Midterm #1

CMSC 433
Programming Language Technologies and Paradigms
Spring 2006

March 29, 2006

Guidelines

This exam has 10 pages (including this one); make sure you have them all. Put your name on each page before starting the exam. Write your answers directly on the exam sheets, using the back of the page as necessary. Bring your exam to the front when you are finished. 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 part of a question with a specifically-assigned point value, you will earn 1/5 of the points for that question (rounded down).

Use good test-taking strategy: read through the whole exam first, and first answer the questions that are easiest for you and are worth the most points.

Question	Points	Score
1	20	
2	20	
3	20	
4	20	
5	20	
Total	100	

1. (Short answer, 20 points)

   (a) (5 points) Give a reason why JUnit tests could be viewed as black box tests, or alternatively give a reason why they could be viewed as white box tests.

   (b) (5 points) A principle underlying many design patterns is "encapsulate what varies." Give an example of a design pattern that follows this principle and justify your answer.

   (c) (5 points) Synchronization (i.e., locking) is one way to avoid data races in multi-threaded programs. Briefly describe another technique for avoiding races.

   (d) (5 points) Is deadlock a violation of safety or liveness?

2. (Generics, 20 points)

Below is non-generic interface for a stack:

```
public interface Stack {
  public void push(Object x);
  public Object pop() throws EmptyStackException;
  public bool empty();
}
```

The `push` method adds an element to the stack, and `pop` removes one, throwing an exception if the stack is empty. `empty` returns true if the stack is empty, false otherwise. Here is some code that uses the stack:

```
Stack s = ...; // some implementation; we don't care how we got it
s.push("hello");
s.push("there");
String x = (String)s.pop();
String y = (String)s.pop();
```

(a) (10 points) Rewrite the code for the `Stack` interface to use generics instead. Rewrite the example code to use your new generic `Stack` (or clearly modify the example code above).

(b) (10 points) Suppose we have some classes for defining shapes, where all shapes have a `draw` method:

```
interface Shape {       class Square implements Shape { ... }
  void draw();          class Circle implements Shape { ... }
}
```

In Java 1.4, we could write the following method for drawing all shapes stored in a stack:

```
static void popAndDrawAll(Stack shapes) {
  while (!shapes.empty()) {
    Shape s = (Shape)shapes.pop()
    s.draw();
  }
}
```

Rewrite this code to use the generic stack you defined for the first part, implementing either a polymorphic method or a method using wildcards. You should not need any dynamic downcasts. Your new code should type check when used as follows:

```
Stack<Shape> s1 = ...;
Stack<Square> s2 = ...;
popAndDrawAll(s1);
popAndDrawAll(s2);
```

3. (Visitor Pattern, 20 points)

Background (Problem statement begins on the next page.) Below are some classes that implement arithmetic expressions, along with an `accept` method to implement the *visitor* pattern. *These are exactly the same classes as provided in the lecture notes on the visitor pattern.*

```
interface Node { }

class Number implements Node {
  int num;
  Number(int n) { num = n; }
  void accept(Visitor v) {
    v.visit(this);
  }
}
```

```
class Plus implements Node {
  Node lhs;
  Node rhs;
  Plus(Node l, Node r) { lhs = l; rhs = r; }
  void accept(Visitor v) {
    v.visit(this);
  }
}
```

As an example, the code for generating the expression `(1+2)+3` would be

```
Node x = new Plus(new Plus(new Number(1),new Number(2)),new Number(3));
```

The visitor pattern is handy in such class hierarchies since it allows one to define data separately from processors of that data. Here is an implementation of a visitor that calculates the result of an arithmetic expression (notice how intermediate results are stored on the call stack):

```
interface Visitor {
  void visit(Number n);
  void visit(Plus n);
}
```

```
class ComputeVisitor implements Visitor {
  int result = 0;
  void visit(Number n) {
    result = n.num;
  }
  void visit(Plus p) {
    int leftresult;
    p.lhs.accept(this);
    leftresult = result;
    p.rhs.accept(this);
    result = leftresult + result;
  }
}
```

To use this visitor on `x` and print the result, we could do the following

```
ComputeVisitor v = new ComputeVisitor();
x.accept(v);
System.out.println(v.result);
```

(next page)

(a) (5 points) What is the purpose of the `accept` methods in each of the `Node` classes? Put another way, why can we not simply write the `ComputeVisitor` as follows, which ignores the `accept` methods and calls `visit` directly:

```
class ComputeNotAVisitor implements Visitor {
  int result = 0;
  void visit(Number n) {
    result = n.num;
  }
  void visit(Plus n) {
    int leftresult;
    this.visit(n.lhs);
    leftresult = result;
    this.visit(n.rhs);
    result = leftresult + result;
  }
}
```

(b) (5 points) Give a concrete, substantive difference between a visitor and an *iterator* (saying that one uses `accept` methods and the other doesn't, or making similar sorts of surface-level observations, is not sufficient).

(c) (10 points) A "payload" visitor can be used to separate the computation of a visitor from its traversal. For example, here is a traversal visitor for our original set of `Node` classes (not including `Power`):

```
class PostTraverseVisitor implements Visitor {
  Visitor payload;
  PostTraverseVisitor(Visitor v) { payload = v; }
  void visit(Number n) {
    payload.visit(n);
  }
  void visit(Plus n) {
    n.lhs.accept(this);
    n.rhs.accept(this);
    payload.visit(n);
  }
}
```

The calls to `accept` do the traversal, and the calls to `payload.visit()` do the processing. We can use this approach to implement visitors like `ComputeVisitor` a bit more generically. For example, using `PostTraverseVisitor`, we could implement a `ComputePayloadVisitor` that results in a visitor equivalent to `ComputeVisitor` as follows:

```
ComputePayloadVisitor v2 = new ComputePayloadVisitor();
Visitor v3 = new PostTraverseVisitor(v2);
x.accept(v);   // the original ComputeVisitor
x.accept(v3);  // one composing v2 and v3
assertEquals(v.result,v2.getResult()); // they give the same result
```

Provide your definition of a the payload visitor on the next page. The `getResult` method should return the final result (as shown in the code snippet above).

```
class ComputePayloadVisitor implements Visitor {


  ComputePayloadVisitor() { // empty constructor }

  void visit(Number n) {









  }
  void visit(Plus p) {









  }
  void int getResult() {




  }
}
```

4. (Threads, 20 points) Suppose we have the following code:

```
class Foo {
  void go(SomeObject o) {
    String s = o.someMethod(1,2);  // CHANGE: run o.someMethod(1,2) in parallel
    ...                            // s not used here
    System.out.println(s);         // CHANGE: print s when it's ready
  }
}
```

Suppose that within the `...` we never use `s`, so it might make sense execute the call to `o.someMethod(1,2)` in parallel, in a separate thread; when this thread completes, the parent would print out `s` as before. Write code to do this below: show modifications to `go` and any new classes you need.

5. (Design Patterns, 20 points)

Recall from project 1 the `Servlet` interface:

```
public interface Servlet {
  public void doGet(String path, String options, OutputStream out)
     throws ServletException, ShutdownException;
}
```

(a) (10 points) You wrote a `Cache` class that implemented the `Servlet` interface. This class cached the results of queries to avoid redundant work on subsequent queries having the same path and options. This class implements the *decorator* pattern. Draw a UML diagram of the decorator pattern as it applies to the `Cache` class in this situation.

(b) (10 points) *Balking* and *guarding* are two design patterns for implementing multi-threading-compatible *state-dependent actions*, which are actions that can sometimes fail depending on the current state. Indicate which methods in the following class you believe are performing balking (if any), guarding (if any), or both (if any):

```
public class FixedLengthStringQueue {
  public FixedLengthStringQueue(int size);
  public void put(String obj);
  public String get();
  public void offer(String obj) throws FullQueueException;
  public String take() throws EmptyQueueException;
  public String takeMS(int timeoutMS) throws EmptyQueueException;
}
```