CS 245
Zuul

Objective:

- Consider issues of good/bad design.
- Experience the process of refactoring code

Problem:

In my day, we didn't have fancy computer graphics, real-time simulations or your so-called Virtual Reality. A 3-color CGA screen? LUXURY! We played text-based adventure games and we liked it. The world-of-Zuul is a game that is designed to recapture that golden age and let you experience some ***real*** fun. No really – this code is designed to abstract away the details of a complex game to help you consider the interplay between objects and how they should (or shouldn't) interact with each other.

For this task, you are simply going to walk through a series of modifications to some existing code. Sometimes, I am going to ask you to do something that you think is nonsensical. In that case, I applaud your intuition for seeing through my folly – but do it anyway. The mistakes maybe transparent and exaggerated in this simplified activity, but they represent the kinds of mistakes that can be made in larger projects, where the complexity of the project masks the silliness of the "solution". This example can serve as a scaffold for you to stand on when you find yourself in the middle of a more complex task.

Activities:

Clone the Zuul Repository to your local workspace.

- Start the game running and play around to see what the game is about.
- Add your name to the top of ZuulWorld.java.
- Create a readme file that you can use to answer questions you work through this activity. You can also use this file to keep notes for yourself. If you have any questions about why we are doing something, add them here.
- Answer the following questions about the game your readme.

    - [Q1] What does the game do?
    - [Q2] What commands does the game accept? What does each command do?
    - [Q3] How many rooms are in the scenario?

- Draw a map of the existing rooms. (just a simple sketch to help you visualize some relationships

- When you are done playing, open each of the six classes in the project and explore what they do. TAKE YOUR TIME. Spend some time on playing the game and examining the code so that you feel comfortable with what is going on.

[Q4] Consider which classes know about and talk with which other classes?
Try to sketch a diagram that shows how these classes relate to each other.

What happens if we want to make a "simple" change to our game?

Now that you have looked at the code for the Zuul game, you might wonder what the big deal is. Why is this considered to be "bad" code?

[Q5] Do you see any glaring problems with the design of this code?

We will start by having you make a couple of "simple" modifications to the game and looking at how difficult this process is when the code isn't well designed.

For example, suppose that we want to add a pool hall on the second floor above the campus pub. In order to do this, we need to add in the ability to move up and down, and we need each room to know whether there are other rooms above it and/or below it. By adding in the concept of an upExit and a downExit to each room in our game we can eventually tell the pub that it has an exit going up to the pool hall and tell the pool hall that it has an exit going down to the pub.

- Open the Room class and look at how it currently stores and sets exits.
- Add two new ***public*** instance variables - an upExit and downExit.
  (I know, you should never to use public instance variables. Do it here anyways).

Once you do this, you need to have a way to set these values. It would seem that the easiest way would be to update the setExits() method.

- Modify the setExits() method so that it accepts SIX parameters - the four already there and parameters which will set the two new instance variables.
- Attempt to build the project. (of course it won't work, but do it anyway)
- As you probably expected, it won't compile because there are dependencies. You changed the way that Game class interacts with the Room class. In particular, setExits() expects six parameters, not four.
- Within the Game class, update all of the calls to setExits() by adding two additional parameters to each call. For the time being, let's let these each be "null"
- Now attempt to compile the Game class. This time, it should work.

Ok, now that we have the functionality in place to allow for rooms to have exits up and down, let's add the pool hall to our game.

[Now is a good time commit your changes using the message: "UpDown"]

- Locate the createRooms() method in the Game class.
- Find the section of code where the various Rooms are declared and constructed. Add a line of code which declares and constructs a variable named something like "pool" Construct the pool variable using a description like "in a seedy pool hall."
- Find the section of code where each room has its exits set. Invoke the setExits() method on this pool hall and set its only exit to be a down exit to the pub (You may have to review which of the six parameters you decided was "down" inside of the Room class.
- Modify the setExits() method for the pub so it now contains an up exit to the pool hall.
- Compile your code and fix any errors you may have introduced.
- Play your game and navigate into the pub. Try to get up to the pool hall. Can you? What problems exist?

OOPS! The pub doesn't **tell** us that it has an up exit so that we have a way to get up to the pool hall. How do we fix that?

- Start at the top of the Game class and slowly scan down the code.
- When you get to the printWelcome() method, notice the long list of "if" statements which check whether or not the code should tell us that there is an exit in a certain direction.

- Update this list to include if statements for the up and down exits.
- Compile your code and fix any errors you may have introduced.
- Play your game and navigate into the pub. Try to get up to the pool hall. Can you? What problems exist?

DANG IT! This didn't fix the problem. How come? Well, notice we just updated the printWelcome() method. That method only runs at the start of the game. This code is repeated (ugh) in the goRoom() method. This is called whenever you enter a new room (like the pub).

- Update this list to include if statements for the up and down exits.
- Compile your code and fix any errors you may have introduced.
- Play your game and navigate into the pub. Try to get up to the pool hall. Can you? What problems exist?

You should observe that while the pub will TELL you it has an exit going up, it won't let you actually do this; that was actually handled in the first part of the goRoom() method.

- Relocate the goRoom() method and update the first long list of ifs so that it includes references to the up and down exits.
- Compile your code and fix any errors you may have introduced.
- Play your game and navigate into the pool hall. Can you do it?
- If not, why not?
- If so, navigate back out to the main entrance of the world so that you test that the other parts work.

[Commit your changes: "Pool Hall accessible"]

Ok, that was harder than it should have been! - Removing Duplicate Code

That should have been easy and it wasn't. Why?

Hopefully, you recognize that this has been so hard because of code duplication. Of course ***you*** would have never written code this bad. When you copy-paste code, you should probably feel bad about yourself.

Code duplication is an indicator of bad design. The Game class you have been working with contains a case of code duplication. The problem with this is that any change to one version of the code must also be made to the other if we are to avoid inconsistency. This increases the amount of work a maintenance programmer has to do, and it introduces the danger of bugs. It happens very easily that a maintenance programmer finds one copy of the code and, having changed it, assumes that the job is done. There is nothing indicating that a second copy of the code exists, and it might incorrectly remain unchanged.

To illustrate this point, notice that the printWelcome() and the goRoom() methods contained in the Game class both contain a block of code that :

- prints the description of the room and
- lists the current "Exits: " by working through a series of if statements which decide if the code should print that there is an exit to a given direction.

As you observed, this is a problem when it comes time to modify the code. You have to KNOW that both exist and you have to REMEMBER to update both of them when you make a simple change.

The easier thing to do is to remove duplicate code to a single, private, helper method.

To do this, implement and use a separate printLocationInfo() method in your Game class and replace the duplicate code discussed above so that it refers to this helper method instead.

Did this change the functionality of your code? No. It will still work the same way, but it makes it easier to maintain and easier to change. Whoever wrote this code was probably in a hurry and just used copy-paste to get things working. That is fine – but if you are going to do that, you should also expect to do some cleanup refactoring of your code at some point. It is worth it!

[Commit Changes: printLocationInfo]

Reducing the coupling

Another issue present in this code is the concept of coupling. Coupling is the interdependency between two (or more) classes. You witnessed what it means to have high coupling when you added in the up/down exits - what should have been a fairly simple change to the Room class turned out to require serious changes to the Game class as well (in fact, you made MORE changes to the Game class than you did to Room). Why? Because the Game class has LOTS of references to the instance variables stored inside the Room class.

If you haven't noticed it yet, notice that the instance variables in the Room class are all public. By having public instance variables in Room, Game has permission to access these variables all over the place. While this might seem like a good thing, it often times makes it impossible to clean up the Room code without seriously breaking the Game code.

Let's push this issue a little more.

The main problem with this "bad" code is that we have the Game class doing things that should be the responsibility of the Room class. For example, in the goRoom() method, the Game class is digging around in the instance variables of the Room class to figure out what the value is for a particular exit so it knows what the nextRoom will be. BUT WAIT, knowing and interpreting a Room's exits should be the job of the Room class. We have our ***responsibilities*** in the wrong place. Instead of having Game determine where to go next, we should just ask the Room class where to go next.

All of this is a sign that it is time to refactor your code. In this case our refactoring is going to "fix" working code with a better design.

Let's reduce the coupling between Game and Room by moving some code around...

- Add a public method to the Room class called getExit()
    - This method should take one String as a parameter. This String represents the direction the user says they want to go. For example, if the game wants to know which room is to the north of the current room, it could say getExit("north")
    - This method should return a Room. This would be the room in the direction that matches the parameter to the method.
    - The body of this method should consider the six different Strings that could have been passed in as a parameter (things like "north", "east", "up" and return the exit that matches the input. If the user entered an invalid direction, the method should return null.
- Once you have the getExit() method working, you can reduce the coupling between Game and Room by having the goRoom() method in the Game class call the getExit() method instead of performing the long if sequence.

[Commit changes: Reduced Coupling, getExit()]

Even More reduction of coupling

The last part helped us, but we still have a heavy link between Game and Room. Notice that the printLocationInfo() method in the Game class is also peeking at all of the instance variables in instances of the Room class. Why is the Game class determining which exits are valid for a given room. Shouldn't that be the Room's job?

Let's reduce the coupling *even more*, by continuing to move some code around...

- Add a public method to the Room class called getExitString()
    - This method should take no parameters.
    - This method should return a String consisting of all of the valid exits for that room.
    - This method should build the return String by considering each of the exit instance variables to see if that direction should be part of the return String. This is very similar to what is happening in the printLocationInfo() method in the Game class, except rather than printing right away, it should be appending a return String.
- Once you have the getExitString() method working, you can reduce the coupling between Game and Room by having the printLocationInfo() method in the Game class call the getExitString() method.

Skim down the code for the Game class. If you have been following all of my directions, and have completed everything correctly, you should notice that Game no longer has ANY references to instance variables in the Room class. Just to prove the point...

- Open the editor for the Room class.
- Change the instance variables to private variables.
- Compile and execute your game code. Does it work?

[Commit Changes: getExitString Decoupled Game, Room]

Why this change made our code better.

So, you might think that this was mostly cosmetic, but notice that we have largely separated behavior from implementation now. From now on, any changes to the way we store exits have been isolated within the Room class where they belong rather than out in the Game class (where they don't belong).

Suppose that we get tired of having six instance variables storing six exits - this is just a huge problem. Ok, six might not be too bad, but maybe we decide to add four more exits on the diagonals (like NorthEast and SouthWest).

To help with this, we will use a HashMap.

The Java API for the HashMap is located at
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashMap.html

However, the basic use is:

```
HashMap< String,String > phoneNumber;
...
phoneNumber = new HashMap< String,String >();
...

phoneNumber.put("Reed","574-2367");
phoneNumber.put("Hughes","574-7429");
...
number=phoneNumber.get("Hughes")
```

If you have had Python, you should probably recognize this as a Dictionary. A HashMap is a non-ordered collection of items that stores things with a <Key,Value> pair. That is, rather than saying, "let me see the 4th object in the collection" you say, let me see the Value associated with the Key named someName. In the example above, I am asking to look up the phone number associated with the name "Hughes"

You must declare the data types of the keys and values in your HashMap at declaration and construction. The keys must all have the same data type and the values must all have the same data type, but they need not be the same between the keys and values as mine are. i.e. it is possible to create:

```
HashMap< String,Integer > wordCount;
```

More to the point, you could make a HashMap that takes a String such as "east" as the key and returns the actual Room to the east as a value.

Suppose you decided to replace these current six exit variables with a single HashMap. To do this:

1. replace the six current Room variables with a single HashMap called exits. You will need to define it's templates to be a String for the key and a Room for the value.
2. In the constructor, make sure you construct exits even though it won't get any values at this time.
3. Modify the setExits() method to change the condition for each if statment to say something like

   exits.put("north", north); //Again, notice the key is a string and the value is a Room

4. BUT WAIT, setExits() really isn't a good method in the grand scheme of things. If I wanted to add NW, SW, NE, and SE exits to a single room than I needed to extend the long parameter list from setExits() to include 4 more parameters - even though most of the rooms would simply set them to null. INSTEAD, refactor setExits() to be a method called setExit(). This method should take in TWO parameters - a String which is the label for a room and the Room object associated with that label. Then, the client code simply calls this for each room that it has an exit for.

   In other words, instead of saying:

   ```
   outside.setExits(null, theatre, lab, pub, null, null);
   ```

   We would say:

   ```
   outside.setExit("east", theatre);
   outside.setExit("south", lab);
   outside.setExit("west", pub);
   ```

5. Modify the getExit() method to use HashMap's get() to use the String parameter (key) to return the Room value.
6. You will also need to adjust Room's getExitString() method. Hint: Look into the concept of a HashMap's key set.

What we have done here is ***delegated*** this functionality to another class. In this case, it is a class that organizes the data and has access rules that are aligned with our task. It is very helpful to choose your data structures wisely.

[Commit Changes: Decoupling – Game, Room]

Reducing coupling between Game and CommandWords

The previous activities showed us a way to reduce explicit coupling between Game and Room. It was called explicit coupling because we explicitly tied the two classes together by letting Game have access to Room's instance variables. This project has even more examples of coupling in it. However, this is an even worse form of coupling: Implicit Coupling.

Implicit coupling is a situation where one class depends on internal information of another, but this dependence is not immediately obvious. The tight coupling in the case of the public fields was not good, but at least it was obvious. If we change the public fields in one class, and forget about the other, the application will not compile anymore and the compiler will point out the problem. In cases of implicit coupling, omitting a necessary change can go undetected.

We can see the problem arising if we try to add further command words to the game.

Suppose that we want to add the command “look” to the set of legal commands. The purpose of look is merely to print out the description of the room and the exits again (we ‘look around the room’) – this could be helpful if we have entered a sequence of commands in a room so that the description has scrolled out of view, and we cannot remember where the exits of the current room are.

We can introduce a new command word by simply adding it to the array of known words in the validCommands array in the CommandWords class:

- Add the String "look" to the array of validCommands stored inside of the CommandWords class.
- Compile and execute the game.
- Enter the command 'look'. What happens?

- In contrast, enter the command 'dance'. What happens when the game doesn't recognize your command?

Ok, we have a couple of problems that need fixing...

- Locate the processCommand() method in the Game class.
- Notice that this contains a series of if/else if statements which process different commands.

We want this list to include an action when the player types in "look." What seems like a good action? Well, we recently wrote a method called printLocationInfo() that describes the location. That seems reasonable.

- Add an additional else if statement into the list of statements under the processCommand() method. This should simply say that if the commandWord is look, then invoke the printLocationInfo() method.

That half fixes our problem, but we still have a coupling problem we were likely not aware of.

- Run the game and enter the command 'help'. What do you notice about the list of commands provided to you?
- Locate the printHelp() method in the Game class. Notice that the commands are hard coded in this method. That means every time you add a command to the list in CommandWords, you have to update the code in Game. THIS IS BAD.

- Add a showAll() method to the CommandWords class. This method should take no parameters and **return a String containing all of the valid command words**. The body of the method should loop over the array of validCommands building the output String (similar to how the getExitString() method worked in the previous problem) and then return this String.

While it would be easy to modify the printHelp() method in the Game class to now call this showAll() method. Notice that the current design doesn't have Game talking to the CommandWords class. Instead, it talks to the Parser and the Parser talks to the CommandWords class. For a variety of reasons, this is a pretty good design, although it makes your job a little more complicated

- Add a method called showCommands() to the Parser class. This method should take no parameters and should return a String. The body of the method should do nothing more than invoke the showAll() method from the CommandWords class and "pass on" the String that showAll() returns.
- Modify the printHelp() method in the Game class to invoke the showCommands() method from the Parser, and print the String that it receives.

[Commit Changes: Decoupled: New Commands]

One of the ideas behind class design is that each class should have a singular responsibility. Update your readme with [Q6] Articulate (in one sentence) the responsibility for each class.

[Commit Changes: Finished]

Thanks to Ben Schafer for modifying an assignment from David J. Barnes and Michael Kolling, the original inspiration behind this assignment.