Comp 212 - Intermediate Programming EXAM #1 February 13, 2002
Rice University - Instructors: Cox & Nguyen NAME: ______________________

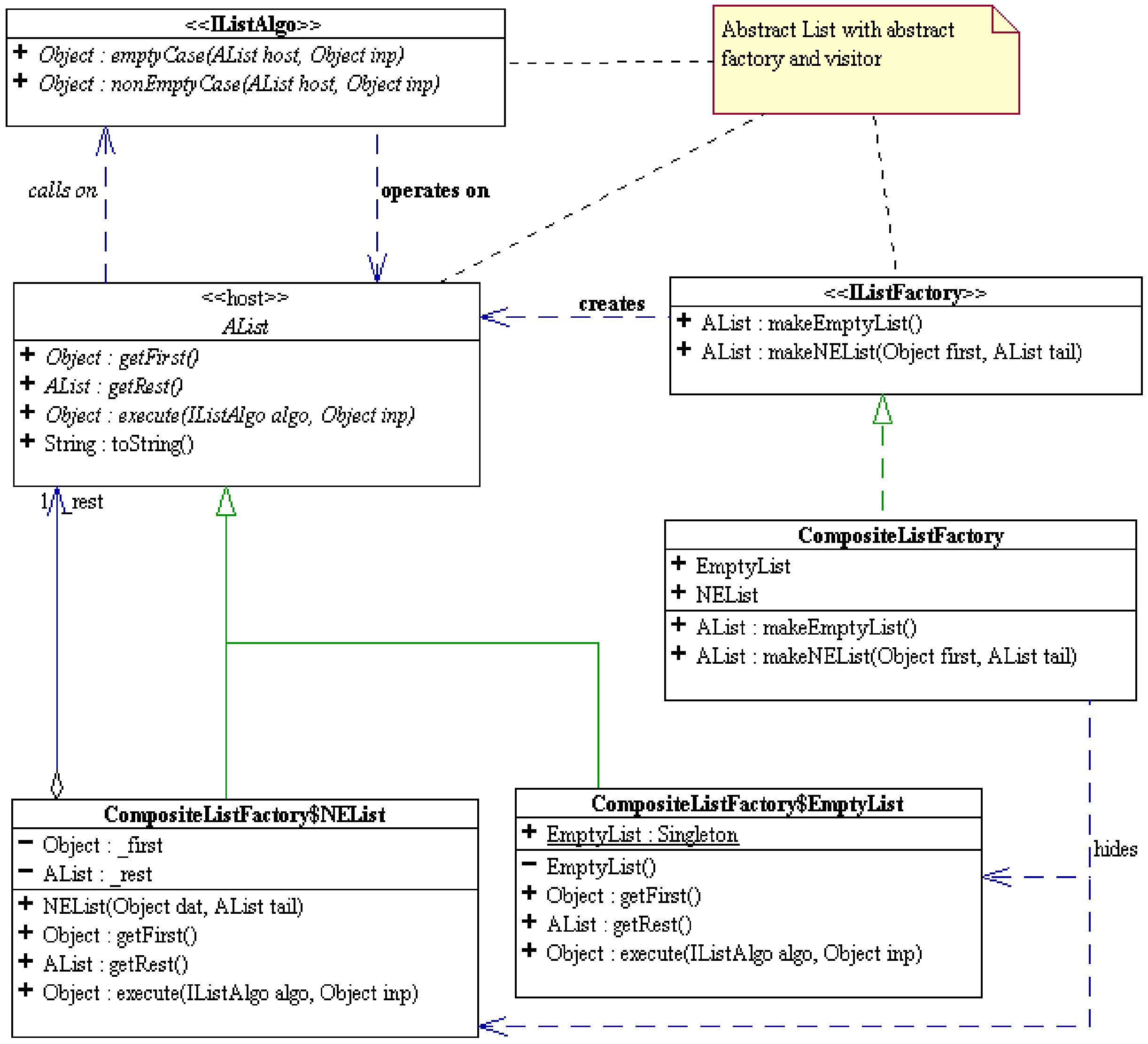
Instructions

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! If you don't remember the code, write your thoughts down in prose.
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 may use as helpers the following visitors from the lectures and the homeworks: GetLength, GetMin, GetSum, ToString, Reverse, MakeClone, LastElement, GetNth, FirstNElements, ConcatWith, without explanation/implementation.
10. You have two hours and a half to complete the exam.

Please State and Sign your Pledge:

1a) 10	1b) 10	2a) 10	2b) 15	3a) 10	3b) 10	4) 20	5) 15	TOTAL 100

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.


Abstract List with abstract factory and visitor
<<IListAlgo>>
+ Object : emptyCase(AList host, Object inp)
+ Object : nonEmptyCase(AList host, Object inp)
calls on
operates on
<<host>>
AList
+ Object : getFirst()
+ AList : getRest()
+ Object : execute(IListAlgo algo, Object inp)
+ String : toString()
creates
<<IListFactory>>
+ AList : makeEmptyList()
+ AList : makeNEList(Object first, AList tail)
1
_rest
CompositeListFactory
+ EmptyList
+ NEList
+ AList : makeEmptyList()
+ AList : makeNEList(Object first, AList tail)
hides
CompositeListFactory$NEList
− Object : _first
− AList : _rest
+ NEList(Object dat, AList tail)
+ Object : getFirst()
+ AList : getRest()
+ Object : execute(IListAlgo algo, Object inp)
CompositeListFactory$EmptyList
+ EmptyList : Singleton
− EmptyList()
+ Object : getFirst()
+ AList : getRest()
+ Object : execute(IListAlgo algo, Object inp)

1. Given an ***AList*** ("list") containing Integer objects and an **Integer** object ("n"), return a new ***AList*** containing only the **Integer** objects from "list" greater than "n".

   a) Implement a solution by adding appropriate method definitions to each of ***AList***, **EmptyList** and **NEList**. You do not have to write out the code for the whole class, just indicate to which class your methods belong. 10 pts

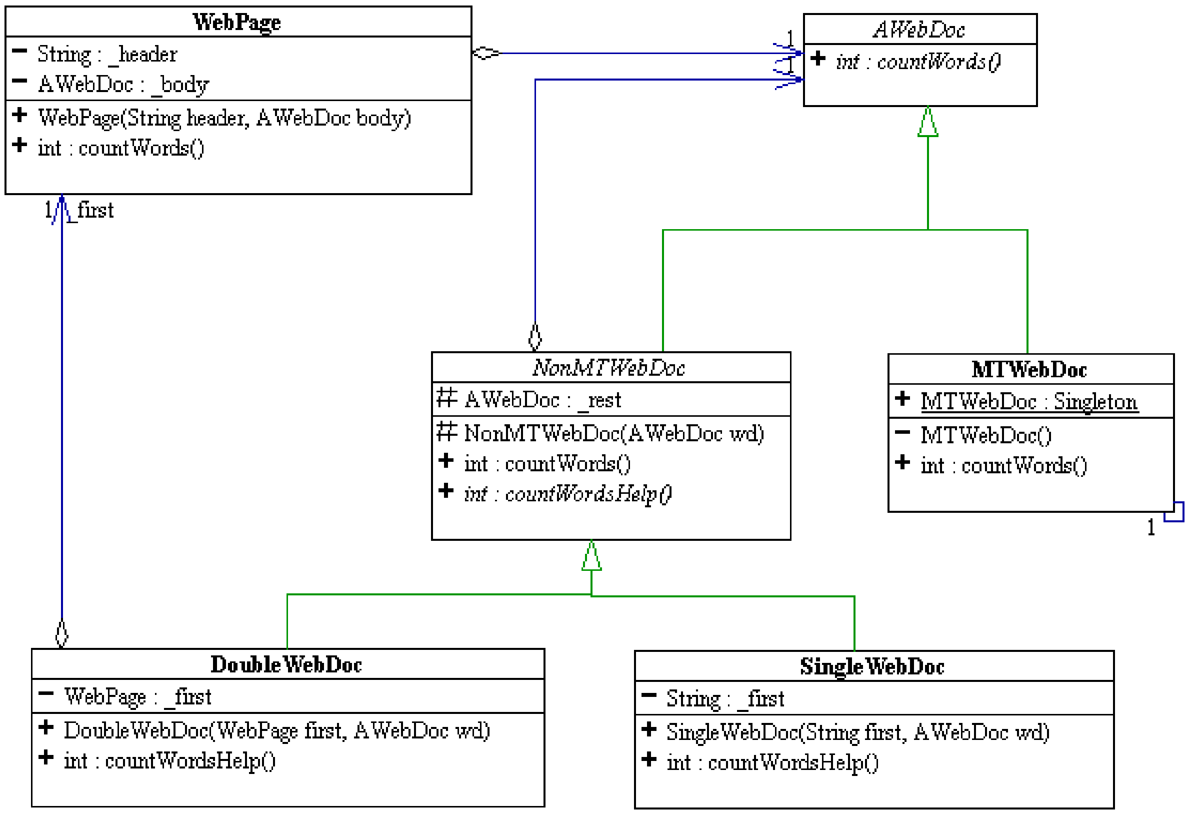
b) Implement a solution using the visitor interface to ***AList***. Use the factory interface in your solution. You do not have to write out the code for the whole class, just indicate to which class your methods belong. 10 pts

2. Given an ***AList*** ("list") containing Integer objects and an Integer object ("n"),

   a) Write a visitor to return **Boolean**.TRUE if there is an **Integer** object in "list" that is equal to "n". Otherwise, return **Boolean**.FALSE. 10 pts

b) Write a visitor to compute and return a new ***Alist*** containing the non-duplicate Integer objects in "list". In other words, no two objects within the new ***AList*** have the same value. Also, use your solution to (a) as a helper and the factory interface for constructing concrete ***AList*** objects. 15 pts

3. Recall the "Tangled Web" problem from Homework 2 that asked you to develop an object model for 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. In addition, the problem asked you to write a method for counting the number of words in a web page. Below, is the UML class diagram for our solution to this problem.



a) Detail the changes necessary to our object model to support a visitor interface, specifically, write the Java code for the interface and abstract and concrete method definitions necessary to support a visitor. You do not have to write out the code for the whole class, just indicate to which class your methods belong. 10 pts

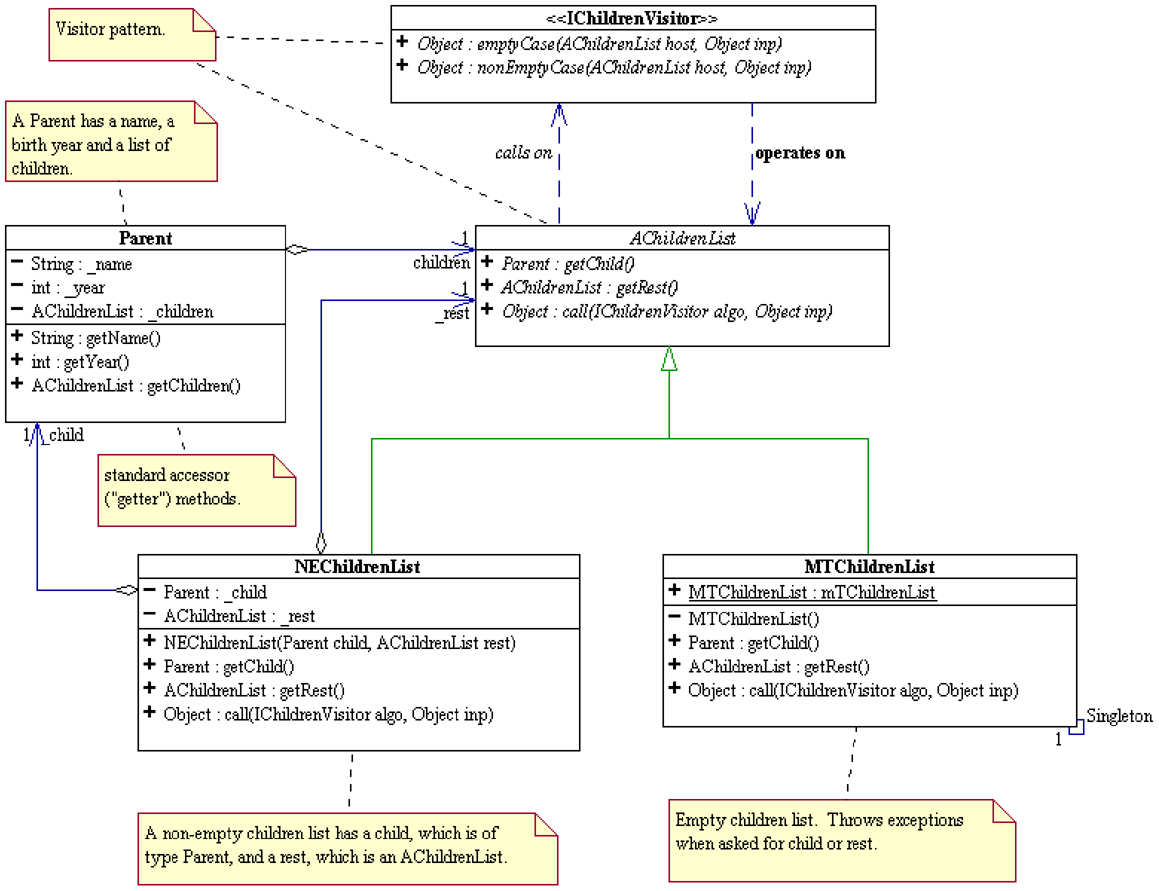
b) Re-implement countWords using your visitor interface. 10 pts

4. Design an object model for the following *immutable* data structure that represents a special type of family trees, one that keeps track of all ancestors of a person. (??? pts total)
   - An ancestor tree may be empty or non-empty.
   - When it is empty, it contains nothing.
   - When it is not empty, it may contain
     - a **String** representing the name of a person,
     - an **int** representing the year the person was born,
     - an ancestor tree for the person's father, and
     - an ancestor tree for the person's mother.

   Your class design should make use of appropriate design patterns to achieve the highest degree of flexibility and extensibility. Do not apply the abstract factory pattern to this problem.

   Draw the corresponding UML class diagrams. The class diagrams should clearly show all the fields with their types, methods with their parameter lists and their return types, and constructors with their parameter lists. The field and method names should be self-explanatory. To save space and time, show only the abstract methods of the super classes but not the concrete implemented methods in the subclasses. Use comment boxes to indicate the various design patterns in the diagram. 20 pts

5. Consider the following UML class diagram for a special type of family trees, one that keeps track of all descendants of a parent.


Visitor pattern.
<<IChildrenVisitor>>
+ Object : emptyCase(AChildrenList host, Object inp)
+ Object : nonEmptyCase(AChildrenList host, Object inp)
A Parent has a name, a birth year and a list of children.
calls on
operates on
Parent
− String : _name
− int : _year
− AChildrenList : _children
+ String : getName()
+ int : getYear()
+ AChildrenList : getChildren()
1
children
1
_rest
AChildrenList
+ Parent : getChild()
+ AChildrenList : getRest()
+ Object : call(IChildrenVisitor algo, Object inp)
1
_child
standard accessor ("getter") methods.
NEChildrenList
− Parent : _child
− AChildrenList : _rest
+ NEChildrenList(Parent child, AChildrenList rest)
+ Parent : getChild()
+ AChildrenList : getRest()
+ Object : call(IChildrenVisitor algo, Object inp)
MTChildrenList
+ MTChildrenList : mTChildrenList
− MTChildrenList()
+ Parent : getChild()
+ AChildrenList : getRest()
+ Object : call(IChildrenVisitor algo, Object inp)
Singleton
1
A non-empty children list has a child, which is of type Parent, and a rest, which is an AChildrenList.
Empty children list. Throws exceptions when asked for child or rest.

Write a concrete ***IChildrenVisitor*** to compute and return an ***AChildrenList*** containing all descendants of a Parent born after a given year N. 15 pts