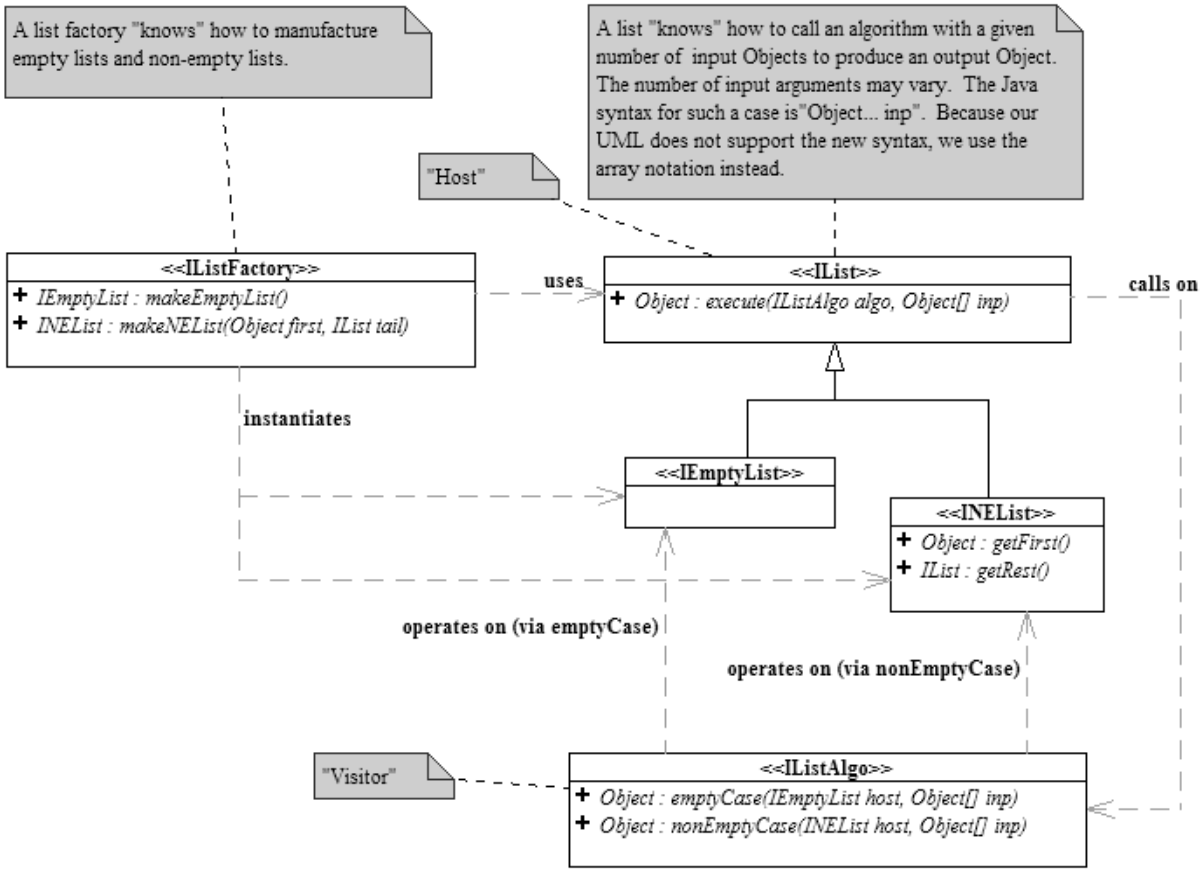
Instructions

1. This exam is conducted under the Rice Honor Code. It is a closed-notes, closed-book exam.
2. Fill in your name on every page of the exam.
3. If you forget the name of a Java class or method, make up a name for it and write a *brief* explanation in the margin.
4. You are expected to know the syntax of defining a class with appropriate fields, methods, and inheritance hierarchy. You will not be penalized on trivial syntax errors, such as missing curly braces, missing semi-colons, etc, but do try to write Java code as syntactically correct as possible. We are more interested in your ability to show us that you understand the concepts than your memorization of syntax!
5. Write your code in the most object-oriented way possible, that is, with the fewest number of control statements and no checking of the states and class types of the objects involved.
6. In all of the questions, feel free to write additional helper methods or visitors to get the job done.
7. Make sure you use the Singleton pattern whenever appropriate. Unless specified otherwise, you do not need to write any code for it. Just write "singleton pattern" as a comment.
8. For each algorithm you are asked to write, 90% of the grade will be for correctness, and 10% will be for efficiency and code clarity.
9. You have two hours and a half to complete the exam.

Please State and Sign your Pledge:

1a) 10 pts	1b) 15 pts	1c) 10 pts	2a) 15 pts	2b) 15 pts	3) 20 pts	4) 15 pts	Total 100 pts

For your convenience, below is the UML class diagram for the scheme list framework with an abstract factory studied in class. You are free to use this list framework without explanation/implementation.



1. The goal is to write an IListAlgo, called ListDuplicates, that returns an IList containing all the elements that appear more than once in the IList host. The resulting IList should not contain any duplicates. For example, if the host IList is (99, 0, 17, 99, 99, 0, 5), then the result is (99, 0) or (0, 99) (the order is immaterial). You are to write this visitor in the following manner.

   a) Write a visitor, called Contains, to check if a given Object is in the host list, as specified in the stub code below. Use the equals method for comparison.

```
public class Contains implements IListAlgo {

    // Student to complete the Singleton pattern.




    /**
     * Returns true if the input object is in the host, false otherwise.
     * @param inp inp[0] is the Object to be found in the host.
     * @return Boolean
     */
    public Object emptyCase(IEmptyList host, Object... inp) {
        // Student to complete




    }


    /**
     * Returns true if the input object is in the host, false otherwise.
     * @param inp inp[0] is the Object to be found in the host.
     * @return Boolean
     */
    public Object nonEmptyCase(INEList host, Object... inp) {
        // Student to complete









    }
}
```

b) Write a visitor, called DupHelp, which first traverses the host list to accumulate two things, 1) the list of elements seen so far in the original list and 2) the list of duplicates built so far, and then returns the complete list of duplicates, as specified in the stub code below.

```
public class DupHelp implements IListAlgo {

    // Student to add appropriate field(s) and constructor(s).




    /**
     * @param inp inp[0] is the list of elements seen so far,
                  inp[1] is the list of duplicates accumulated so far.
     * @return IList the complete list of duplicates.
     */
    public Object emptyCase(IEmptyList host, Object... inp) {
        // Student to complete



    }

    /**
     * @param inp inp[0] is the list of elements seen so far,
                  inp[1] is the list of duplicates accumulated so far.
     * @return IList the complete list of duplicates.
     */
    public Object nonEmptyCase(INEList host, Object... inp) {
        // Student to complete
        // Hint:
        // Check to see if the first of host is in inp[0], the list of elements seen so far;
        // If it's has been seen already then check to see if it is already in inp[1], the
        // accumulated list of duplicates; make appropriate updates to the accumulators
        // and recurse.
        // If it has not been seen, update the seen-so-far list and recurse.









    }
}
```

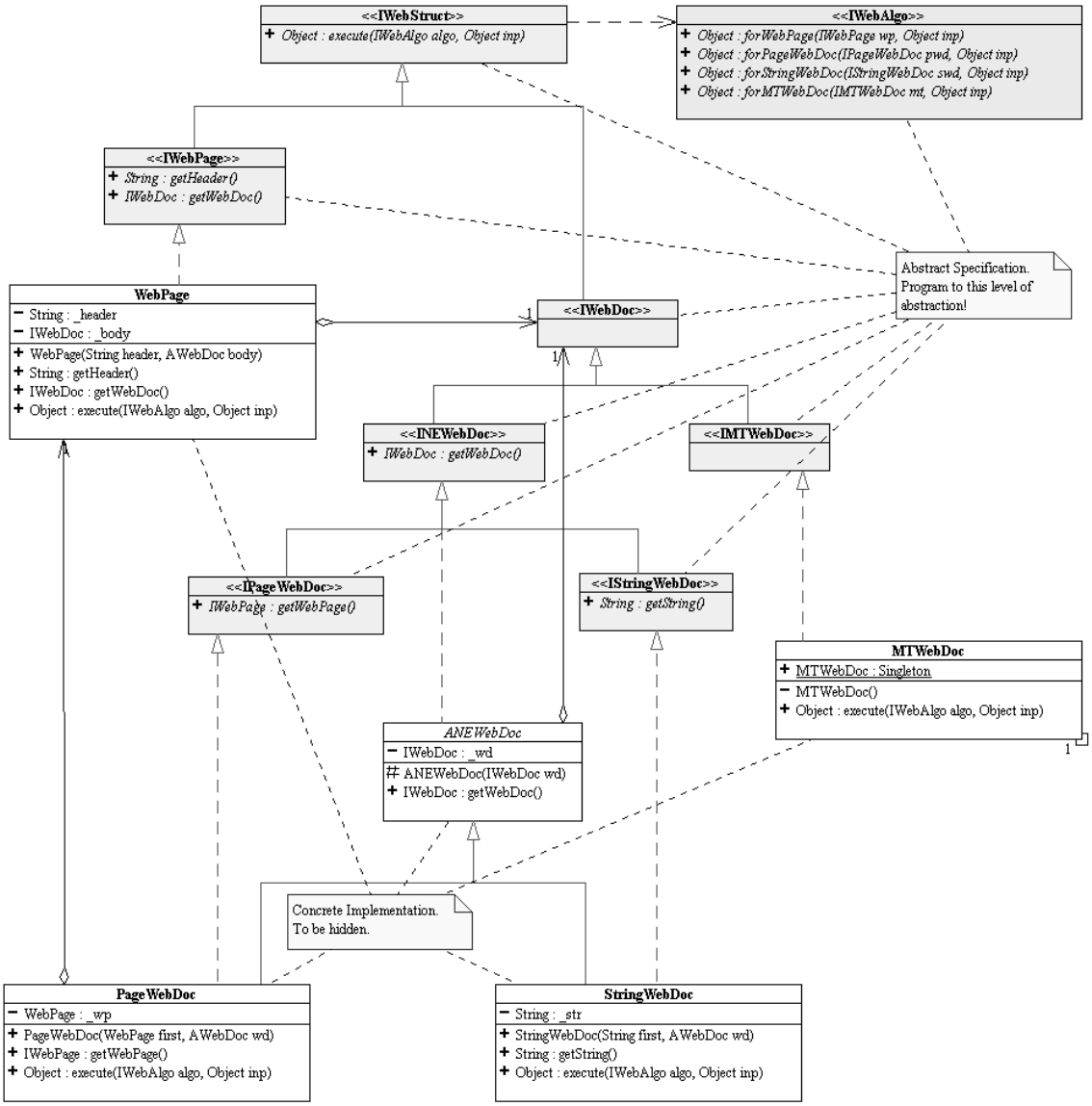
c) Write ListDuplicates using Contains and DupHelp.

```
public class ListDuplicates implements IListAlgo {

    // Student to add appropriate field(s) and constructor(s).









    /**
     * @param nu not used.
     * @return IList the complete list of duplicates.
     */
    public Object emptyCase(IEmptyList host, Object... inp) {
        // Student to complete


    }


    /**
     * @param nu not used.
     * @return IList the complete list of duplicates.
     */
    public Object nonEmptyCase(INEList host, Object... nu) {
        // Student to complete using Contains and DupHelp









    }
}
```

2. Consider the following description of a *recursive immutable* data structure that represents web pages.

   - A web page has a header, which is a String, and a body, which is a web document.
   - A web document may be empty or non-empty. When it is empty, it contains nothing. When it is not empty, it may contain a String and web document, or it may contain a web page and a web document.

   The following UML class diagram represents the object model for the above web page structure. Because of the limitation of our UML tool, we cannot display the variable argument list (varargs), `Object… inp`, for the "execute" method and the visitor methods. Still, you should assume that all these methods have the varargs `Object… inp`.



a) Write an IWebAlgo, called CountWebPages, that counts and returns the number of IWebPage that are embedded in the IWebPage host, including the host itself.

b) Write an IWebAlgo, called FindPage, that takes as input a String s, finds and returns an IWebPage embedded in the host that contains s as its header. If there is no match, returns `null`. Use the equals method for String comparison.

3. Consider the set of Natural Numbers, 1, 2, 3, 4, ..., i.e., the non-negative integers. For this problem, you will complete the design and implementation of a class that represents the set of Natural Numbers in the form of a list, that is, a Natural Number has a value and a successor that is a Natural Number. (The Natural Number's value and successor are analogous to the first and the rest of an IList.).

   A problem is that the set of Natural Numbers is infinite. Thus, our design employs a technique known as ``Lazy Evaluation"; in other words, we do not instantiate the object(s) representing a Natural Number until there is need for it. Furthermore, the object(s) representing a Natural Number should be unique, that is, created once and never recreated. For instance, the object(s) representing the Natural Number 7 should be created only when 7 is required and created one time only. Only the object(s) representing the first Natural Number, 1, is instantiated when your program begins execution.

   To implement lazy evaluation, we apply the state pattern to the design and implementation of the successor. Specifically, a Natural Number's successor is in one of two states, evaluated or unevaluated. If it is evaluated, then the object(s) representing the successor to the current Natural Number is already instantiated. On the other hand, if it is unevaluated, no object(s) yet exist(s). Asking for the unevaluated successor of a Natural Number should instantiate that Natural Number, changing the state of the current Natural Number.

   Please replace the comments below with your code to complete the design and implementation of the class NaturalNumbers.

```
class NaturalNumbers {

    private int _value;

    private interface IState {
        public NaturalNumbers getRest(NaturalNumbers owner);
    }

    private static class EvaluatedState implements IState {
                // Student to complete









    }

    private static class UnevaluatedState implements IState {
                // Student to complete









    }

    private IState _successor;
```

```
    public final static NaturalNumbers One = new NaturalNumbers(1);

    private  NaturalNumbers(int value) {
        // Student to complete




    }

    public final int getValue() {
        // Student to complete




    }

    public final NaturalNumbers getSuccessor() {
        // Student to complete




    }

    private final void setSuccessor(IState successor) {
        // Student to complete




    }
}
```

4. There is an abstract notion of a processor. Processors are manufactured by fabrication plants (fab-plants). A fab-plant takes as input a "version" and creates a processor of that type. A fab-plant needs to remember all processors it creates. A processor has a unique serial number, and remembers its version and its fab-plant of manufacture. A version is the design specification of a given type of processor. It remembers the list of fab-plants that manufacture processors of that version. When a fab-plant starts creating processors of a given version, it "registers" itself with that version. The version then adds this fab-plant to its list of fab-plants.

   Occasionally, "upgrades" are issued for a version. An upgrade consists of a string representing code and a unique identifier. A version takes as input this code to create a new upgrade. An upgrade remembers its version. A version can have multiple upgrades and remembers them all. When a processor of a given version is created by any fab-plant, all upgrades for the corresponding version must be applied to the processor. Similarly, when a new upgrade is issued, all processors of that version produced by all fab-plants need to apply that upgrade.

   Your task is to produce an object-oriented design simulating processors, fab-plants, versions and upgrades. Express your design as Java classes/interfaces with appropriate data fields and methods. Pseudo code in the form of comments is acceptable for the body of the methods. Use inner classes whenever appropriate.

A blank page for writing code…