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 I will come to you. Note, that I am unlikely to answer general questions however. 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.

NOTE: There are quite a few questions to answer. Get started right away and budget your time wisely.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
1. Short answers (15 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) One key principle of design patterns is to "program to an interface, not to an implementation." What does that mean? Give two reasons why this is beneficial.

   **Answer:**
   
   Write code that relies on supertype APIs. Possible benefits: (1) Can change implementation, (2) Enables reuse / creation of reusable components. Other answers accepted.

   (b) Synchronization in Java serves three functions: visibility, ordering, atomicity. Briefly explain each function.

   **Answer:**

   i. **Atomicity.** Allows operation to be done without interference from other threads.
   
   ii. **Visibility.** Signals when values will be updated across threads.
   
   iii. **Ordering.** Allows programmers to ensure that an operation A occurs before operation B.

   (c) Information hiding refers to a criteria for modularizing software systems. What is this criteria?

   **Answer:**

   Modules should encapsulate things that are likely to change.
2. The Bridge Pattern (10 points). You are designing software for devices in a home. Each of these devices has a switch. Actual Switches come in several varieties, such as a pull chain, a two-position switch, or a dimmer switch. Explain how the Bridge pattern could be used in this context. Keep your answer under 2–3 paragraphs.

Answer:

The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. So the operations interface would support at least two methods on() and off(). Subclasses could add dimmer capabilities. The actual switches can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches. So the SwitchImpl interface would implement on() and off() in terms of the methods provided by the actual devices.
3. Visitor Pattern (30 points). Implement 2 visitor classes for the following scenario. The class VisitorDemo simulates an airline flight attendant taking care of various passengers. The flight attendant can help seat a passenger and can feed a passenger. These two tasks are implemented in the HelpSeat and Feed classes. These 2 tasks are applied to a cabin of passengers. Passengers come in 3 types: First, Business and Economy. Different types of passengers get different treatment. Consider the driver program below.

```java
public class VisitorDemo {
    public static Passenger[] list = { new First(), new Business(), new Economy() };

    public static void main(String[] args) {
        Visitor sit = new HelpSeat(), eat = new Feed();
        for (int i = 0; i < list.length; i++)
            list[i].accept(sit);
        for (int i = 0; i < list.length; i++)
            list[i].accept(eat);
    }
}
```

It should print out the text below:

This way your grand exalted Highness to First Class
Follow me Boss to Business
Economy is that way, pal
Another glass of Champagne?
May I pour you an after dinner cordial?
You want more water? That will be $5

Using the following interfaces, implement the First, Business, Economy, HelpSeat and Feed classes needed by the code above.

```java
public interface Visitor {
    public void visit(First e);
    public void visit(Business e);
    public void visit(Economy e);
}
```

```java
public interface Passenger {
    public void accept(Visitor v);
}
```
Answer:

class First implements Passenger {
    public void accept(Visitor v) { v.visit(this); }
}
class Business implements Passenger {
    public void accept(Visitor v) {
        v.visit(this);
    }
}
class Economy implements Passenger {
    public void accept(Visitor v) {
        v.visit(this);
    }
}

public class HelpSeat implements Visitor {
    public void visit(First e) {
        System.out.println("This way your grand exalted Highness to First Class");
    }
    public void visit(Business e) {
        System.out.println("Follow me Boss to Business");
    }
    public void visit(Economy e) {
        System.out.println("Economy is that way, pal");
    }
}

class Feed implements Visitor {
    public void visit(First e) {
        System.out.println("Another glass of Champagne");
    }
    public void visit(Business e) {
        System.out.println("May I pour you an after dinner cordial");
    }
    public void visit(Economy e) {
        System.out.println("You want more water? That will be $5");
    }
}
4. Java Concurrency (25 points). Consider the following program.

```java
public class ProducerConsumer {
    private boolean valueReady = false;
    private int bufferValue;

    void produce(int i) {
        while (valueReady) {} // busy loop
        synchronized (this) {
            bufferValue = i;
            valueReady = true;
        }
    }

    int consume() {
        while (!valueReady) {} // busy loop
        synchronized (this) {
            valueReady = false;
            return bufferValue;
        }
    }

    public static void main(String[] args) {
        final ProducerConsumer p = new ProducerConsumer();
        // 3 THREADS CALL PRODUCE AND CONSUME ON P (see below)

    }
}
```

(1) Although this program is intended to follow normal buffer semantics, under some circumstances the same data can be read multiple times from the buffer.

Assuming 3 threads that loop forever, each one calling only p.produce() or only p.consume(). Provide a smallest program trace you can that shows how the above program can fail.

In describing your program trace, first briefly describe your 3 threads (i.e., are they producers or consumers). Then create a table like the one below, showing the steps executed by each thread. A step will be defined as an entry to p.produce’s busy loop (written EPLP), an exit from p.produce’s busy loop (written XPLP), an entry to p.consume’s busy loop (written ECLP), an exit from p.consume’s busy loop (written XCLP), acquiring a lock on p (written AL), and releasing a lock on p (written RL). (The steps should be interleaved so that only one thread takes a step on each line in the table.)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
<th>bufferValue</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Put your answer on the following sheet if necessary.

(2) This program might also deadlock. Give a short description how this might happen.
Answer:

Part 1. There are 2 consumer threads and 1 producer thread.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>C2</td>
<td>P1</td>
</tr>
<tr>
<td>ECLP</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>ECLP</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>EPLP</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>XPLP</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>AL</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>RL</td>
</tr>
<tr>
<td>-</td>
<td>XPLP</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>AL</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>RL</td>
<td>-</td>
</tr>
<tr>
<td>XPLP</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AL</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RL</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Part 2. There is a race condition on the value of valueReady. It’s possible that valueReady could be seen as false by the consumer threads and as true by the producer thread. All three threads could then get stuck in the busy loop.
5. Java Concurrency (20 points) Some executions of the following program result in deadlock. Read the code carefully to determine how deadlock could occur. Then draw a wait graph that illustrates the deadlock situation resulting from your execution. Remember that wait graphs show Threads as circles, objects as rectangles, lines from Thread t to Object o when t holds a lock on o, and lines from Object o to Thread t when t is blocked while trying to acquire the lock on o.

```java
public class Worker extends Thread{
    protected static class Resource {
        protected boolean inUse;
        public synchronized boolean get() {
            return inUse ? false : (inUse = true);
        }
        public synchronized void put() {
            inUse = false; notify();
        }
        public synchronized void waitFor() {
            while (!get())
                try {wait();} catch (InterruptedException e) {}
        }
    }

    protected Resource one, two;
    Worker(Resource one, Resource two) {
        this.one = one; this.two = two;
    }

    public void run() {
        for (;;) {
            one.waitFor();
            two.waitFor();
            // do work
            one.put();
            two.put();
        }
    }

    public static void main(String args[]) {
        Resource f0 = new Resource(), f1 = new Resource(), f2 = new Resource();
        (new Worker(f2, f1)).start();
        (new Worker(f0, f2)).start();
        (new Worker(f1, f0)).start();
    }
}
```
Figure 1: Wait graph showing deadlock. Dotted arrows represent “logically” holding a lock.