hw10details_S11

HW10 Code Details

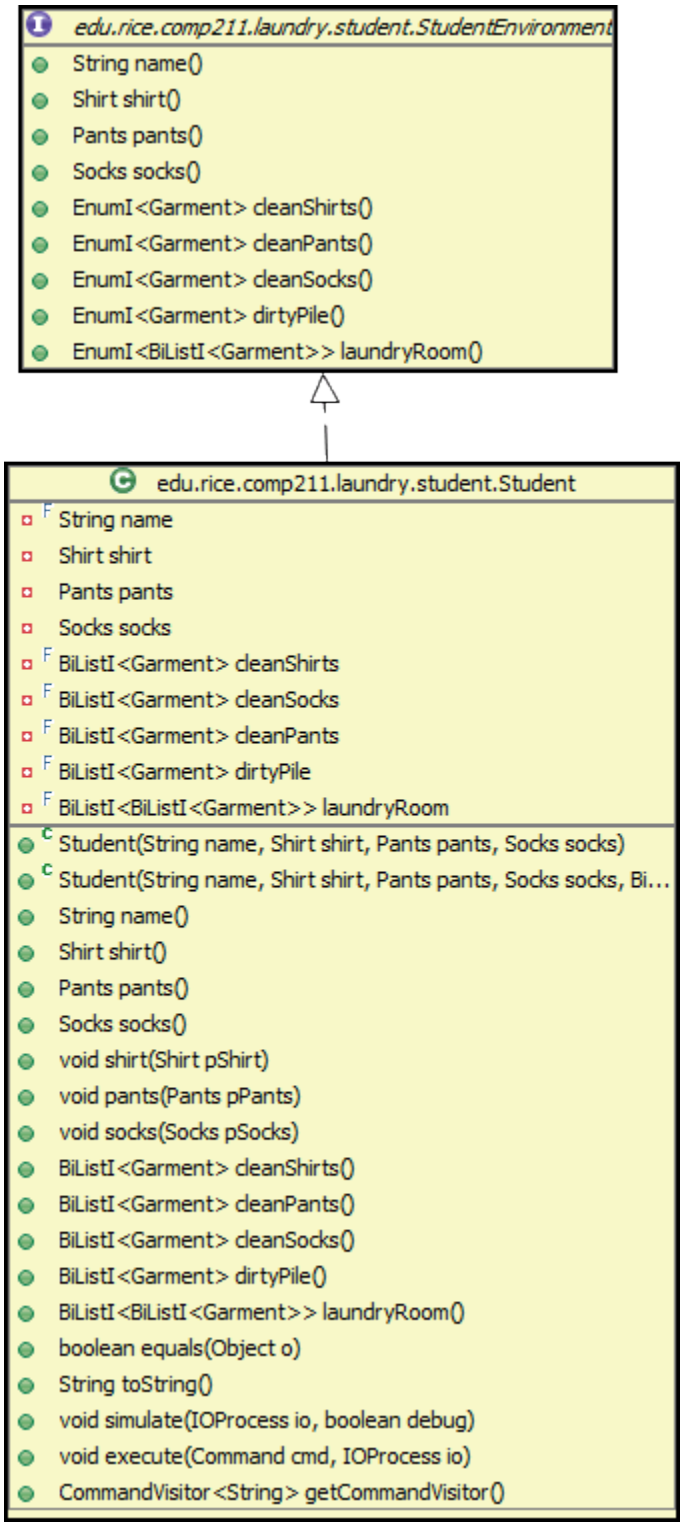
- Most of the supplied code is support code that you do not have to worry about. You are certainly free to use whatever you wish, but only the classes listed here should be needed.
- You WILL need to make anonymous inner classes and possibly some named classes if you feel you need them. The staff's solution only uses anonymous inner classes, so it definitely can be done that way.
- In viewing the UML diagrams below, keep in mind that a class or interface always inherits all of the methods of its parents, i.e. everything above it. Just because a class or interface only shows 2 methods in its little box in the UML diagram doesn't mean that's all the methods it has--look at all the methods in its superclasses/superinterfaces too!
- Don't forget to `import` a package before attempting to use any of its classes.
- Check the Javadoc comments of each class to learn more about what each class does.

Student Package

A student is wearing a shirt, pants and socks. The student also has references to several lists (ordered piles) of garments: clean shirts, clean pants, clean socks, a dirty clothes pile and a laundry room which is a list of list of garments because it represents the collection of multiple loads of laundered garments that have been cleaned, but not yet folded and returned to the individual clean garment piles.

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



Garment Package

Garments are of 4 types:

1. `Shirt` - represents shirts of various types
2. `Pants` - represents pants of various types
3. `Socks` - represents socks of various types
4. `NullGarment` - represents the absence of a garment, such as one might get on a failed search for a garment.

All `Garments` have an "`adjective`" field that is used to differentiate different instances of `Shirts` and `Pants` and `Socks`, e.g. a "blue" Shirt vs. a "red" Shirt.

`Garments` support a visitor, `GarmentVisitor`, which has cases for each type of visitor. **NEVER test for the type of a `Garment` object!** **Always delegate to it** by having the garment object accept a visitor whose different cases provide the garment-type-dependent processing you desire.

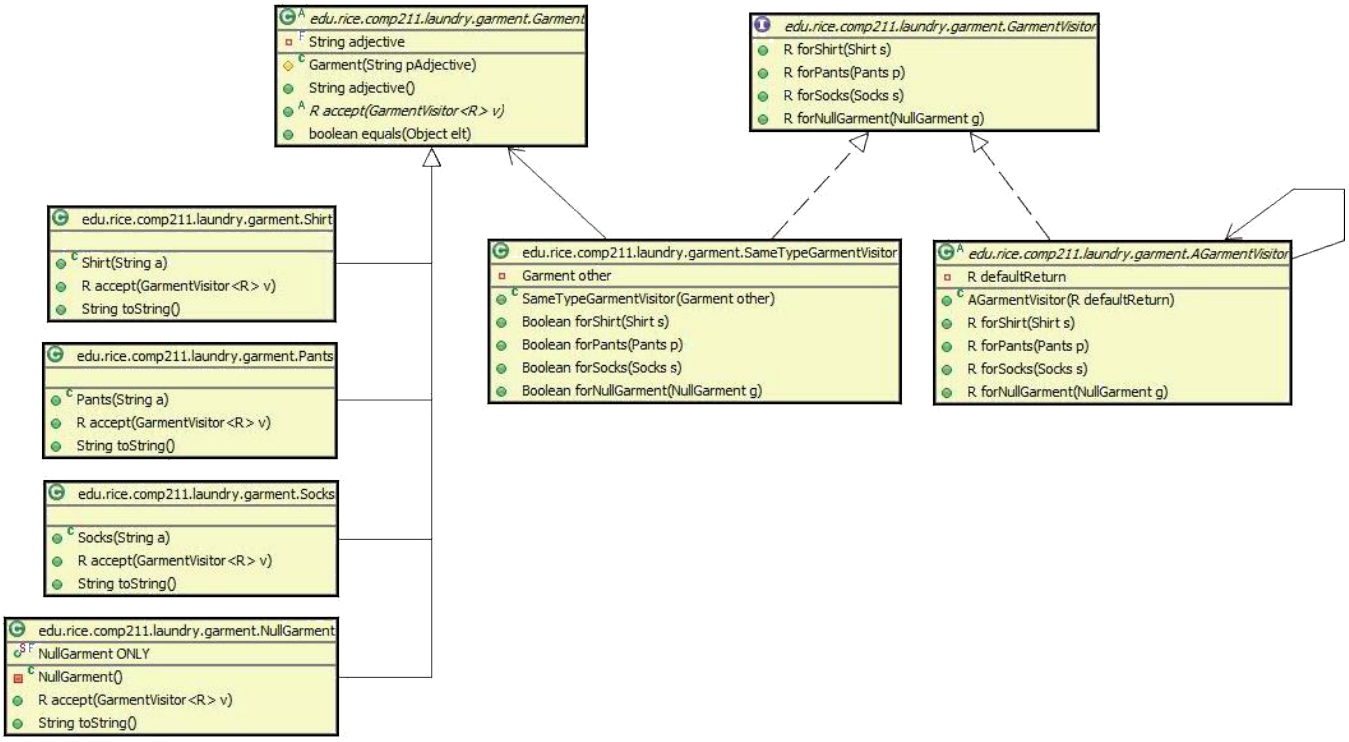
`AGarmentVisitor` is a convenience class that provides concrete default behavior for all the cases. If you only need to specify one or two cases and just return a default value for the rest, create your visitor by sub-classing `AGarmentVisitor` and overriding only the cases you desire. Note that the `AGarmentVisitor`'s constructor requires an input parameter, the default value to use for the non-overridden cases.

Here's an example of using `AGarmentVisitor` to write a little visitor to return either "Not a Sock!" or "Got some socks!" depending on whether the visitor is accepted by a `Socks` object or some other kind of `Garment` object.

```
GarmentVisitor isSockGarmentVisitor = new AGarmentVisitor<String>("Not a Sock!") {
   public String forSocks(Socks sockHost) {
      return "Got some socks!";
   }
}

aShirt.accept(isSockGarmentVisitor) --> "Not a Sock!"
aPants.accept(isSockGarmentVisitor) --> "Not a Sock!"
aNullGarment.accept(isSockGarmentVisitor) --> "Not a Sock!"
aSock.accept(isSockGarmentVisitor) --> "Got some socks!"
```

The `SameTypeGarmentVisitor` is a utility visitor that can be used to return a true or false if two `Garment` objects are the same type, i.e. a `Shirt` and a `Shirt` or a `Pants` and a `Pants`, regardless of their "`adjectives`".



Lists package

`BiLists` are used to represent the various piles of clothes or a collection of piles of clothes (the laundry room).

Using `BiList`:

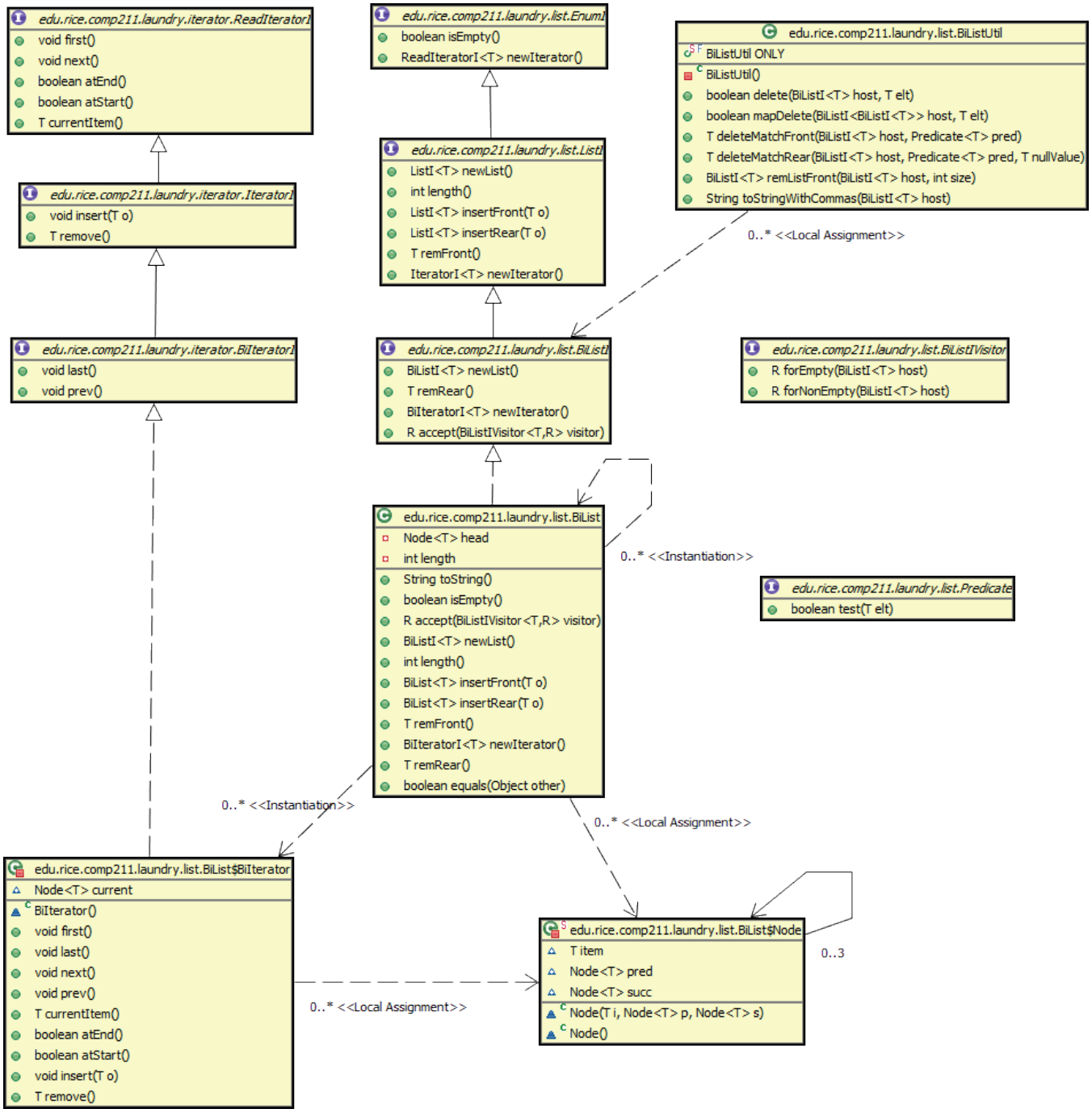
- `BiList<Garment>` is a pile of clothes.
- `insertFront()` and `remFront()` are the methods to insert and remove from the top of the pile respectively.
- `insertRear()` and `remRear()` are the methods to insert and remove from the bottom of the pile respectively.
- `newIterator()` is a factory method that will create an iterator for you specifically for that list, initialized to point at the first (top) element of the list. **Do NOT try to make an iterator by instantiating one yourself!**

- {[BiLists]} (the super-interface BiListI actually) accept BiListIVisitors that can be used to perform operations that depend on whether the list is empty or not. That is, BiListIVisitor has 2 cases, forEmpty and forNonEmpty.
- BiLists also have an isEmpty() method that can be used for imperative/procedural-style conditional processing based on the list's emptiness. Using a delegation-style processing using the BiListIVisitor is recommended however and will result in cleaner code in some cases.

Using the BiIteratorI<Garment> iterator a BiList<Garment> creates:

- The BiList will initialized the iterator to point to the top of the list.
- atEnd() will return true if you are at the end of the list, so your while loop's conditional should have something like "while(!myIterator.atEnd()) {....}"
- Read the data at the current iterator location by using currentItem().
- The iterator, unlike most iterators in the world, supports mutation, so insert() and remove() will mutate the list at the current iterator location. *Be VERY CAREFUL when advancing the iterator after these operations or you will skip elements!!*
- Typically, a call to next() will be used to advance the iterator one element towards the bottom of the pile.
- You can iterate backwards through the list by using the prev() method rather then next(). You'll need to initialize the iterator at the bottom of the pile (end of the list) by first calling the last() method to set the iterator to the end of the list. The loop conditional should be something like !myIterator.atStart().

BiListUtil is a singleton that provides utility methods that you might find useful for various operations on your lists. It is also a handy source of examples on how to use the BiLists and their iterators.



Command package

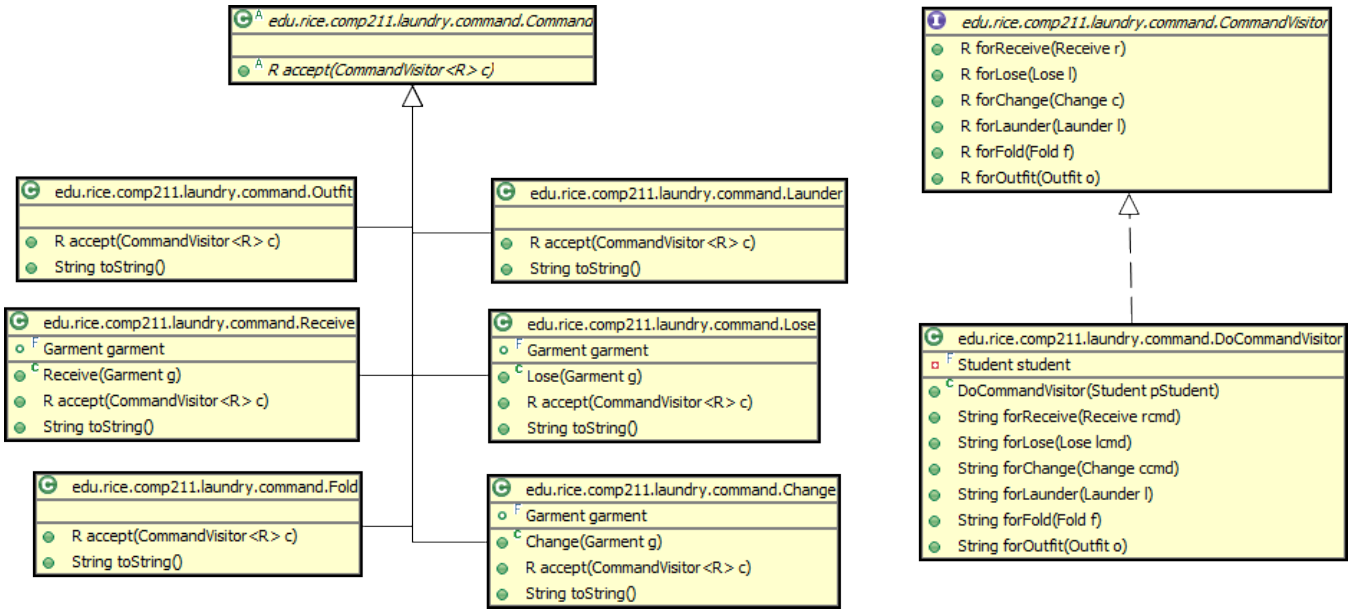
`Command` objects (not to be confused with the Command Design Pattern) are "messages" (in OO parlance) sent to the laundry simulation system to effect particular operations. These operations are described in detail in the previous wiki page. There is one `Command` sub-class corresponding to each type of operation the simulation supports, namely an `Outfit` query, `Receive` a new `Garment` item, `Change` a particular type of garment worn, `Lose` a specific `Garment`, `Launder` the pile of dirty clothes (to with maximum bounds), and `Fold` a pile of clean clothes and return them to their clean piles.

Every `Command` object can accept a `CommandVisitor` which has cases for each possible type of `Command`.

`DoCommandVisitor` is the visitor that you have to write (fill in the missing code) for this assignment. Other than the `student` field, your implementation may have additonal fields and private utility methods that you decide you need for your implementation. *Thus, your `DoComandVistor` may appear slightly different than what is shown here.*

The `student` field in `DoCommandVisitor` will give you access to the various clothes piles, what the student is currently wearing and the laundry room. **Y ou do NOT have to code up the instantiation of the `DoCommandVisitor`!** The supplied simulation infrastructure already does that for you.

A word of explanation: A big deal was made in class and lab about how one should never have `public` fields because they allow uncontrolled mutations to occur. You've problaby noticed the public `garment` field in some of the `Command` subclasses here. For convenience's sake, we're exploiting the fact that `Commands` never need to be mutated (so far as we know). The `garment` field in the `Receive`, `Lose` and `Change` commands is marked `fina` , which makes it immutable. So, we can get away with a `public` field because the field is immutable. That said, in general, using an accessor method to explicitly control access to a class's data is still the preferred technique.



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).

Addtional Technical Information

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 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 `IOProcess` interface includes a method `PrintStream open(StudentEnvironmenta, 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` .