

Name: Solution

Midterm

CMSC 433

Programming Language Technologies and Paradigms
Spring 2004

March 18, 2004

Instructions

This exam contains 11 pages, including this one. Make sure you have all the pages. Write your name on the top of this page before starting the exam. The last page contains a cheat sheet of methods for `LinkedLists` and `Iterators`.

Write your answers on the exam sheets. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

You may avail yourself of the *punt* rule. If you write down *punt* for any numbered or lettered part of a question, you will earn 1/5 of the points for that question (rounded down).

Question	Score	Max
1		25
2		15
3		30
4		30
Total		100

Question 1. Short Answer (25 points).

a. (5 points) What is the difference between white box and black box testing? Explain.

Answer: White box testing takes the source code into account, using metrics such as statement, branch, and condition coverage to judge the level of completeness. Black box testing does not use the source code, but instead checks the behavior of the code compared to its specification.

b. (5 points) What is an abstract class? Give one example usage of abstract classes (possibly from the class projects or from Java's libraries) and explain the benefits of using an abstract class in that case rather than using an interface. If you can't remember a concrete example, make up a hypothetical example and describe it.

Answer: An abstract class is one in which some of the methods are declared but do not contain code. An example is the `ServletFilter` class from project 1. The benefit compared to interfaces is that subclasses get to reuse some code. If we had made `ServletFilter` an interface, it could not contain code, hence we couldn't get reuse.

c. (5 points) In project 3, our abstract syntax tree accepted a visitor that implemented `NodeVisitor`. We wrote visitors that were subclasses of `NodePrePostVisitor`, and then our `makeVisitor` methods wrapped those to turn them into `NodeVisitors`:

```
class SomeRefactoring extends NodePrePostVisitor {
    public static NodeVisitor makeVisitor(...) {
        return new PrePostOrderVisitor(new SomeRefactoring(...));
    }
    ... }
```

What design pattern (aside from Visitor and Factory) describes this idea—this wrapping a `NodePrePostVisitor` in `PrePostOrderVisitor` to be compatible with `NodeVisitor`? Explain.

Answer: This is an example of the Adapter pattern. We adapt the `NodePrePostVisitor` subclasses to meet the AST's visitor interface. This is *not* an example of the decorator or proxy patterns, because `NodePrePostVisitor` does not have the same interface as `NodeVisitor`. It is also not an example of the template pattern, because the invariant part of the algorithm is in `PrePostOrderVisitor`, not in `NodePrePostVisitor`.

d. (5 points) In project 3, each of the abstract syntax tree classes had an `accept()` method, declared in the `Node` interface, that looked like the following:

```
class SomeASTNode implements Node {
    ...
    public void accept(NodeVisitor v) {
        v.visit(this);
    }
}
```

It seems like this is a lot of trouble—we have to copy the definition of `accept()` to each of the AST node classes, even though the code is always the same. Why couldn't we just make `Node` a regular class and include the code for `accept()` there, while adjusting the definitions of the other AST nodes appropriately? In other words, why didn't we write

```
class Node {
    public void accept(NodeVisitor v) {
        v.visit(this);
    }
}
class SomeASTNode extends Node {
    ...
    // inherit accept() from Node
}
```

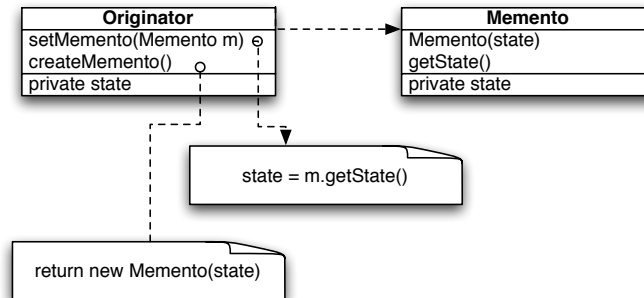
One reason might be that `Node` must be an interface, but that's not a good reason; it's easy enough to change the other classes so that `Node` can be a concrete class. Explain the other reason we can't make this transformation.

Answer: Because `accept()` uses overloading to resolve the particular `visit()` method that is invoked. If `accept()` is defined in `Node`, then only the `visit(Node)` methods will be invoked.

e. (5 points) We've talked about two kinds of *polymorphism* during the semester. Explain what we mean by subtype polymorphism and what we mean by parameteric polymorphism.

Answer: Subtype polymorphism allows a subclass to be used whenever a superclass is expected. Parametric polymorphism is when a class is parametrized by a type. For example, the `List` class might be parameterized by the type of elements stored in the list.

Question 2. UML Diagrams (15 points). This question is about understanding UML diagrams. The *memento* design pattern is used to allow a third party to checkpoint internal state without having direct access to it. Below is the structure of this pattern (copied from Gamma et al). Here the Originator has some internal state that is private. A third party can save or restore that state with `createMemento` and `setMemento`.



Write Java code equivalent to the above structure diagram. That is, write Java code for the classes `Originator` and `Memento`. Your code must compile. The state field of the `Originator` should be a `String`. (*Hint*: If you understand the above diagram, this is an easy problem...don't think it's complicated.)

```

public class Originator {
    private String state;

    public void setMemento(Memento m) {
        state = m.getState();
    }

    public Memento createMemento() {
        return new Memento(state);
    }
}

public class Memento {

    private String state;

    public Memento(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }
}
  
```

Grading note: I was very picky in grading this question, since the code was so simple.

Question 3. Visitor Pattern (30 points). Write a visitor class that traverses a `ClassDef` and verifies that

- All `IdentifierExpressions` refer to variables or fields that are in-scope (according to the usual Java rules).
- At the declaration of a local variable, no other local variable of the same name is currently in scope. But a local variable *may* have the same name as a field.
- No field has the same name as another field.

Your visitor should throw an `IllegalInputProgram(String)` exception if it detects an error; otherwise it should do nothing. Make sure you include support for nested compound statements.

For this question, we will use a simplified version of the AST from project 3, described below. The simplifications are (a) the root node is `ClassDef` (not `SourceFile`); (b) it doesn't have as many kinds of nodes; (c) there is no type information; (c) methods have no parameters; and (d) instead of separate classes `ArithmeticExpression`, `AssignExpression`, and `BinaryOpExpression`, we just use the single class `BinaryExpression`. You can assume there is a `NodeVisitor` interface that has one (overloaded) `visit` method for each of the classes (but not for the `Statement` interface).

Hint: You will probably want to use a `LinkedList` to maintain the sets of fields and variables currently in-scope.

(This table is repeated on the cheat sheet)

Class	Fields	Description
<code>ClassDef</code>	<code>String name</code> <code>List fields</code> <code>List methods</code>	The name of the class A <code>FieldDef</code> list A <code>MethodDef</code> list
<code>FieldDef</code>	<code>String name</code>	The name of the field
<code>MethodDef</code>	<code>String name</code> <code>CompoundStatement body</code>	The name of the method The body of the method
<code>CompoundStatement</code>	<code>List statements</code>	A <code>Statement</code> list
<code>Statement</code>	(interface)	An <code>VariableDef</code> , <code>IdentifierExpression</code> , or <code>BinaryExpression</code>
<code>VariableDef</code>	<code>String name</code> <code>Expression init</code>	The name of the local variable The initializer of the variable (may be null)
<code>IdentifierExpression</code>	<code>String id</code>	The name of the identifier
<code>BinaryExpression</code>	<code>Expression lhs</code> <code>String op</code> <code>Expression rhs</code>	The left-hand side The operation (+, -, &&, =, +=, or -=) The right-hand side

```
/* Fill this class in. NodeVisitor is just like in project 3, and you
can assume each accept(NodeVisitor) method of the AST node classes
contain the single statement v.visit(this). */
public class IdentifierCheckVisitor implements NodeVisitor {
```

```
    private LinkedList vars = new LinkedList();
    private LinkedList fields = new LinkedList();
```

```
    public void visit(ClassDef n) {
        for (Iterator i = n.fields; i.hasNext(); )
            ((FieldDef) i.next()).accept(this);
        for (Iterator i = n.methods; i.hasNext(); )
            ((MethodDef) i.next()).accept(this);
```

```
    }
```

```
    public void visit(FieldDef n) {
        if (fields.contains(n.name))
            throw new IllegalInputProgram("duplicate field " + n.name);
        fields.addLast(n.name);
```

```
    }
```

```
    public void visit(MethodDef n) {
        n.body.accept(this);
```

```
    }
```

```

/* Continue IdentifierCheckVisitor */
    public void visit(VariableDef n) {
        if (vars.contains(n.name))
            throw new IllegalInputProgram(n.name + " shadows");
        n.init.accept(this);
        vars.addLast(n.name);

    }

    public void visit(CompoundStatement n) {
        int mark = vars.size();
        for (Iterator i = n.statements; i.hasNext(); )
            ((Statement) i.next()).accept(this);
        while (vars.size() != mark) vars.removeLast();

    }

    public void visit(IdentifierExpression n) {
        if (!vars.contains(n.id) && !fields.contains(n.id))
            throw new IllegalInputProgram(n.id + " not in scope");

    }

    public void visit(BinaryExpression n) {
        lhs.accept(this);
        rhs.accept(this);

    }
}

```

Question 4. Testing (30 points). Below is part of a version of the `LinkedList` class from Java 1.4.2. (We've simplified the code a bit.) As you can see, our version of `LinkedList` implements singly-linked lists with a dummy head element. In other words, the empty list contains one `Entry` node, and a list with one element contains two `Entry` nodes. We've given you the code for the `indexOf(Object)` method, which either returns the index of the first occurrence of the parameter, or -1 if the element isn't in the list. Notice that we start our search with node `header.next` to skip over the dummy head element. Also notice the support for lists containing null.

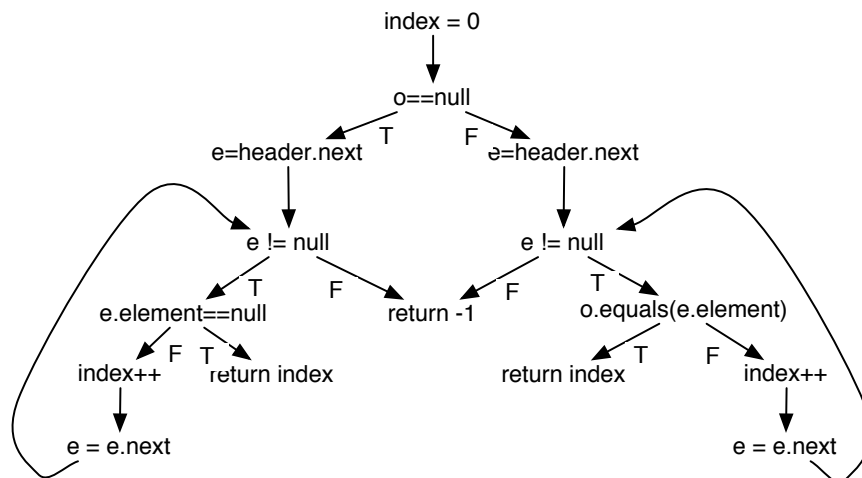
```
public class LinkedList {

    private Entry header = new Entry(null, null);

    private static class Entry { Object element; Entry next; }

    public int indexOf(Object o) {
        int index = 0;
        if (o==null) {
            for (Entry e = header.next; e != null; e = e.next) {
                if (e.element==null)
                    return index;
                index++;
            }
        } else {
            for (Entry e = header.next; e != null; e = e.next) {
                if (o.equals(e.element))
                    return index;
                index++;
            }
        }
        return -1;
    }
}
```

Here is the control-flow graph for the `indexOf(Object)` method.



a. (15 points) Write a series of test cases that will collectively cover all **branches** in the `indexOf(Object)` method. Since the exact kind of object doesn't matter, use lists of Strings. Don't write code—instead, write down math-like representations of lists, e.g., `{"foo", "bar"}`. Write down `null` (no quotes) to denote null. You may write down more than 4 test cases if you need to.

LinkedList list	The parameter param in list.indexOf(param)
<code>{"foo", null}</code>	<code>null</code>
<code>{}</code>	<code>null</code>
<code>{"foo", "bar"}</code>	<code>"bar"</code>
<code>{}</code>	<code>"bar"</code>

b. (10 points) Pick one of your test cases above and write a JUnit test class that implements the test. You should both invoke the `indexOf(Object)` method and check that the result is correct (something we did not bother with in part (a)). Use the regular `LinkedList` methods from the cheat sheet to build the list.

```
public class indexOfTest extends TestCase {

    public void testOne() {
        LinkedList l = new LinkedList();
        l.addLast(new String("foo"));
        l.addLast(new String("bar"));
        assertTrue(l.indexOf(null) == 1);
    }

}
```

c. (5 points) Suppose for part (a) instead of branch coverage we had wanted to cover all statements in the program. Would we need more test cases to do so? How are the test cases for statement and branch coverage related in general? Justify your answer.

Answer: Branch coverage is always a superset of statement coverage, since every statement in the program is reachable on some branch. So we would not need any more test cases for part (a).

This page intentionally left blank.

Cheat sheet: Useful classes and methods. You may use methods from these classes that we haven't listed, if you like.

```
public class LinkedList implements List {

    /* Constructs an empty list. */
    public LinkedList();

    /* Returns the number of elements in this list. */
    int size();

    /* Appends the specified element to the end of this list */
    void addLast(Object o);

    /* Removes and returns the last element from this list; throws
       NoSuchElementException if this list is empty. */
    Object removeLast();

    /* Returns true if this list contains the specified element at least once */
    boolean contains(Object o);

    /* Returns an iterator over the elements in this list in proper sequence. */
    Iterator iterator();
}
```

```
public class Iterator {

    /* Returns true if the iteration has more elements */
    boolean hasNext();

    /* Returns the next element in the iteration; throws
       NoSuchElementException if the iterator has no more elements */
    Object next();
}
```

Repeat of table from question 3

Class	Fields	Description
ClassDef	String name List fields List methods	The name of the class A FieldDef list A MethodDef list
FieldDef	String name	The name of the field
MethodDef	String name CompoundStatement body	The name of the method The body of the method
CompoundStatement	List statements	A Statement list
Statement	(interface)	An VariableDef, IdentifierExpression, or BinaryExpression
VariableDef	String name Expression init	The name of the local variable The initializer of the variable (may be null)
IdentifierExpression	String id	The name of the identifier
BinaryExpression	Expression lhs String op Expression rhs	The left-hand side The operation (+, -, &&, =, +=, or -=) The right-hand side