211hw10_S11

Homework 10: SimLaundry 2010

Due Wednesday, 13 April 2010 at 11:00 A.M.

Preface

This assignment has a long description but the coding involved is straightforward. Most of the code for the full application has been written as support code by the course staff. In our solution, the remaining code that you must write (excluding test code) consists of approximately 250 lines (including comments and whitespace lines).

Overview

Rice student J. H. Acker has decided to drop out of school and become a high-tech billionaire by marketing a virtual reality game based on Acker's own personal hygiene. The game is called SimLaundry 2010, and it models the laundry habits of a typical college student. In this assignment, you will help Acker create this game, in return for a cut of the profits and a good Comp 211 grade.

We will assume there are only three types of clothing: shirts, pants, and socks. (We will all enjoy this assignment a lot more if we don't have to think about Acker's underwear.) Acker neatly stacks clean shirts, pants, and socks in *separate* piles on a shelf in his closet.

When changing clothing, Acker throws dirty clothing onto a pile in the corner of the closet, then selects the top clean item of a particular type from the closet shelf; the resulting outfits rarely coordinate, but Acker is no slave to fashion. If there are no clean clothes of a particular variety, Acker resorts to using dirty laundry and removes the *least* recently worn article of that type from dirty laundry pile, smells it, and always decides it can be worn again after all. (Acker never has to go naked, because there is at least one item of the desired type in the laundry, namely the one Acker just removed.)

When doing laundry, Acker removes **fifteen** (`StudentEnvironment.MAX_LOAD`) or fewer, if the pile isn't that large, items from the top of the dirty clothes pile. In the simulation, a load of clothes is laundered and dried instantaneously and placed on a table for clean clothes reserved for Acker in the laundry room. Acker changes clothes so infrequently that the washing and drying time is negligible, so our simulation is a good approximation. The garments in each load of clean clothes are piled in exactly the same order they appeared in the dirty pile. Acker fills the washer and dryer so full that the clothing doesn't get jumbled up.

Eventually Acker retrieves the oldest clean laundry load, folds it, and places it on the closet shelf. In the process, he reverses the order of the clothing within the load; whatever was on the bottom of the pile on the laundry table is now on top of the appropriate pile (shirts, pants, or socks) of clean clothes on the shelf. Hence, if a blue shirt was on top of a white one in the dirty clothes pile and they are washed in the same load, then the white shirt will be on top of the blue one on the closet shelf.

Acker periodically receives gifts of clothing from relatives, which are placed on top of the appropriate pile on the closet shelf. He *never* buys any clothes.

Acker never discards clothing, no matter how threadbare, but does, on rare occasions, lose some. While Acker does not lose clothes being worn, but they can be lost from anywhere else, including the closet shelf, the dirty laundry pile, and the laundry room.

For the purposes of this assignment, a pair of socks is an indivisible article of clothing; we make the unrealistic assumptions that single socks are never lost and that Acker does not wear mismatched socks. Also, you needn't be any more concerned than Acker is about separating white and dark laundry or other such niceties.

What You Must Do

Download the supplied code here: HW10.zip

The course staff is providing a framework for writing this program that includes many classes and interfaces. The framework is packaged as a zipped DrJava project. This file will unzip into a self-contained file tree with root directory `HW10`. This directory contains the following:

- `hw10.drjava` -- the DrJava project file for this assignment. Everything should already be set properly in it. Just start DrJava and go to the `Project/Open...` menu item to open it.
- `src` folder - the root of the source tree. The packages are `edu.rice.comp211.laundry.XXX`, so the actual Java files are quite a ways down. This is the "Project Root" of the DrJava project.
- `classes` folder- the root of the compiled code tree. This is this is the "Build Directory" of the DrJava project.
- `data` folder -- Test files ("`xxxIn.txt`) and expected output files (`xxxOut.txt`) that you can use to test your work. This is the "Working Directory" of the DrJava project.
- `doc` folder -- The Javadocs for the project. Open the `index.html` file in your browser to see the Javadocs

After unzipping the `laundry.zip` file, you can open the DrJava laundry project by starting DrJava, setting the `Language Level` to `Full Java`, **pulling down the `Project` menu and selecting the `Open` command**. In the file chooser that pops up, select the project profile file `laundry.drjava` embedded in the file in the unzipped file tree for `laundry.zip`. You can save the project state at any point during a DrJava session using the `Save` command in the `Project` menu. You can also save individual files within the project using the `Save` button on command file or the `File` menu.

Your assignment is to fill in the stubbed out members of the `DoCommandVisitor` All required areas in the code are clearly marked with comments for the student to complete the code in that area. In the process you may choose to define some new classes to support your `DoCommandVisitor` class implementation. The `Student` class which repeatedly invokes `DoCommandVisitor` models the laundry habits of Acker. In our test simulations, we will typically only create a single instance of `Student` representing Acker, but your code should support multiple students (e.g., Acker and his brothers) at a time. Since these students do not interact with each other, supporting this form of multiplicity is a trivial consequence of OO coding style used in the framework.

The `String` return value of the cases of the `DoCommandVisitor` is the output of processing that `Command`. For example, if the expected output of processing a `Change` command is "`doffed blue shirt, donned red shirt`" then that is the `String` object that you should return from the appropriate case when processing that `Change` command object, i.e. it `accepts` your `DoCommandVisitor` visitor object.

Running the Simulation

GUI Mode

The supplied DrJava project is set up to run the GUI version of the simulation. Thus, clicking on the `Run Project` button in DrJava will run the GUI version of the simulation (`edu.rice.comp211.laundry.ui.SimLaundry2010Application`), which enables you to load test files and either step through them one command at a time or run all the commands automatically. The output will appear in the Interactions pane.

The GUI version also allows you to run individual commands. Select the "`GUI`" mode from the drop list in the lower left corner of the window. The GUI will then allow you to create and run one command at a time. This is useful for testing a single command, but may be more tedious than running in text mode (see below).

The "`Threaded`" mode allows you to simulate multiple students sharing laundry piles. Each student runs a difffernt set of commands from individually specified input files. There appears to be a latent threading bug that has nothing to do with the student code that can pop up occasionally and crash the simulation. Running multiple students is not a requirement here and nothing in the student-written code would affect this, so don't worry about running in `Threaded` mode, though it is kind of interesting to watch.

Text Mode

You may find it easier to run the simulation as a textual input application. Simply right-click the {edu.rice.comp211.laundry.Main.java}} file and select "`Run File`" or change the the DrJava project properties to change the `Main Class` to this classand then use the `Run Project` button.

In the text mode, you simply type in each command, such as "`receive blue socks`" into the input box that will appear in the Interactions pane. See below for more information on the command format. The output will show immediately below in the Interactions pane and new input box will appear.

Note: `adjectives` MUST be <u>SINGLE</u> words.

Testing

Complete testing of your `DoCommandVisitor` cases is required.

Test each case by creating a test cases that call each case explicitly with known host and `Command` parameters. Then proceed to test the visitor as a whole.

See sample laundry test class `edu.rice.comp211.laundry.tests.LaundryTest` for examples of how to test an entire command both manually and by using an input test file and comparison output file.

The `Test Project` button in DrJava runs all of the JUnit test files in the project.

Use the test files in the `data` folder as guides for inputs and expected outputs. For example, given the text input in `sampleIn.txt`, your program should generate the text in `sampleOut.txt`. `testIn.txt` is a fairly extensive test. You'll probably want to start with something smaller such as `sampleIn.txt` or `tinyIn.txt` though.

Initially, the provided framework should compile but `LaundryTest` will fail because most of the members in the key class `DoCommandVisitor` have been stubbed out.

Important Note: *When the simulation begins, Acker is wearing white pants, white socks, and a white shirt. The closet shelf, dirty laundry pile, and laundry facilities are all initially empty.*

Assume that the supplied test files are NOT exhaustive!! You are responsible for the complete testing of your code!

Development Process Recommendation

It is highly recommended that you take a step-by-step, highly structured approach to this assignment. Take SMALL steps, *testing thoroughly* before moving to the next step.

Write and test the easiest cases of `DoCommandVisitor` first and then move on to the harder cases. In the opinion of the staff, the case in order from easiest to hardest are approximately (there is definitely room for argument here!)

1. `forOutfit`
2. `forReceive`
3. `forFold`
4. `forLaunder`
5. `forLose`
6. `forChange`

Grading

Your solution will be graded using the textual interface. Graphical interfaces are notoriously difficult to test and all of the graphical interface code is part of our support code anyway. Your correctness and testing scores (which each count 25% of your grade) will be based on how well your implementation of each command complies with the given specifications and on how well you demonstrate this compliance with test cases. You can test your DoCommandVisitor using the same approach given in our `LaundryTest.java` class. These tests use the `simulate` method in `Student` to drive the execution of `DoCommandVisitor`. If you write some utility methods fopr `BiLists` you should separately test these methods. You are NOT responsible for testing any of our support code in `HW10.zip` including the `BiList` class.

A major portion of your grade (35%) will be based on your program style. If you write your code in the OO style practiced in this course, you should do very well on this aspect of the assignment. The remaining 15% of your grade is based on your documentation, particularly your `javadoc` comments for classes and methods.

Delegation Model Programming vs. Imperative Programming

This assignment will require that you write code that is a mixture of delegation model programming, where one delegates from one object to another to achieve the desired processing, and imperative programming where a defined control flow is set up, such as with `if-else` or {while}} loop constructs and the objects are processed by following this constructed control flow and processing information extracted from the relevant objects. This mixture is how you can expect a real world program to be constructed -- the world isn't so clean that only one paradigm can totally rule in any situation.

The best approach is to default to a delegation mode, looking to create your processing algorithms as a delegation chain from one object to the next. However, you will discover that the nature of certain objects, the `BiList` in particular, will force you into an imperative mode where you will have to use the `BiList`'s iterators in conjunction with conditionals and loops. Effeciency concerns (see below) will also play a role in which programming style you are using at any given moment.

Remember the mantra of delegation model programming: *If your process depends on the type of an object, then delegate to that object.* Visitors are expressly designed for exactly this sort of type-dependent delegation model processing.

Use imperative programming sparingly -- only if and when you absolutely need it!

You are not at all required to do this, but as a benchmark, it should be noted that the staff solution implements each case of `DoCommandVisitor` as a single `return` statement. This should tell you something about the power of delegation in this assignment.

Form of Event Commands

Your program executes a loop that repeatedly reads input from an input "process" that returns `Command` objects. The input process (provided by our supporting framework) reads a series of event description commands, one to a line, either from the console or from a file. The input process converts a stream of characters to `Command` objects which are passed to your program.

In addition to performing the specified command, your program should output a brief description of for each command that it performs in the *exact format described below*. In the following list of commands, the *output* line specifies what your program should print.

- The command

  ```
  receive <adjective>  <article>
  ```

  means Acker received a gift of the specified article (<adjective> <article>) of clothing. In response, the simulation outputs

  ```
  received <adjective> <article>
  ```

  and updates the state of the `StudentEnvironment`. For example,

  ```
  receive argyle socks
  ```

  generates

  ```
  received argyle socks
  ```

  and adds the `argyle socks` to the top of the `socks` pile on the shelf.

- The command

```
lose <adjective> <article>
```

means Acker misplaced the specified article of clothing. If the item exists and Acker is not wearing it, the simulation outputs

```
lost <adjective> <article>
```

and updates the state of the `StudentEnvironment` accordingly. If Acker is wearing it, the simulation outputs

```
Acker is wearing <adjective> <article>
```

and leaves the `StudentEnvironment` unchanged. If the item *does not exist*, the simulation outputs

```
<adjective> <article> does not exist
```

and leaves the `StudentEnvironment` (i.e. Acker) unchanged.

- The command

```
change <article>
```

means Acker doffed the specified article of clothing, discarding it in the dirty laundry pile, and donned a replacement article using the protocol described above. In response, the simulation outputs

```
doffed <adjective> <article>, donned <adjective> <article>
```

describing the article doffed and the article donned.
If Acker has no clean garment of the specified type, the status string returned should indicate this. For instance, suppose Acker was asked to change his pants when he has no clean pants and is already wearing `black-ink-grunge pants`:

```
Nothing to change into! Doffed black-ink-grunge pants, donned black-ink-grunge pants
```

- The command

```
launder
```

means Acker washed and dried a load of laundry. If the dirty clothes pile is not empty, the simulation outputs

```
washed <adjective> <article>, ..., <adjective> <article>
```

listing the clothes in the order they were removed from the dirty clothes pile. If the dirty clothes pile is empty, the simulation outputs

```
nothing to wash
```

- The command

```
fold
```

means Acker retrieved a load of laundry, folded it, and put it on the closet shelf. If a load of laundry is available, the simulation outputs

```
folded <adjective> <article>, ..., <adjective> <article>
```

for the oldest unfolded load. List the clothes in the order they are placed on the shelf. Hence the top garment on the shelf should be the last one listed. If no load of laundry has been washed and dried, then the simulation outputs

```
nothing to fold
```

If the oldest load is empty (because all items in it were lost), the simulation outputs

```
folded empty load
```

- The command

```
outfit
```

asks "what is Acker wearing?" The simulation outputs

```
wearing <adjective> <shirt>, <adjective> pants, <adjective> socks
```

Click here for Supporting Code and Programming Details

Efficiency

For this assignment, you should be concerned about relevant asymptotic efficiency. Choose the simplest representation that yields good performance on inputs of plausible size.

Changing an article of clothing should take constant time (*i.e.*, no searching should be done) provided there's an appropriate garment on the shelf. If the shelf contains no clothing of that type, then in the common case we expect to find one of those near the bottom of the pile, no matter how big the pile is: make that case fast. Infrequent operations need not be particularly fast, because they have little impact on the running time of the entire system. (Suppose one operation accounts for 5% of the running time, and we can make it run 10 times as fast. How does that compare to making an operation that accounts for 25% of the running time twice as fast?)

Example

With Acker initially wearing white shirt, socks, and pants, given the input:

```
receive blue socks
receive green pants
receive red shirt
change socks
receive yellow shirt
change shirt
outfit
change socks
launder
change pants
fold
change socks
```

your program should produce:

```
received blue socks
received green pants
received red shirt
doffed white socks, donned blue socks
received yellow shirt
doffed white shirt, donned yellow shirt
wearing yellow shirt, white pants, blue socks
doffed blue socks, donned white socks
washed blue socks, white shirt
doffed white pants, donned green pants
folded blue socks, white shirt
doffed white socks, donned blue socks
```

The sample input and output files `tinyIn.txt` and `tinyOut.txt` are a good starting point for testing your program but they are far from exhaustive.

You are responsible for testing your own program.

Supplemental Program Running Information

The program starts execution using the special method `public static void main(String[] args)` in class `Main`. The `main` method interface is the only vehicle for executing Java programs directly from the command line. (DrJava has a `main` method for this reason.)

Since your class containing `main` is called `edu.rice.comp211.laundry.Main`, you can enter the line

```
java edu.rice.comp211.laundry.Main -t <infile>
```

in the DrJava Interactions Pane to run the program on input from file `<infile>`. Output will be displayed in the DrJava console. If you simply hit the `Run Project` button, this action is equivalent to entering the line

```
java edu.rice.comp211.laundry.Main
```

which runs the program with terminal input as the input file. In this case, the program will prompt you for the input of each command in a box within the Interactions Pane.

You can also run the program from the command line (a terminal) in the `laundry` directory of the program file tree.
The input line

```
java edu.rice.comp211.laundry.Main -t <infile>
```

should produce exactly the same results as executing the same line in the DrJava Interactions Pane. You can redirect the output to a the file `<outfile>` by typing

```
java edu.rice.comp211.laundry.Main -t <infile> > <outfile>
```

DrJava does not support output redirection but you can copy the text printed in the console and paste it into a file using your editor of choice.