Stage 1: Commands

In Homework 3, you were asked to make improvements to your text adventure game making it possible to add additional directions of movement. The changes you made was an example of *explicit coupling*. The coupling was obvious because of the public fields in the Room class. This code base has an even worse form of coupling: *implicit coupling*. Implicit coupling is a situation when one class depends on the internal information of another, but this dependence is not immediately obvious. We see an example of implicit coupling when we try to add commands to our game.

Currently, there are only three commands possible in our game: *go*, *help*, and *quit*. Obviously you will want to add additional commands to the game. Before we can do this, we will remove as much as the implicit coupling as we can. To this end, let's look at what it would take to add a new command to our game: the *look* command. The purpose of *look* is merely to print out the description of the room and exits again.

1. Identify all the locations in the code that will need to be changed in order to add a *look* command. How many are there? Be sure to list them all.

2. Add the *look* command to your game in the following way:
   (a) Add *look* as a new command word by adding it to the array of known words in the validCommands array in the CommandWords class. Execute the game and type the command *look*. What happens?

(b) Implement an action for the look command by adding a private method in the Game class with the following signature. For now, this method should simply call the printLocationInformation method. We will keep its implementation separate so that we can enhance its functionality later.

```
/**
 * Print out the description of the room and exits.
 */
private void look()
```

(c) Add a case for the *look* command in the processCommand method of the Game class that calls the method you wrote in part b).

(d) **Test your game and be sure that your new command is complete.** What is missing?

(e) Did you notice that the text in the *help* command is incomplete? The new commmand, *look* is not listed.

3. Responsibility-driven design says that since the CommandWords class is responsible for command words, it should also be responsible for generating a string containing all the command words. Modify your game to make CommandWords responsible for printing the list of commands in the following way:

(a) Add the following method to the CommandWords class:

```
/**
 * Return a list of the available commands, of the form:
 *       Your command words are:
 *           look go quit help
 *
 * @return A string containing the list of available commands.
 */
public String getCommandString()
```

(b) When we try to call our new method from the Game class, we realize that Game does not have a reference to the CommandWords object. Adding this would increase the coupling in our application. This is not necessarily a good idea. Instead we can use Parser. Let the Game talk to the Parser which in turn talks to CommandWords. We implement this by adding the following method to the Parser class and calling it in the printHelp method in Game.

```
/**
 * Return a list of the available commands
 */
public String getCommandString() {
   return commands.getCommandString();
}
```

4. To further enhance our project, we would like to remove some of the string comparisons that our game has to do. String comparisons are very inefficient and tend to be error-prone as many beginning (and advanced) programmers forget that they cannot use == to compare strings. In order to do this we need to change the CommandWords class where the valid list of commands is stored, and the Game class where the processCommand method explicitly compares each command word. We are going to do this using *enumerations*:

   (a) To start, add an enumeration to the project called CommandEnum.

   ```
   public enum CommandEnum {
       LOOK, GO, QUIT, HELP;
   }
   ```

   (b) Change the implementation of validCommands from an array of string to define valid commands to a map between strings and CommandEnum objects. This will make it easy to convert the command typed by a user into its corresponding enumerated type value. You should put the commands into the HashMap in the constructor of the CommandWords class.

   (c) Modify any methods that were using the array to now use the map. Use the compiler to help you identify every room that the validCommands is used. *NOTE:* You can (and should) also eliminate some bad style as the original author returned from the method from inside the loop.

   (d) Add a method to the CommandWords class which will convert a String into an CommandEnum object. It should have the following signature:

   ```
   /**
    * Convert a string into a CommandEnum object.
    * @param aString The string containing the command word.
    * @return The CommandEnum object representing the command.
    */
   public CommandEnum getCommand(String aString)
   ```

   (e) Change the implementation of Command so that the first word of every command is now a CommandEnum object instead of a String. This is best done by changing the type of the commandWord field and using the compiler to identify where changes need to be made.

   (f) Change the implementation of the getCommand method in the Parser to use the CommandEnum instead of a String.

   (g) Modify the processCommand method of Game to use a **switch** command with the CommandEnum object instead of using String comparisons.

5. Currently, whenever a new command is introduced into the game, we must add a new value to the CommandWords class. We want Comand to be more self-contained. Make this happen by enhancing CommandEnum in the following way:

   (a) Add a field to the CommandEnum class to contain the text of the command associated with each value

   (b) Define a private constructor which has a single parameter containing the initial value for the field.

   (c) Modify the list of enumeration constants to provide the default text associated with each type value.

(d) Modify the toString method to return the text associated with a particular type value.

(e) Modify the validCommands map in CommandWords to use the command text that is now stored in CommandEnum. If you do this right, you will remove the coupling between CommandWords and CommandEnum making it possible to add commands simply by adding them to the CommandEnum.

6. All this refactoring has left us in a great place for adding additional commands. Here are some that you need to add to your game:

Command	**Description**
look	Prints out the complete description of the current room.
back	Takes the player into the previous room he/she was in. This command does not have additional words. To keep it simple, your back command only has to remember the last place the player was, not the entire path the player took through the game. Be sure to think about what happens if the player enters 'back' more than once.
score	Prints out the current score. This should print out the current score in the following form: `Your current score is 42 out of 450 total possible points.`
time	Prints out the current number of turns that a player has used so far.
status	Prints out current state of the game including the complete description of the current room, the player's score, and number of turns that a player has used so far.

Deadline: Be sure to complete this by the assigned deadline.