Final Exam
CMSC 433
Programming Language Technologies and Paradigms
Fall 2008
December 18, 2008

Guidelines

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. If you finish with more than 15 minutes left in the class, then bring your exam to the front when you are finished and leave the class as quietly as possible. Otherwise, please stay in your seat until the end.

If you have a question, raise your hand and the proctor will come to you. Note, that he will not answer general questions. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen and egregious error, however, you shouldn’t need to do this at all, so be careful when making assumptions.

NOTE: There are quite a few questions to answer. Get started right away. Read each question carefully and answer the question that has been asked. Budget your time wisely. If you can’t answer a question in a few minutes, move on and come back to the difficult question later. Do not get bogged down on a single question.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
1. Design Principles: Short answers (10 points). Give very short (1 to 2 sentences for each issue) answers to the following questions. Longer responses to these questions will not be read.

(a) In the context of software design, what is abstraction? **Answer:**

   decompose system into abstract components. Abstractions emphasize essential characteristics and suppress implementation details

(b) In the context of software design, what is hierarchy? **Answer:**

   restrict module interactions by restricting topology of module relationships

(c) In the context of software design, what is information hiding? **Answer:**

   Defining modules such that each module encapsulates something that is likely to change.
2. The Decorator Pattern (15 points). You are developing a graph plotting application that uses the Decorator pattern. Your application has three decorator classes: BasePoint, SnapToGrid, and PrintPoint. Each of these classes implements the Point interface. The Point interface defines three methods: getX() - which returns the Point’s X coordinate; getY() - which returns the Point’s Y coordinate; and move (int dx, int dy) - which moves the point dx units on the X axis and dy units on the Y axis.

interface Point {
    public void move (int dx, int dy);
    public int getX();
    public int getY();
}

Assume the graphing plane has an origin of (0,0) and that the plane encompasses all non-negative XY coordinates. Also assume only valid move operations which leave the point within the plane. That is, don’t worry about error handling for illegal move() operations.

BasePoint represents a point in a 2D plane. BasePoint cannot decorate other Points. The BasePoint constructor takes an initial XY coordinate for the Point.

SnapToGrid can decorate other Points. The SnapToGrid constructor takes two arguments: the gridSize and a Point to decorate. SnapToGrid restricts Points to lie on XY coordinates that are integer multiples of the gridSize (gridlines). So if you try to create or move a SnapToGrid point with/to coordinate (22,25) and the gridSize is 10, then the Point should be placed at (20,30). This is because the nearest X coordinate to 22 that is on the grid is 20 and the nearest Y coordinate to 25 that is on the grid is 30 (round up for Points that are equidistant to two grid lines).

PrintPoint logs the effect of each operation to standard out.

The PointClient Driver and BasePoint classes are as follows:

```java
public class PointClient {
    public static void main (String [] args) {
        Point a = new BasePoint (33,17); // New Point created at (33,17)
        Point b = new SnapToGrid(10,a); // Point moved to nearest gridline (30,20)
        Point c = new PrintPoint(b); // Adds logging capability
        c.move(-16,5); // Point moves to (10,20). Prints: "Moved to (10,30)"
        c.getX(); // Prints: "getX() returns: 10"
        c.getY(); // Prints: "getY() returns: 30"
    }
}
```

```java
public class BasePoint implements Point {
    int x, y;
    BasePoint (int x, int y) {
        this.x = x; this.y = y;
    }
    public void move (int dx, int dy) {
        this.x += dx; this.y += dy;
    }
    public int getX () {return x;}
    public int getY() {return y;}
}
```

Fill in the code needed to implement the PrintPoint and SnapToGrid classes below:

```java
public class PrintPoint implements Point {
    Point p;

    PrintPoint (Point p) {

    }

    public int getX() {

    }

    public int getY() {

    }

    public void move(int dx, int dy) {

    }
}
```
public class SnapToGrid implements Point {
    int gridSize;
    Point p;

    SnapToGrid(int gridSize, Point p) {
    }

    public void move(int dx, int dy) {
    }

    public int getX() {
    }

    public int getY() {
    }
}
public class PrintPoint implements Point {
    Point p;
    PrintPoint (Point p) {
        this.p = p;
    }
    public int getX() {
        int x = p.getX();
        System.out.println("getX() returns: ' + x);
        return x;
    }
    public int getY() {
        int y = p.getY();
        System.out.println("getY() returns: ' + y);
        return y;
    }
    public void move(int dx, int dy) {
        p.move(dx, dy);
        System.out.println("Moved to (" + p.getX() + "," + p.getY() + ")");
    }
}

public class SnapToGrid implements Point {
    int gridSize;
    Point p;
    SnapToGrid(int gridSize, Point p) {
        this.gridSize = gridSize;
        this.p = p;
        move(0, 0);
    }
    public void move(int dx, int dy) {
        int tmpX = getX() + dx, tmpY = dy + getY();
        int newX = (tmpX % gridSize);
        if (newX >= gridSize / 2) {
            tmpX += gridSize - newX;
        } else {
            tmpX -= newX;
        }
        int newY = (tmpY % gridSize);
        if (newY >= gridSize / 2) {
            tmpY += gridSize - newY;
        } else {
            tmpY -= newY;
        }
        p.move(tmpX - getX(), tmpY - getY());
    }
    public int getX() { return p.getX(); }
    public int getY() { return p.getY(); }
}
3. Test Coverage (15 points). Consider the code below. In class we discussed three different kinds of coverage criteria: statement, branch and condition. Give a set of test cases (each test case is a list of values for a, b and c) that achieves: (a) 100% statement coverage, but less than 100% branch and condition coverage, (b) 100% branch coverage but less than 100% condition coverage, and (c) 100% condition coverage.

class CoverageTest {
    static public void main(String[] args) {

        boolean a = Boolean.parseBoolean(args[0]);
        boolean done = Boolean.parseBoolean(args[1]);
        int c = Integer.parseInt(args[2]);

        if (a && (c < 0 || done)) {
            c = 100;
        } else {
            done = !done;
        }
        while (c <= 0 || !done) {
            c--;
            if (c == 23) {
                done = true;
            }
        }
    }
}

Answer:

a) statement coverage
T F -1  \
F F -1  \\

b) branch coverage
T F -1  \
F F -1  \\

c) condition coverage
T F -1  \
F T 0  \
F T -1  \\
4. State Dependent Actions (10 Points). The following code models a situation in which three actors take turns using a 2-seat loveseat. To use the loveseat you must first wait for a seat to open up. Then you sit down, then rest, then get up. The code should enforce the following policy: If someone is waiting to sit down and a seat becomes free, then they should be the next to sit. The code, however, does not enforce the policy. Give an trace that demonstrates a violation of the policy.

```java
public class Seat {
    private int seats = 2;

    public synchronized void sitDown() {
        while (seats == 0) {
            try {wait();} catch (InterruptedException e) {} 
        }
        seats--;
    }

    public synchronized void getUp() {
        if (seats == 0) notifyAll();
        seats++;
    }
}

public class LoveSeatManager {
    public static void main(String[] args) {
        final long restTime = 1000L;
        final Seat seat = new Seat();
        Runnable sitter = new Runnable() {
            public void run() {
                while (true) {
                    seat.sitDown();
                    try {Thread.sleep (restTime);} catch (InterruptedException e) {} 
                    seat.getUp();
                }
            }
        };

        ExecutorService exec = Executors.newCachedThreadPool();
        for (int i = 0; i < 3; i++) {exec.execute(sitter);}
    }
}
```

Answer:

- T1 - sitting
- T2 - sitting
- T3 - waiting
- T2 - gets up
- T3 - receives notifyAll
- T2 - sits down
5. Multi-threading (10 points). The following program can deadlock. Give an execution trace showing how this can occur.

```java
public class Deadlock {
    static class Friend {

        public synchronized void bow(Friend bower) {
            System.out.println(‘‘I bow to you’’);
            bower.bowBack(this);
        }

        public synchronized void bowBack(Friend bower) {
            System.out.println(‘‘I bow back to you’’);
        }
    }

    public static void main(String[] args) {
        final Friend f1 = new Friend(), f2 = new Friend();
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new Runnable() {public void run() {while (true) {f1.bow(f2);}}});
        exec.execute(new Runnable() {public void run() {while (true) {f2.bow(f1);}}});
    }
}
```

Answer:

- T1 enters bow - holds lock on f1
- T2 enters bow - holds lock on f2
- T1 calls f2.bowback (f1) - waits to acquire lock on f2
- T2 calls f1.bowback (f2) - waits to acquire lock on f1
6. Optimistic Retry (15 points). Suppose you want to build a graphing application similar to the one we saw earlier in the exam. Suppose further that you want to support multiple concurrent operations on your Points. An example driver shown below:

```java
public class OptimisticPointClient {
    public static void main(String[] args) {
        final OptimisticPoint a = new OptimisticPoint(33, 17);
        Runnable mover = new Runnable() {
            public void run() {
                while (true) {
                    // read move parameters dx and dy from somewhere
                    a.move(dx, dy);
                }
            }
        };
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(mover);
        exec.execute(mover);
    }
}
```
For this application you will need to handle overlapping move commands. However, you want the move() operation to block as little as possible. Therefore, you decide implement a new class called OptimisticPoint that wraps a BasePoint. You then implement an optimistic retry strategy for moving OptimisticPoints. Fill in the necessary code below. You can only use the methods defined below and Basepoint’s getX() and getY() methods. Do not add any further synchronization blocks.

```java
public class OptimisticPoint {
    protected BasePoint loc;

    public OptimisticPoint(int x, int y) { loc = new BasePoint(x, y); }

    public synchronized BasePoint location() { return loc; }

    protected synchronized boolean commit(BasePoint assumed, BasePoint next) {
    }

    public void move(int dx, int dy) {
    }
}
```
class OptimisticPoint {
    protected BasePoint loc;

    public OptimisticPoint(int x, int y) {
        loc = new BasePoint(x, y);
    }

    public synchronized BasePoint location() { return loc; }

    protected synchronized boolean commit(BasePoint assumed, BasePoint next) {
        if (loc == assumed) {
            loc = next;
            return true;
        }
        else {
            return false;
        }
    }

    public void move(int dx, int dy) {
        boolean success = false;
        do {
            BasePoint old = location();
            BasePoint next = new BasePoint(old.getX() + dx, old.getY() + dy);
            success = commit(old, next);
        } while (!success);
    }
}
7. Garbage Collection (10 points): In class we discussed several methods for garbage collection. Three of these were Reference Counting, Mark and Sweep, and Copying Garbage Collection. Using no more than 2 paragraphs (6-8 simple sentences total) for each method, 1) briefly explain how each technique works, and give 1 positive and 1 negative consequence of using that method.

(a) Reference Counting: Answer:

Technique: Each object tracks the number of pointers to it from other objects and from the roots. When count reaches 0, object can be deallocated.
Pro: Incremental technique
Con: Data in cycle’s can’t be collected

(b) Mark and Sweep: Answer:

Technique: Every so often, stop the world and do GC: Mark all objects on stack as live Mark object reachable from live object as live. Deallocate any non-reachable objects
Pro: No problem with cycles, Memory writes/dropped aliases have no cost
Con: Fragmentation, Cost proportional to heap size, need to stop the world

(c) Copying Garbage Collection: Answer:

Technique: Divide heap into two equal parts (semispaces), Only one semispace active at a time GC copies live data from one to the other. Afterwards, declare everything in current semispace dead; switch to other semispace
Pro: Only touches live data, No fragmentation; automatically compacts
Con: Requires twice the memory space, need to stop the world
8. RMI. (15 Points). You are asked to write an RMI application that serves weather information to distributed clients. The RMI Server interface for this application is called WeatherService. It appears below:

```java
public interface WeatherService extends Remote {
    public String getWeatherInformation() throws RemoteException;
}
```

(a) Fill in the code needed to implement the RMI Server implementation called WeatherServiceImpl. You should start the registry on the default port from within the Server and do all setup activities needed so that Clients can later access the Server.

```java
public class WeatherServiceImpl implements WeatherService {
    public String weatherInformation = null;

    public WeatherServiceImpl() { updateWeatherInformation(); }

    public String getWeatherInformation() throws RemoteException {
        return weatherInformation;
    }

    public void updateWeatherInformation() {
        weatherInformation = "Snowing";
    }

    public static void main(String[] args) throws Exception {
        java.rmi.registry.LocateRegistry.createRegistry(1099);
        WeatherService weatherService = new WeatherServiceImpl();
        WeatherService stub = (WeatherService)
            UnicastRemoteObject.exportObject(weatherService, 0);
        String objectURL = "WeatherService";
        Naming.rebind(objectURL, stub);
    }
}
```

Answer:

```java
public class WeatherServiceImpl implements WeatherService {
  public String weatherInformation = null;

  public WeatherServiceImpl() { updateWeatherInformation(); }

  public String getWeatherInformation() throws RemoteException {
      return weatherInformation;
  }

  public void updateWeatherInformation() {
      weatherInformation = "Snowing";
  }

  public static void main(String[] args) throws Exception {
      java.rmi.registry.LocateRegistry.createRegistry(1099);
      WeatherService weatherService = new WeatherServiceImpl();
      WeatherService stub = (WeatherService)
          UnicastRemoteObject.exportObject(weatherService, 0);
      String objectURL = "WeatherService";
      Naming.rebind(objectURL, stub);
  }
}
```
public class WeatherServiceClient {
    private WeatherService weatherService = null;

    // put reference to RMI Server in weatherService
    public WeatherServiceClient(String server) {
        String remoteName = "rmi://" + server + ":1099/WeatherService";
        try {
            weatherService = (WeatherService) Naming.lookup(remoteName);
        } catch (Exception e) {}  
    }

    private String getInfo() {
        try {
            return weatherService.getWeatherInformation();
        } catch (RemoteException e) { e.printStackTrace(); }  
        return null;
    }

    public static void main(String[] args) {
        WeatherServiceClient wsc = new WeatherServiceClient("serverMachine");
        System.out.println("Received Weather Information: " + wsc.getInfo());
    }
}