes that represent a large variety of elements in lots of games. Each game has its own type of room, its own set of monsters, and treasures that are specific to it. For instance, a game set in a medieval fantasy could have dark dungeon rooms full of orcs and goblins while a game set in the future could have rooms full of digital screens full of other humans and aliens. To ensure that their worlds are built to their specification, they will often construct levels using the Builder pattern where the Builder object is not only responsible for creating the final `Level` object, but will also ensure that the objects that are added to the level are from a predefined set.

What to do

Package: dungeon

Start by downloading **this dungeon.zip archive** that contains four classes that can be used to create a level in a game: `Level`, `Room`, `Monster`, `Treasure`. After adding these files to your project take a moment to familiarize yourself with each of them.

Implement a `MedievalLevelBuilder` class that can be used to construct a level in a game that is set in a medieval fantasy. This class will be used to not only build a level for a game, but it will also ensure that all of the details of the game are consistent. To do this your builder class should have the following:

- A constructor that takes the number of the level that is being created since most games have many levels. It should also take non-negative values for the target number of rooms, monsters, and treasures that the level is expected to have.
- An `addRoom` method that has a single parameter for the room's description and adds a room with that description to the level. The method throws an `IllegalStateException` if too many rooms are added to the level.
- Methods for adding four different types of "monsters" to the specified room as the first parameter (by index starting with 0):
  1. A **goblin** is a "mischievous and very unpleasant, vengeful, and greedy creature whose primary purpose is to cause trouble to humankind" and are the weakest type of monster in our

level. They are quite numerous and often travel in packs. The `addGoblins` method should add the specified number of goblins to the specified room giving each 7 hit points.

2. An **orc** is a "brutish, aggressive, malevolent being serving evil" but tends to be more of a loner than the goblins. The `addOrc` method should add a single orc to the specified room giving them 20 hit points.
3. Even stronger than orcs are orges. An **ogre** is a "large, hideous man-like being that likes to eat humans for lunch". They have 50 hit points.
4. Our dungeon can also contain humans that will be stored as a type of monster. The details of the human must be provided to the `addHuman` method.

Adding monsters to the level should raise an `IllegalStateException` if the target number of monsters has already been reached and an `IllegalArgumentException` if the target room has not yet been created.

- Methods for adding four different types of treasure to the specified room as the first parameter (by index starting with 0):

1. The `addPotion` method should add "a healing potion" (value = 1) to the specified room.
2. The `addGold` method should add "pieces of gold" of the specified value to the specified room.
3. The `addWeapon` method should add the specified weapon to the specified room. All weapons have a value equal to 10.
4. The `addSpecial` method can be used to add the most exclusive treasures to the specified room.

Similar to adding monsters, adding treasures should raise an `IllegalStateException` if the target number of treasures has already been reached and an `IllegalArgumentException` if the target room has not yet been created.

- Finally, your builder class should have a `build` method. This method should return the level only after it has been completely constructed. If it is called before completion, it should raise an `IllegalStateException`.

Testing

Whenever you write a class, you should also write tests for that class that proves not only that your code CAN work but that it WILL ALWAYS work. Your goal here is to write tests to achieve as close to 100% coverage as possible. But even more importantly, your tests should be sufficient to ***convince someone else that your code works correctly***.

What to Submit

1. Create a zip file that directly contains only your src/ and test/ folders. When you unzip the file, you must see only these two folders.

2. Log on to the **Handins server** **(https://handins.ccs.neu.edu/)**
3. Navigate to this lab and submit the zip file.
4. Wait for a few minutes for the grader's feedback to appear, and take action if needed.

Grading Criteria

This assignment will be assessed via the automatic testing available on the Handins server. You should ensure that your code does not have any code style violations and that it passes all of the test cases. This is worth 1% of your final grade.

Additionally, your code will be assess via peer evaluation. Each student will be required to review one other student's code for an additional 0.5% of their final grade. During the peer review your code will be evaluated based on three criteria:

1. Design and organization of classes, interfaces, and methods including
   - How appropriately you captured the data and operations in the implementation
   - Whether code looks well-structured and clean (not unnecessarily complicating things or using unwieldy logic)
   - How well your design and implementation embraces various design principles that you have learned so far
2. Understandability and quality of documentation including
   - Whether the documentation enhances the understandability of the code rather than just repeating how something is implemented
   - Whether it is stand alone, meaning whether the documentation is sufficient for use without reading the implementation
3. Quality and coverage of tests including
   - Whether tests are modular -- do you write one big, monolithic test method or does your test class have separate methods for each test
   - Whether tests are written in such a way so that the reader knows what is being tested
   - Whether the tests convince the reader that if all tests pass then the code it is testing works correctly