211hw10

Homework 10: SimLaundry 2010

Due Friday, 9 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** (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. Not only does Acker 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

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** laundry.zip. This file will unzip into a self-contained file tree with root directory `laundry`. This directory contains the source tree for the laundry framework with root directory `edu`, a DrJava project file `laundry.drjava` and two text files `sampleIn` and `sampleOut` used by the sample laundry test class `edu.rice.comp211.laundry.tests.LaundryTest`. Given the text input in `sampleIn`, your program should generate the text in `sampleOut`. The provided framework compiles but `LaundryTest` will fail because most of the members in the key class `DoCommandVisitor` have been stubbed out.

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.

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

Your assignment is to fill in the stubbed out members of the `DoCommandVisitor` (members with degenerate bodies (either `return;` or `return null;`). 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 `Student` class includes:

- the name of the student,
- the closet shelf with its piles of clean clothes,

- the dirty laundry pile, and
- the laundry room with its piles of laundered garmets sitting on tables.
- and methods to manipulate those data representations to perform the specified simulation

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. 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.)

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 `laundry.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.

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` 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.

- 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
```

Supporting Code and Programming Details

Our supporting framework is provided in the zip file laundry.zip. The test input in the file `sampleIn` (with corresponding output) `sampleOut` is far from comprehensive.

All of our supporting code is included in the unzipped project. Each file resides in a package that is identified by a `package` statement at the beginning of the file. Most support classes are `public` so that they can be accessed anywhere. Classes without a visibility modifier have "default" visibility, which means that they can only be accessed from classes within the same package.

Our supporting framework includes an input processor that reads event commands from the input stream and returns high level data representations for these commands. The input processor can also print debugging output describing the state of your simulation before each command is performed. To communicate with your code, the input processor uses four interfaces:

`IOProcess` which describes the visible methods supported by the input processor;

`StudentEnvironment` which describes methods for inspecting the state of Acker's environment;

`EnumI` which describes methods for inspecting (but not mutating!) lists within Acker's environment; and

`ReadIteratorI` which includes methods for moving a cursor through lists implementing the `EnumI` interface.

The interfaces are already defined in the framework provided by the course staff.

The input processor class `TerminalIO` implements the `IOProcess` interface. You are welcome to inspect the code of `TerminalIO` but it relies heavily on the Java I/O library, particularly the class `StreamTokenizer` . To understand this code, you will need to read Chapter 11 of JLS (or similar reference). The framework also includes implementations of `EnumI` and `ReadIteratorI` as part of a `BiList` (mutable circular doubly linked list) class implementation.

The `Student` class implements the `StudentEnvironment` interface, which includes the `simulate` method supporting the laundry simulation. The simulate method contains a loop that reads commands from its `IOProcess` and invokes `DoCommandVisitor` on each command. Within the `DoCommandVistor` class you must implement methods that process the various possible commands.

The program includes two class definitions defining unions (composites without recursion): `Garment`, specifying the representation of garments that appear in the input stream, and `Command`, specifying the representation of event description commands. Both classes include the hooks required to support the visitor pattern. The data definition for `Garment` is important because the graphical version of the user interface included in the framework animates the state of your implementation before each command. This graphical user interface (GUI) expects the garments that appear as elements in lists (as revealed by the `EnumI` and `ReadIteratorI` interfaces) to be instances of the `Garment` class. Hence, you must use the representation of garments that our class `Garment` provides.

The file `DoCommandVisitor.java` contains comments describing all of the members that you need to write. This class could have been defined as an inner class of `Student` but we made it a top level class to simplify debugging.

The `IOProcess` interface includes a method `PrintStream open(StudentEnvironment a, boolean debug)` which initializes an `IOProcess` object for a laundry simulation of the specified environment and returns the `PrintStream` object to be used for terminal output. (Up to now you have implicitly used the `PrintStream` object `System.out`.) The `PrintStream` method `println(String s)` prints the string `s` followed by a newline character to the `PrintStream`. The `boolean debug` argument indicates whether or not debugging output should be produced. The `IOProcess` interface also includes a method `nextCommand` which reads the next command from the input channel supported by the `IOProcess` object.

Each call on `nextCommand` returns the next command in the stream provided by the `IOProcess` object, until it reaches an end-of-file (`<control>-d` from the keyboard). End-of-file is reported as a null reference of type `Command`.

The `nextCommand` method in `TerminalIO` processes character strings consisting of words separated by ``space'' characters such as `' '` and `'\n'` . A `word` is any sequence of printable characters other than space, `'\n'` (newline), and `'\r'` . (return). An `adjective` must be a single word. An `article` must be one the words `shirt` , `pants` , or `socks` . The same adjective, say `argyle` may be applied to garments of different types, but there are no duplicate items of clothing.

The program passes a boolean debug `flag` to (`TerminalIO`). The value of the flag is true iff the command line argument `-d` or `-debug` is passed to `main` .

The Graphical User Interface (GUI)

The initialization of the GUI creates an Acker Student object and associated DoCommandVisitor. Each GUI event triggers the execution of DoCommandVisitor; in some cases, such as reading input from a file, it triggers the execution of DoCommandVisitor on a stream of Commands. In essence, the event-handling loop built-in to the Java Swing framework is used to drive the computation rather than a separate loop in the main thread such as the one in the simulate method in Student.

The GUI could also have been written as an implementation of the IOProcess interface. This approach, which conforms to the classic "model-view-controller" (MVC) pattern, is more flexible because it decouples the GUI (the view in MVC terminology) from the model, but it is also more complex because it involves the cooperative execution of
two loops in separate threads--a main program loop in the simulate method of Student and the loop driving the event-handling thread supporting the processing of GUI inputs. The SimLaundryApplication dispenses with the main program loop by absorbing the application (the model) into the GUI (the view).

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` and `tinyout` are a good starting point for testing your program but they are far from exhaustive.

You are responsible for testing your own program. 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.