Comp 212 - Intermediate Programming EXAM #1 February 14, 2007
Rice University - Instructors: Cox & Nguyen NAME: ______________________

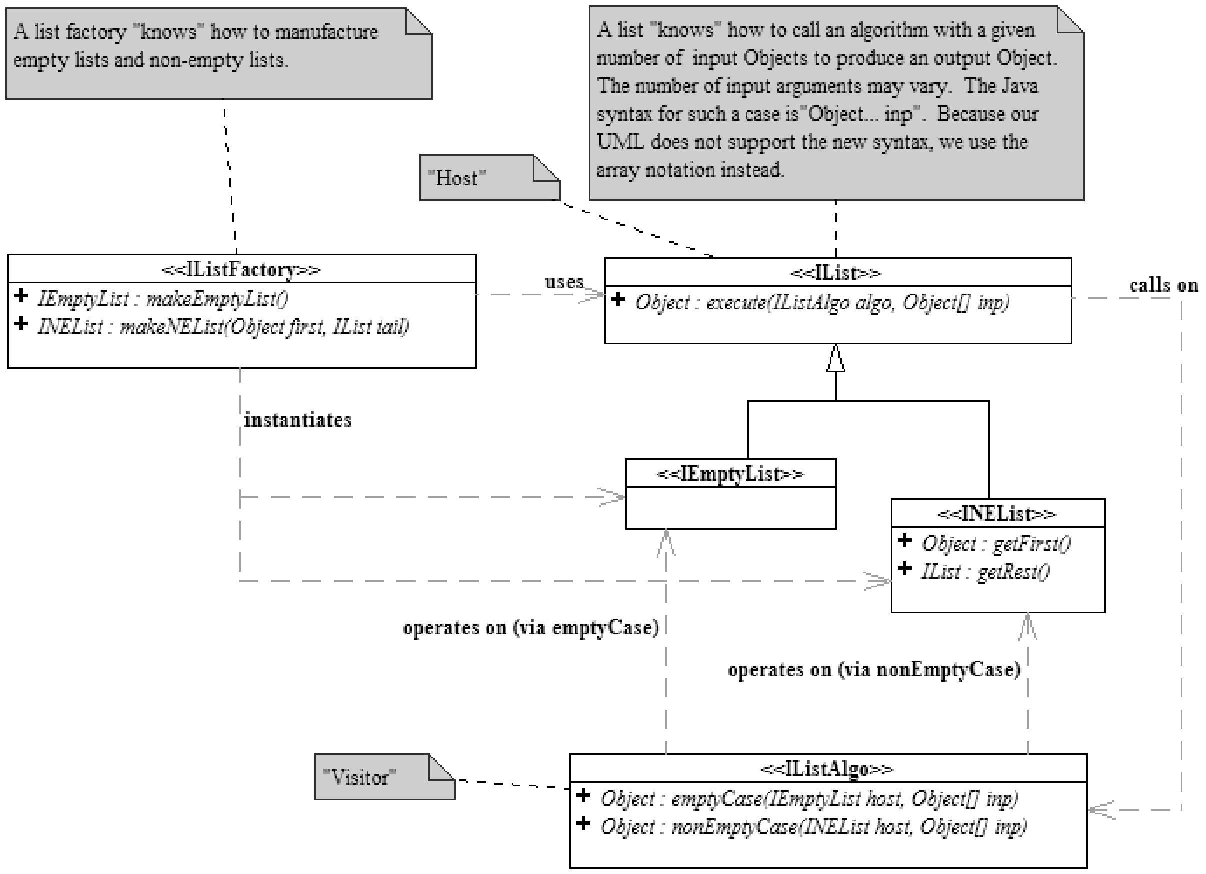
Instructions

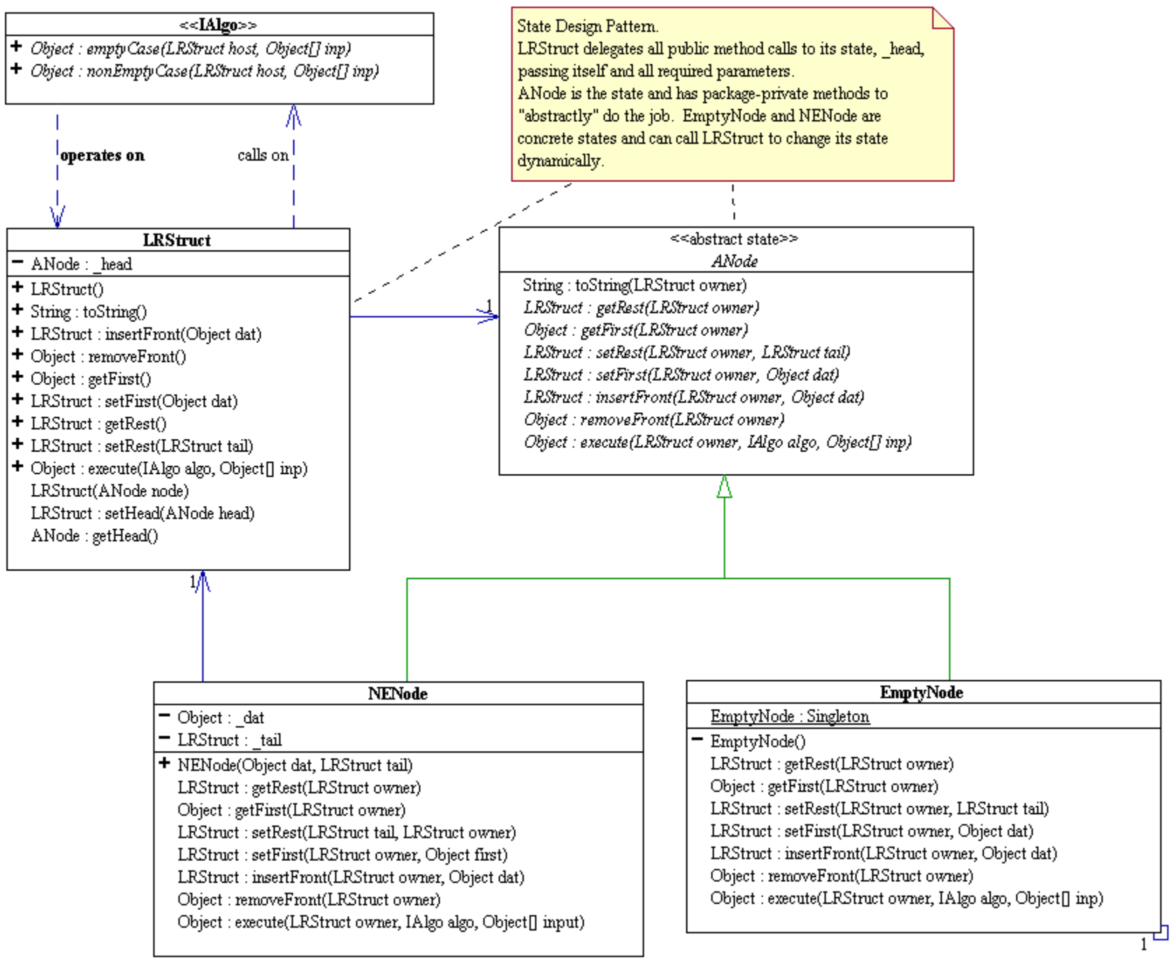
1. This exam is conducted under the Rice Honor Code. It is a closed-notes, closed-book exam.
2. Fill in your name on every page of the exam.
3. If you forget the name of a Java class or method, make up a name for it and write a *brief* explanation in the margin.
4. You are expected to know the syntax 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. You are free (and encouraged) to re-use any of the visitors done in the lectures/labs/homeworks without having to rewrite them. When reusing such a published visitor, just comment that it comes from the lectures or labs or homeworks and describe what it is supposed to do.
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, unless specified otherwise.
9. You have two hours and a half to complete the exam.

Please State and Sign your Pledge:

1) 20 pts	2a) 15 pts	2b) 15 pts	3) 20 pts	4) 30 pts		Total 100 pts

For your convenience, below are the UML class diagrams for the immutable list framework (IList) and the mutable list framework (LRStruct) studied in class. You are free to use these list frameworks without explanation/implementation. Note that for the mutable list framework, you are allowed to call only the public methods of LRStruct and IAlgo.


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)


<<IAlgo>>
+ Object : emptyCase(LRStruct host, Object[] inp)
+ Object : nonEmptyCase(LRStruct host, Object[] inp)
operates on
calls on
State Design Pattern.
LRStruct delegates all public method calls to its state, _head, passing itself and all required parameters.
ANode is the state and has package-private methods to "abstractly" do the job. EmptyNode and NENode are concrete states and can call LRStruct to change its state dynamically.
LRStruct
− ANode : _head
+ LRStruct()
+ String : toString()
+ LRStruct : insertFront(Object dat)
+ Object : removeFront()
+ Object : getFirst()
+ LRStruct : setFirst(Object dat)
+ LRStruct : getRest()
+ LRStruct : setRest(LRStruct tail)
+ Object : execute(IAlgo algo, Object[] inp)
LRStruct(ANode node)
LRStruct : setHead(ANode head)
ANode : getHead()
1
<<abstract state>>
ANode
String : toString(LRStruct owner)
LRStruct : getRest(LRStruct owner)
Object : getFirst(LRStruct owner)
LRStruct : setRest(LRStruct owner, LRStruct tail)
LRStruct : setFirst(LRStruct owner, Object dat)
LRStruct : insertFront(LRStruct owner, Object dat)
Object : removeFront(LRStruct owner)
Object : execute(LRStruct owner, IAlgo algo, Object[] inp)
1
NENode
− Object : _dat
− LRStruct : _tail
+ NENode(Object dat, LRStruct tail)
LRStruct : getRest(LRStruct owner)
Object : getFirst(LRStruct owner)
LRStruct : setRest(LRStruct tail, LRStruct owner)
LRStruct : setFirst(LRStruct owner, Object first)
LRStruct : insertFront(LRStruct owner, Object dat)
Object : removeFront(LRStruct owner)
Object : execute(LRStruct owner, IAlgo algo, Object[] input)
EmptyNode
EmptyNode : Singleton
− EmptyNode()
LRStruct : getRest(LRStruct owner)
Object : getFirst(LRStruct owner)
LRStruct : setRest(LRStruct owner, LRStruct tail)
LRStruct : setFirst(LRStruct owner, Object dat)
LRStruct : insertFront(LRStruct owner, Object dat)
Object : removeFront(LRStruct owner)
Object : execute(LRStruct owner, IAlgo algo, Object[] inp)
1

1. Write an `IListAlgo` called `EveryThird` that returns an `IList` containing every third element of the host list. It should return an empty list in the case of an empty host list. The following table illustrates a few of the desired results.

IList Host	**Returned IList**
() – the empty list	() – the empty list
(1)	(1)
(1, 2)	(1)
(1, 2, 3)	(1,)
(1, 2, 3, 4)	(1, 4)
(1, 2, 3, 4, 5)	(1, 4)
(1, 2, 3, 4, 5, 6)	(1, 4)
(1, 2, 3, 4, 5, 6, 7)	(1, 4, 7)

Hint: The `EveryThird` algorithm can be viewed as a generalization of the `EvenIndexed` and `OddIndexed` algorithms discussed in class. Below is the code for these two algorithms.

Note: Although the following code does not use factories, your solution to this problem should.

```java
/**
 * Extracts the even-indexed elements of a list.
*/
public class EvenIndexed implements IListAlgo {

    public static final EvenIndexed Singleton = new EvenIndexed();
    private EvenIndexed() {
    }

    public Object emptyCase(IEmptyList host, Object... nu) {
        return host;
    }

    public Object nonEmptyCase(INEList host, Object... nu) {
        return new NEList(host.getFirst(),
                        (IList)host.getRest().execute(OddIndexed.Singleton));
    }
}

/**
 * Extracts the odd-indexed elements of a list.
*/
public class OddIndexed implements IListAlgo {

    public static final OddIndexed Singleton = new OddIndexed();
    private OddIndexed() {
    }

    public Object emptyCase(IEmptyList host, Object... nu) {
        return host;
    }

    public Object nonEmptyCase(INEList host, Object... nu) {
        return host.getRest().execute(EvenIndexed.Singleton);
    }
}
```

A blank page for writing code…

2. Write an `IListAlgo` to remove the last element of the host `IList`. Specifically it should return an `IList` that is a copy of the host list without its last element. For an empty list, it should return an empty list.

    - A simple (but inefficient) way to accomplish this task is to reverse the list in some appropriate number of times. Implement this algorithm and name it `RemLast1`. You may use the `Reverse` visitor done in one of your homeworks without having to include its code as part of your solution.

```
public class RemLast1 implements IListAlgo {

    private IListFactory _fact;

    public RemLast1(IListFactory fact) {
        _fact = fact;
    }

    public Object emptyCase(IEmptyList L, Object... nu) {
        // student to complete



    }

    public Object nonEmptyCase(final INEList L, final Object... nu) {
        // student to complete




    }

}
```

- Implement a more efficient algorithm to remove the last element by traversing the host list only once. Name it `RemLast2`.

```
/**
* Note: the IListFactory is a parameter to the vistor methods.
*/
public class RemLast2 implements IListAlgo {

    public static final RemLast2 Singleton = new RemLast2();

    private RemLast2() {
    }

    /**
     * @param fact[0] is an IListFactory.
     */
    public Object emptyCase(IEmptyList L, Object... fact) {
        // student to complete


    }

    /**
     * @param fact[0] is an IListFactory.
     */
    public Object nonEmptyCase(final INEList L, final Object... fact) {
        // student to complete










    }

}
```

3. Write an IAlgo `First2Last` that moves the first element of the host LRStruct to the end of the host LRStruct. Note that the host LRStruct is mutated in this case.

```
public class First2Last implements IAlgo {
    public static final First2Last Singleton = new First2Last ();
    private First2Last () {
    }

    public Object emptyCase(LRStruct host, Object... nu) {
        // student to complete


    }

    public Object nonEmptyCase(final LRStruct host, Object... nu) {
        // student to complete








    }
}
```

4. In this problem you will write a collection of Java classes that model the supervisory relationship between employees in a simple, small company. Specifically, every employee has a name and a unique identification number. Being a small company, the owner is an employee. The owner has a collection of employees who report directly to him or her. However, not every employee reports to the owner. A manager is an employee. A manager has an employee to whom he or she reports ("the manager's manager") and a collection of employees who report directly to him or her ("the team"). A worker is an employee. A worker has an employee to whom he or she reports.

   There are several operations that can be applied to an employee.
   - The simplest are `getName` and `getId`.
   - The operation `getTotal` returns the total number of employees who work for the company. When an employee is hired, he or she is always a worker and must be assigned an identification number and manager. When an employee quits or is fired, the number of total employers should be reduced, but his or her ID should never be reused.
   - A more complex operation is `promote`. When `promote` is applied to a worker, the worker becomes a manager, with a new manager and a team. When `promote` is applied to a manager, the manager remains a manager, but the manager's manager and team change. Thus promote takes two parameters: one for the new manager and one for the new team. It is an error to apply `promote` to the owner.
   - `getAllWorkers` is a method that returns a list of all non-manager workers who report directly and or indirectly to the receiver.

   In the space below write all of the classes that constitute your model. You must define all of the above methods in appropriate classes, implement them all, and be sure to define any fields required by these methods.

A blank page for writing code…