CSCI 345
Assignment 3
Due: Midnight, May 13

Introduction

For this assignment we are going to build an engine for an extremely simple Interactive Fiction (IF) game. (See Assignment 2 for background on IF.) The game we will implement has five rooms and allows you to move between the rooms.

Please read the whole assignment before you start coding.

Problem Statement

You are to build a simple engine, in Java, for playing an IF game. This initial version of the engine will have *Words, VocabTerms, Actions,* a Command Interpreter, and *Room*s, *Path*s and the ability for the *Player* to move between *Room*s on *Path*s.

In order to make this a little easier (maybe) for you I am providing some code for your use in making the game. All of this code can be found in the file Assign3Data.zip which is on moodle.

- *Game.java*: This is the main program for the game. You should not change this class. For testing purposes, I will be using a somewhat different version of this class.
- *CommandInterp.java*: This is a skeleton for the command interpreter class. You can modify this class as you see fit as long as you do not change the public interface. Note that this class sets the commandIn and messageOut attributes in GameGlobals. You must use those two variables for all input and output for your game. You can use the System.in and System.out attributes from the java System class for debugging code. However, do not use System.in or System.out for any input or output associated with the game.
- *GameGlobals.java*: This is a start on the global variables for the game. I expect you to add additional variables to this class. The only variables included here are the ones required to make the other supplied code work. In the case of the ignoreWords attribute, you will need to provide the code that initializes this object.
- *HardCodedGame.java*: *HardCodedGame* defines the game we are building. You are not allowed to modify this file. This file depends on the following interfaces:
    - *IWord.java*, *IVocabTerm.java*, *IRoom.java*, *IPlayer.java*, *ICommandInterp.java*: These are interfaces describing the expectations within *HardCodedGame* for the *Word*, *VocabTerm*, *Room*, *Player*, and *CommandInterp* classes that you will be implementing. I expect you to add to these interfaces. Your *Word*, *VocabTerm*, *Room*, and *Player* classes should implement these interfaces. (The *CommandInterp* class I provided already implements *ICommandInterp*.)
    - *IBuilder.java*: This describes the interface that is used by *HardCodedGame* to build the objects that are part of the game. An object of type *IBuilder* is passed to the constructor of *HardCodedGame* by *Game.java*. You should not need to modify this interface.

    - *ActionMethod.java*: This is the interface for methods that are used as part of *Action*s. Objects that implement this interface are passed to the *makeAction* method in *IBuilder*. You should not modify this interface.
- *GameBuilder.java*: This is a skeleton for a class that implements the *IBuilder* interface. You must provide a class named *GameBuilder* that implements the *IBuilder* interface. (*GameBuilder* is used by *Game* and *IBuilder* is used by *HardCodedGame*.) Otherwise, you are free to modify this class as you see fit.
- *MatchType.java*: This is an enumeration that is used by *HardCodedGame* to specify the whether words are prefix matches or exact matches. You may modify this class as you see fit. However, there must be two values, *EXACT* and *PREFIX*, for the enumeration that specify exact and prefix matches for words. If you wish, you can add additional values and add attributes and methods to the enumeration.
- *GameUtil.java*: This is a small set of convenience functions that I found useful. You are free to use these or not as you see fit.

Note that these classes compile without errors or warnings. They don't do anything, and if you attempt to run the application, it will fail with a NullPointerException.

Packages

All the code I have given you is in package *cs345name*. (By convention, package names are all lower case.) The first thing you need to do is change the package name to *cs345* followed by your CS department user id. For example, in my case the package name would be *cs345reedyc2*. I will be unhappy (that means lost points) if I get files back without this change having been made.

All your new code must be in that same package or a sub-package of that package. For example, I could put all my command interpreter code in *cs345reedyc2.command*. (I didn't do this for this assignment, but might yet.) In all cases, do not have any code in the "default" package or any package that is not a sub-package of the *cs345<<yourname>>* package. That will also make me unhappy.

Note that it is not necessary to import into a class other classes that are in the same package.

HardCodedGame Class

In order to make *HardCodedGame* work, you must supply the following:

- Classes implementing *IWord*, *IVocabTerm*, *IRoom*, and *IPlayer*.
- Complete the *GameBuilder* class so that it creates and returns the appropriate objects when the *make...* methods are called.

Once you have a set of classes and methods defined with these properties, you should be able to run the game without a *NullPointerException* in *HardCodedGame*.

Command Interpretation

The command interpreter needs to work as described below. This is an expansion of the description in the previous assignment.

1. Read a line of text from the user and split the line into individual words. (The *canonicalCommand* method in *GameUtil* can do this.) If the line is empty, that is, it has no words, ignore it and continue with the next user input.
2. Match the words in the line against all words that are known to the program:
    - Word matching is case insensitive. That is “Go”, “GO”, “go”, and “gO” are all the same word. Note that words in *HardCodedGame* are all in lower case and that *canonicalCommand* returns lower case words. This makes case insensitivity very easy.
    - Words are matched either exactly, or by prefix. An exact match means that all the letters of the word were input as given. A prefix match means that some prefix of the word matched. For example, “nor” is a prefix match for “north” but only “north” is an exact match.
    - If multiple words match, that is an ambiguity unless one of the matches is an exact match and the other matches are prefix matches. If the match is ambiguous an appropriate message is output and processing continues with the next user input.
    - If there are words that are not known to the interpreter, an error message is to be output and processing continues with the next command.
3. If there are more than two words, an error message is produced and processing continues with the next user input. If there are zero words, that is treated the same as an empty line.
4. Check the matched words against the list of all actions. The words match the action if
    - there are two words and there are two non-null vocabulary items for the action and each word is in the corresponding vocabulary item.
    - There is one word and the word is in the first vocabulary item for the action and the second vocabulary item for the action is null.
5. There should be exactly one action matched. If no action matches, an appropriate error message should be produced and processing continue with the next user input.
6. Finally, having found the unique *Action* for this command, call the *doAction* method on the *Action*. That completes processing of this command.

Noise Words

A noise word is a word that is ignored by the command interpreter. These are small words such as “a”, “the”, and “to”. The purpose of noise words is to allow the user to enter commands like “go to the south” and have the command interpreter treat this the same as “go south”.

This change should be implemented by having the CommandInterpreter ignore any noise words, that is, Words in the noiseWords VocabTerm, encountered in the input.

After adding noise words you should be able to enter commands like "go to the south" or "look at room" and get the same result as "go south" or "look", respectively.

Order of Development

There is fair amount of work to do for this assignment, probably too much to do all at once. Below I give one possible order for approaching the work. I have tried to indicate in

HardCodedGame.java which parts of the file correspond to each of these items. At any stage you can comment out those parts of HardCodedGame that are not yet relevant to what you are doing.

1. Implement enough of the command interpreter to get the user input and recognize the Word objects corresponding to the Strings the user input. Simply output the recognized Words to test this.
2. Add VocabTerms and Actions. This should give you enough to do the "kill" and "quit" commands, which do not require any rooms, paths, etc.
3. Now add Rooms, Paths, and the Player. This should allow you to move between rooms.
4. Finally add noise words to the command interpreter. (This could also be done as step 3.)

Hints

1. You will need a number of source files to complete this. My solution had 21. (Seven new ones in addition to the 14 that I supplied.) That, plus or minus a couple, is about the right number. Eclipse, which I recommend you use, will make it easy for you to deal with this.
2. When you read the *HardCodedGame* code you will see code that looks like this:

```
builder.makeAction(vQuit, null, new ActionMethod() {
      @Override
      public void doAction(Word w1, Word w2) {
            interp.setExit(true);
      }
});
```

   This code is creating an *anonymous class* that implements *ActionMethod*, creating a single instance of that class and passing that instance as an argument to *makeAction*. The *doAction* method of *ActionMethod* is overridden in the anonymous class.

   Your *Action* class will need to call the *doAction* method of the anonymous class instance when the Action is to be executed. Note: This is the standard way in Java to have an object which is a function.

   Anonymous classes are very useful when you want to create “one of a kind” classes with a single instance. This feature was added in Java 1.5. See the Java language documentation or *Java in a Nutshell, 5th edition* for more on this. *HardCodedGame* is the only use of anonymous classes in my version. I did use a *local class*, that is, a class defined inside a class or method, in one case.

3. The constructor to CommandInterp needs to take parameters (*BufferedReader in, PrintStream out*). Both BufferedReader and PrintStream are classes in java.io.

   You can get the next line of input using the readLine method on *in*. Something like

```
String inputLine = in.readLine();
```

   If inputLine is null, that's an end of file. If you pass inputLine to the canonicalCommand function in GameUtil, that will convert the line to a list of Strings. Your command interpreter can then attempt to find the corresponding Word for each String.

   Interesting methods on PrintStreams are print(x) and println(x) which will output x and output x followed by a newline, respectively, and printf(...) which will build complex

messages. I used printf in a couple of places in HardCodedGame. Here's the documentation on the printf method:

http://docs.oracle.com/javase/7/docs/api/java/io/PrintStream.html#printf%28java.lang.String,%20java.lang.Object...%29

Follow the link to "Format String Syntax" to get all the gory (and they are!) details on how to do format strings.

4. You are free to create additional classes over and above the ones I've described here.
5. The easiest way to get the assignment started in Eclipse (if you choose to use Eclipse) is:
    a) If you have not already done so, open Eclipse and create a new Workspace in some appropriate location. (An appropriate location is one where you can find it to work on it later.)
    b) Once you have your workspace, Create a new project named something like "Assign3".
    c) Within the project there will be a src folder. Right click on the src folder and create a new Package named "cs345name".
    d) Right click on the package (within src) and click on "Import ...". This will open the import dialog.
    e) In the import dialog, under "General", click on "File System".
    f) Browse to the directory where the Java source I provided you is located, and select that directory.
    g) Select all the .java files, and click "Finish". This will copy the files into the cs345name package in the workspace.
    h) Finally, right click on the package and select Refactor/Rename … and change the package name to cs345<<your id>>, where <<yourid>> is your CS department account id. You can ignore the error message about how changing the package name may break external scripts.
6. Note that *HardCodedGame* does not make any provision to save any of Words, VocabTerms, Actions, Rooms, or Paths anywhere. It is up to you to do whatever you need to do in that regard in order to make your game work.
7. Warning messages: When you are first writing your code, you will have a lot of warnings about something or other not being used. Don't worry about that, as you make use of things, the warnings will go away.

    But, when you reach the point where you think the code is almost complete, you shouldn't have any warnings. Warnings at this point are indicators of potential problems, or, at best, stuff that's hanging around that is unused. This is why many software shops mandate that software that you're making available to other developers should compile without warnings.

    Also, the compiler will warn that an attribute is unused until you do something with it. Assigning it a value doesn't count as "doing something". So, if you initialize an attribute in a constructor, you may still get the warning message. If you create a getter for the attribute, that will count as "doing something."
8. The "default package": *cs345name* is not the default package.

In Java, you can have a class (interface, enum, etc.) that is not in any package. You recognize this by a .java file that does not have a package statement at the top. Such a class is said to be in the default package, that is no package.

9. I have supplied a JAR (Java Archive) file, Assign3.jar, also in Assign3.zip, that contains the .class files for my version of the program. You can run this with the command:

```
java -cp Assign3.jar cs345name.Game
```

I have tested this on the Windows desktop in my office and my Mac OS X laptop and it seems to work on both. I have not tested it on Linux.

10. Here's a sample interaction from my version of the program:

```
$ java -cp Assign3.jar cs345name.Game
You are on a balcony facing west, overlooking a beautiful garden. The only exit
from the balcony is behind you.
? go east
You are in the north end of the Big Room. The room extends south from here.
There is an exit to the outside to the west.
? go out
You are on a balcony facing west, overlooking a beautiful garden. The only exit
from the balcony is behind you.
? go in
You are in the north end of the Big Room. The room extends south from here.
There is an exit to the outside to the west.
? go south
You are in the south end of the Big Room. The room extends north from here.
There is a door in the east wall.
? go n
You are in the north end of the Big Room. The room extends south from here.
There is an exit to the outside to the west.
? go s
I have more than one way to interpret s.
? go souo
I don't understand souo.
? go to tha sou room
You are in the south end of the Big Room. The room extends north from here.
There is a door in the east wall.
? go magic
I don't know how to "go magic".
? magi
I don't understand magi.
? magic
You are in the magic workshop. There are no doors in any of the walls.
? magic
You are in the south end of the Big Room. The room extends north from here.
There is a door in the east wall.
? look
You are in the south end of the Big Room. The room extends north from here.
There is a door in the east wall.
? look around
You are in the south end of the Big Room. The room extends north from here.
There is a door in the east wall.
? kill
What exactly is it you want me to kill?
? kill gold
How exactly do you propose that I kill the gold?
```

```
? quit
Goodbye.
```

What to Submit

This assignment is due by midnight, May 13. Make a zip file with all your source. (If you are using Eclipse, just zip the contents of the `src` directory from your project.)

Also, include a text file that shows the results of your test(s) of the version of the game that you are submitting.

Submit that zip file on moodle.

Grading

I will update the assignment with the grading criteria.