Comp 212 - Intermediate Programming EXAM #1 February 15, 2006
Rice University - Instructors: Cox & Nguyen NAME: ______________________

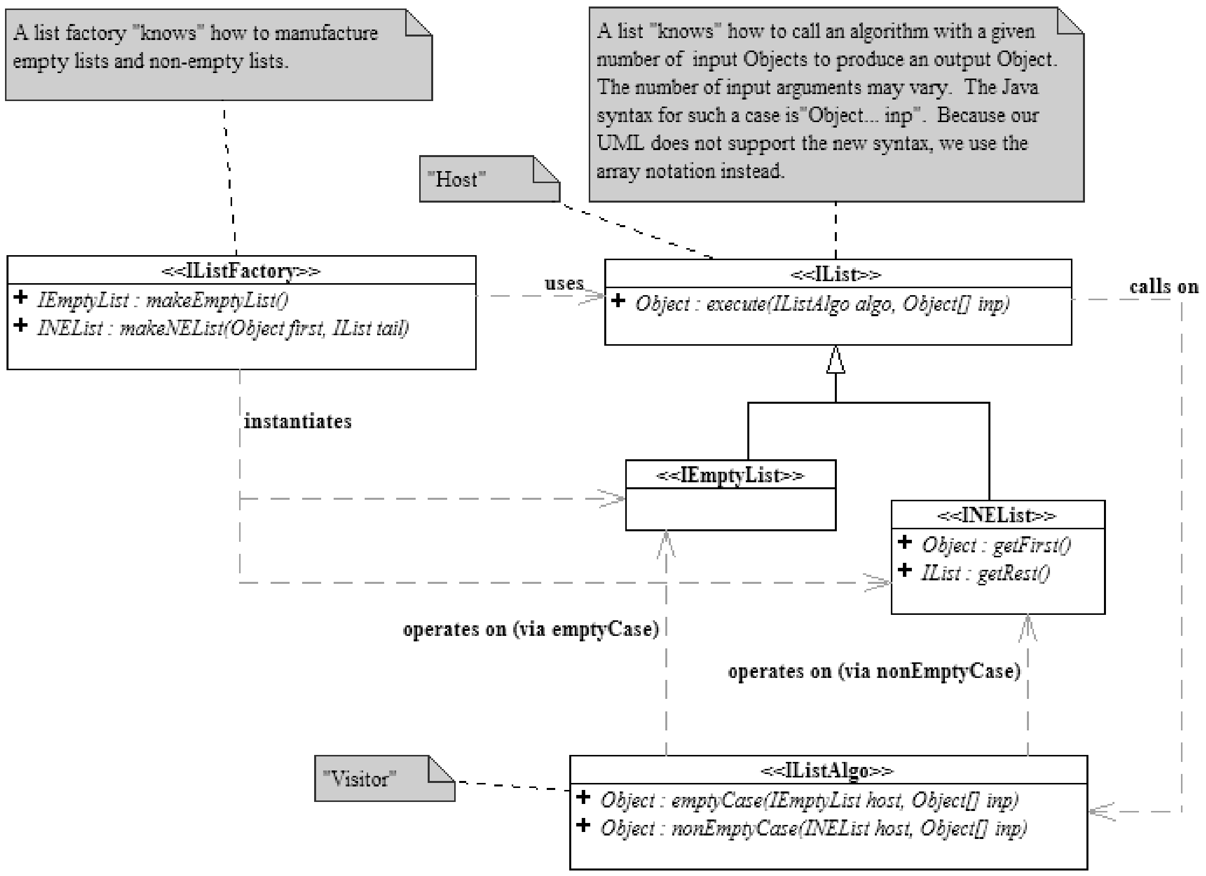
Instructions

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. 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.
8. You have two hours and a half to complete the exam.

Please State and Sign your Pledge:

1) 15 pts	2a) 20 pts	2b) 25 pts	3) 25 pts	4) 15 pts		Total 100 pts

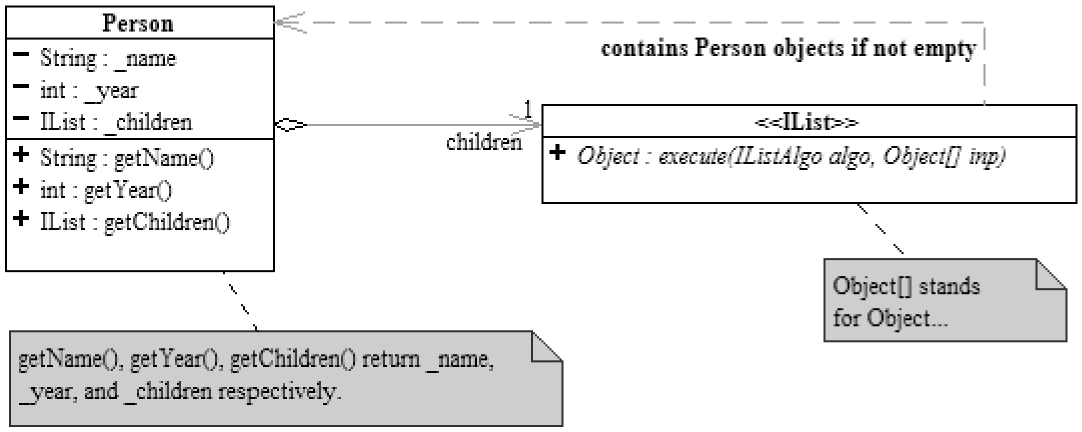
For your convenience, below is the UML class diagram for the scheme list framework with an abstract factory studied in class. You are free to use this list framework without explanation/implementation.


A list factory "knows" how to manufacture empty lists and non-empty lists.
A list "knows" how to call an algorithm with a given number of input Objects to produce an output Object. The number of input arguments may vary. The Java syntax for such a case is "Object... inp". Because our UML does not support the new syntax, we use the array notation instead.
"Host"
<<IListFactory>>
+ IEmptyList : makeEmptyList()
+ INEList : makeNEList(Object first, IList tail)
uses
<<IList>>
+ Object : execute(IListAlgo algo, Object[] inp)
calls on
instantiates
<<IEmptyList>>
<<INEList>>
+ Object : getFirst()
+ IList : getRest()
operates on (via emptyCase)
operates on (via nonEmptyCase)
"Visitor"
<<IListAlgo>>
+ Object : emptyCase(IEmptyList host, Object[] inp)
+ Object : nonEmptyCase(INEList host, Object[] inp)

1. Write an IListAlgo called Concat to concatenate the IList input to the host. The specification of this visitor is shown the stub code below. Complete the code as specified.

```
/**
 * Concatenates (appends) the input list to the end of the host list.
 */
public class ConCat implements IListAlgo {
    private IListFactory _fact;

    /**
     * Initializes _fact to fac.
     */
    public ConCat(IListFactory fac) {
        // STUDENT TO COMPLETE;



    }

    /**
     * Concatenates (appends) the input list to the end of the host list.
     * @param inp[0] an IList.
     * @return IList
     */
    public Object emptyCase(IEmptyList host, Object... inp) {
        // STUDENT TO COMPLETE;




    }

    /**
     * Concatenates (appends) the input list to the end of the host list.
     * @param inp[0] an IList.
     * @return INEList
     */
    public Object nonEmptyCase(INEList host, Object... inp) {
        // STUDENT TO COMPLETE;





    }
}
```

2. Consider the following data definition of objects of type Person. A Person has a name, a birth year and a list of children, each of whom is a Person. The list of children may or may not be empty. Thus a Person is like a family tree where his/her children, the children of the children, etc represent all the descendants of this Person. The following is the UML class diagram that models Person. It is immutable by design and thus cannot have any circular reference. Note that Object[] stands for the var arg Object... in the diagram.


Person
− String : _name
− int : _year
− IList : _children
+ String : getName()
+ int : getYear()
+ IList : getChildren()
contains Person objects if not empty
1
children
<<IList>>
+ Object : execute(IListAlgo algo, Object[] inp)
Object[] stands for Object...
getName(), getYear(), getChildren() return _name, _year, and _children respectively.

a/ Write an IListAlgo that takes an IList of Person as the host list and as input parameter a Person whose children list is the host list itself and returns the youngest Person found in the family tree of the input Person. In the case there are more than one Person objects with the same birth year, return any one of them. In the case the Person has no children, then return the Person himself/herself. The specification of this visitor is shown the stub code below. Complete the code as specified.

```
/**
 * Gets the youngest Person from the descendants tree of the given Person.
 */
public class GetYoungest implements IListAlgo {
    //STUDENT TO ENTER SINGLETON PATTERN CODE HERE.







    /**
     * @param kids the children list of person[0].
     * @param person person[0] is Person whose children list is kids.
     */
    public Object emptyCase(IEmptyList kids, Object... person) {
        // STUDENT TO COMPLETE






    }
    /**
     * @param kids the children list of person[0].
     * @param person person[0] is Person whose children list is kids.
     */
    public Object nonEmptyCase(INEList kids, Object... person) {
        // STUDENT TO COMPLETE









    }
}
```

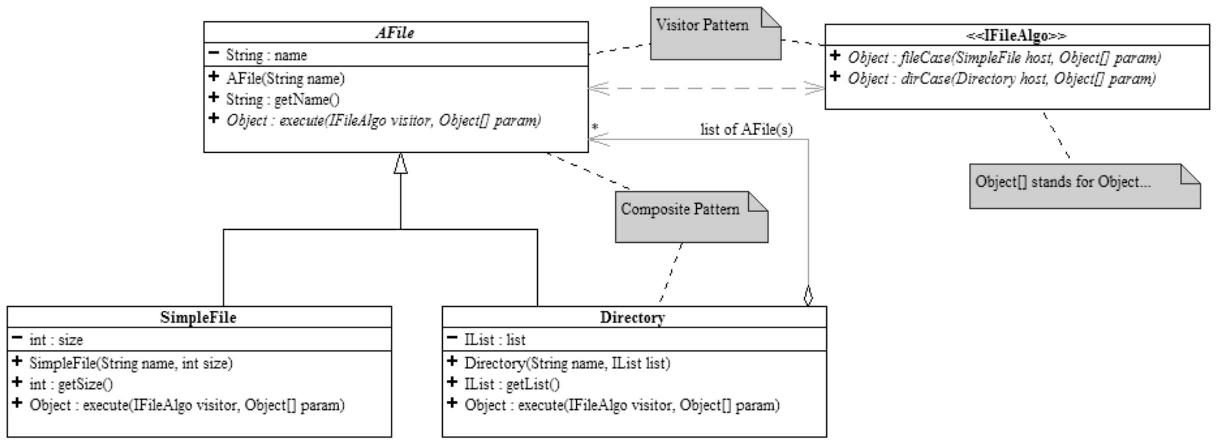
b/ Write an IListAlgo that takes an IList of Person as the host list and as input parameter a Person whose children list is the host list itself and returns an IList of all the Persons with no children in the family tree. In the case the Person has no children, then return the list consisting of the Person himself/herself. The specification of this visitor is shown the stub code below. Complete the code as specified. *Add helpers if needed.*

```
/**
 * Builds the list of Person(s) that have no children from the given input Person.
 */
public class GetNoHeirList implements IListAlgo {
    private IListFactory _fact;  // concrete factory to make lists.
    public GetNoHeirList(IListFactory fac) {
        _fact = fac;
    }

    /**
     * @param kids the children list of person[0].
     * @param person person[0] is Person whose children list is kids.
     */
    public Object emptyCase(IEmptyList kids, Object... person) {
        // STUDENT TO COMPLETE


    }
    /**
     * @param kids the children list of person[0].
     * @param person person[0] is Person whose children list is kids.
     */
    public Object nonEmptyCase(INEList kids, Object... person) {
        // STUDENT TO COMPLETE
         // HINT: Use a helper.






    }
}
```

3. A file system similar to the one used in Unix can be modeled as follows.

- A file has a name.
- A simple file is a file with a size that is an int.
- A directory is a file that contains a list of files. This list may or may not be empty.

Below is the UML class diagram representing the above file system. It is immutable by design and as a result cannot have circular references. Note that Object[] stands for the var arg Object... in the diagram.


AFile
− String : name
+ AFile(String name)
+ String : getName()
+ Object : execute(IFileAlgo visitor, Object[] param)
Visitor Pattern
<<IFileAlgo>>
+ Object : fileCase(SimpleFile host, Object[] param)
+ Object : dirCase(Directory host, Object[] param)
*
list of AFile(s)
Object[] stands for Object...
Composite Pattern
SimpleFile
− int : size
+ SimpleFile(String name, int size)
+ int : getSize()
+ Object : execute(IFileAlgo visitor, Object[] param)
Directory
− IList : list
+ Directory(String name, IList list)
+ IList : getList()
+ Object : execute(IFileAlgo visitor, Object[] param)

The above design dictates that in order to process an AFile, we need to write an appropriate IFileAlgo. And when processing a Directory (IFileAlgo.dirCase), it is very likely that we need to write an appropriate IListAlgo to process the list of files inside the Directory.

Write an IFileAlgo that adds up the file size of all the SimpleFiles in the AFile host. The specification of this visitor is shown the stub code below. Complete the code as specified.

```java
/**
 * Adds all the sizes of all SimpleFile in the host AFile.
 */
public class TotalFileSize implements IFileAlgo {
    public static final TotalFileSize Singleton = new TotalFileSize();
    private TotalFileSize() {
    }

    /**
     * Return the size of this SimpleFile.
     * @param host the SimpleFile that is the host
     * @param nu not used
     * @return size of the SimpleFile host
     */
    public Object fileCase(SimpleFile host, Object... nu) {
        // FOR STUDENT TO DO


    }


    /**
     * Return the total file size of all SimpleFiles contained in this directory
     *(and any of its subdirectories).
     * @param host the Directory that is the host
     * @param nu not used
     * @return total file size of all SimpleFiles contained in this directory
     * and its subdirs
     */
    public Object dirCase(Directory host, Object... nu) {
        // FOR STUDENT TO DO





    }
}


/**
 * IList visitor to add all the file sizes in the host list containing AFile(s).
 */
class AddFileSizes implements IListAlgo {
    public static final AddFileSizes Singleton = new AddFileSizes();
    private AddFileSizes() {
    }

    /**
     * No files, nothing there...
     * @param lof an empty list of files
     * @param nu not used
     */
    public Object emptyCase(IEmptyList lof, Object... nu) {
        // FOR STUDENTS DO TO




    }

    /**
     * This gets called if the list is non-empty. The first is an AFile.
     * Add the total file size of the first to the total file size of the rest
     * of this list.
     * @param lof a list containing AFiles.
     * @param nu not used
     */
    public Object nonEmptyCase(INEList lof, Object... nu) {
        // FOR STUDENTS TO DO



    }
}
```

4. Consider the following description of a mutable recursive data structure that represents a node in a computer network:

- A node is either a terminal node (like a computer or a printer) or a communication node (like a router or a switch).
- A terminal node has a String representing its IP address.
- A communication node has a String representing its IP address and a list of the nodes that it is connected to.

Design the system so that you can perform any kind of algorithm on the node without changing the code of the node classes. You may use data structure frameworks you have seen in class.

Note: In those algorithms, you only distinguish between terminal nodes and communication nodes: abstractly they treat computers the same as printers, and routers the same as switches.

Draw the UML class diagram for the design above. Identify the design patterns you use. NO coding is required.

Note: If you use a data structure framework from class, you do not need to include all the framework's classes in your UML diagram. If you use LRStruct, for example, just draw a box with "LRStruct" and nothing else in it.

A blank page for writing code…