Name:	**ID:**

50.003: Elements of Software Construction

Date: April 29, 2015
Time: 9:00 a.m.
Duration: 2 hours

Instructions to candidates:

1. Write your name and ID at the top of this page.
2. This paper consists of 5 questions and 17 printed pages.
3. You are required to write your answers in this exam booklet using a blue or black pen. All answers will be manually graded.
4. You may refer to the course slides and your personal notes.
5. You are **not** allowed to use your laptop, personal computer or any Internet accessing or communicating device during the exam.
6. You may **not** communicate via any means with anyone (aside from the instructors).

For staff's use:

Qs 1	/14
Qs 2	/26
Qs 3	/20
Qs 4	/20
Qs 5	/20
Total	**/100**

Question 1: UML Diagrams *(14 points)*

Draw a UML Class Diagram representing the following elements from the problem domain for a hockey league. A hockey league is made up of at least four hockey teams. Each hockey team is composed of six to twelve players, and one player captains the team. A team has a name and a record. Players have a number and a position. Hockey teams play games (against each other). Each game has a score and a location. Teams are sometimes lead by a coach. A coach has a level of accreditation and a number of years of experience, and can coach multiple teams. Coaches and players are people, and people have names and addresses. Draw a class diagram for this information, and be sure to label all associations with appropriate multiplicities.

Question 2: Thread Safety *(26 points)*

Java offers multiple Queue implementation. Some are array-based (i.e., the underlying implementation uses an array). Some are based on linked list. Linked queues typically have higher throughput than array-based queues. For instance, the following shows an implementation of a bounded linked queue.

```
public class LinkedListQueue {
    private Node head = null;
    private Node tail = null;
    private int bound, size;

    public LinkedListQueue (int bound) {
        this.bound = bound;
        size = 0;
    }

    public void enqueue (String value) {
        if (size < bound) {
            Node temp = new Node();
            temp.value = value;
            temp.next = null;

            if (head == null) {
                head = temp;
                tail = temp;
            }
            else {
                tail.next = temp;
                tail = temp;
            }
            size++;
        }
    }

    public String dequeue () {
        String value = null;

        if (head != null) {
            value = head.value;
            head = head.next;
            if (head == null) {
                tail = null;
            }
            size--;
        }
        return value;
    }
}
```

```
class Node {
    public String value;
    public Node next;
}
```

(a) Implement a thread-safe bounded linked queue by modifying the above sequential program. Note that your implementation should be blocking, i.e., if *enqueue* or *dequeue* is not immediately possible, it should be delayed until it becomes possible. *(16 points)*

```
    public void enqueue (String value) {

    }
```

```
public String dequeue () {

}
```

(b) Java offers a class `ConcurrentLinkedQueue<E>` with only the following documentation.

> *An unbounded thread-safe queue based on linked nodes. This queue orders elements FIFO (first-in-first-out). The head of the queue is that element that has been on the queue the longest time. The tail of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue. A ConcurrentLinkedQueue is an appropriate choice when many threads will share access to a common collection. Like most other concurrent collection implementations, this class does not permit the use of null elements.*

The following methods are offered in the class.

- `ConcurrentLinkedQueue()`: Creates a ConcurrentLinkedQueue object which is initially empty.
- `poll()`: Retrieves and removes the head of this queue, or returns null if this queue is empty.
- `offer(E e)`: Inserts the specified element at the tail of this queue.

Implement a thread-safe blocking queue of strings, with a bound given as a parameter to the constructor, based on `ConcurrentLinkedQueue<E>`. Support `enqueue` and `dequeue` (as above) using the above methods. *(10 points)*

Question 3: Building Blocks *(20 points)*

Phaser (*java.util.concurrent.Phaser*) are similar in functionality to CyclicBarrier and CountDownLatch but supporting more flexible usage. The following are some methods defined in the class.

- `arrive()`: Arrives at this phaser, without waiting for others to arrive.
- `arriveAndDeregister()`: Arrives at this phaser and deregisters from it without waiting for others to arrive.
- `register()`: Adds a new unarrived party to this phaser.

(a) Draw a UML State Machine Diagram describing the behavior of Phaser objects. For simplicity, we assume that there are only these three methods; there is no registration initially; and there are at most 2 registrations. Further assume that there are only 2 phases (i.e., phase 0 and phase 1) and a special `terminated state` is reached after phase 1 has finished. Recall that the a phaser moves to the next phase as soon as the number of registration equals the number of arrivals. Furthermore, the number of registrations minus the number of arrivals is never negative. *(10 points)*

(b) Given the following code, answer the questions below.

```
import java.util.concurrent.Phaser;

public class Question {
    public static void main(String[] args) throws Exception {
        Phaser phaser = new Phaser();
        new MyExaminer(phaser).start();
        new MyStudent(phaser).start();
    }
}

class MyExaminer extends Thread {
    private Phaser phaser;
    public MyExaminer (Phaser phaser) {
        this.phaser = phaser;
        this.phaser.register();
    }

    public void run() {
        System.out.println("examiner waiting for students to get ready;");
        phaser.arriveAndAwaitAdvance();
        phaser.arriveAndAwaitAdvance();
        System.out.println("exam has ended");
    }
}

class MyStudent extends Thread {
    private Phaser phaser;
    public MyStudent (Phaser phaser) {
        this.phaser = phaser;
        this.phaser.register();
    }

    public void run() {
        System.out.println("student ready;");
        phaser.arriveAndAwaitAdvance();
        System.out.println("student handing in exam;");
        phaser.arrive();
        System.out.println("student leaves");
    }
}
```

Is it possible that "exam has ended" would be printed before "student ready"? Explain your answer. In addition, if your answer is yes, suggest a way of fixing the program so that NO 'student handing in exam' or 'student leaves' is printed out after "exam has ended". *(10 points)*

Question 4: Concurrency Pitfalls and Testing *(20 points)*

```
class Taxi {
    private Point location, destination;
    private final Dispatcher dispatcher;

    public Taxi(Dispatcher dispatcher) {
        this.dispatcher = dispatcher;
    }

    public synchronized Point getLocation() { return location; }

    public synchronized void setLocation(Point location) {
        this.location = location;
        if (location.equals(destination))
            dispatcher.notifyAvailable(this);
    }

    public synchronized Point getDestination() { return destination; }
}

class Dispatcher {
    private final Set<Taxi> taxis;
    private final Set<Taxi> availableTaxis;

    public Dispatcher() {
        taxis = new HashSet<Taxi>();
        availableTaxis = new HashSet<Taxi>();
    }

    public synchronized void notifyAvailable(Taxi taxi) {
        availableTaxis.add(taxi);
    }

    public synchronized Image getImage() {
        Image image = new Image();
        for (Taxi t : taxis)
            image.drawMarker(t.getLocation());
        return image;
    }
}

class Image {
    public void drawMarker(Point p) {}
}

class Point {}
```

(a) In class, we have already revealed that there is a potential deadlock in the above code. Write a program (and modify the above code if necessary) to demonstrate the deadlock. *(10 points)*

(b) In class, we have already shown a way of fixing the potential deadlock by using open calls. Fix the problem in an alternative way and write the relevant code below. *(10 points)*

Question 5: Performance *(20 points)*

Lock splitting/stripping is the idea of using a set of locks on a set of independent objects to improve performance. One example is the class below.

```
public class StripedMap {
    // Synchronization policy: buckets[n] guarded by locks[n%N_LOCKS]
    private static final int N_LOCKS = 16;
    private final Node[] buckets;
    private final Object[] locks;

    class Node {
        Node next;
        Object key;
        Object value;
        Node(Object key, Object value, Node next) {
            this.next = next;
            this.key = key;
            this.value = value;
        }
    }

    public StripedMap (int numBuckets) {
        buckets = new Node[numBuckets];
        locks = new Object[N_LOCKS];
        for (int i = 0; i < N_LOCKS; i++) {
            locks[i] = new Object();
        }
    }

    private final int hash(Object key) {
        return Math.abs(key.hashCode() % buckets.length);
    }

1.  public int size() {
2.      int num = 0;
3.      for (int i = 0; i < buckets.length; i++) {
4.          synchronized (locks[i % N_LOCKS]) {
5.              for (Node m = buckets[i]; m != null; m = m.next)
6.                  num++;
7.          }
8.      }
9.      return num;
10. }

    public Object get(Object key) {
        int hash = hash(key);
        synchronized (locks[hash % N_LOCKS]) {
            for (Node m = buckets[hash]; m != null; m = m.next)
```

```
                if (m.key.equals(key))
                    return m.value;
        }
        return null;
    }

    public Object remove(Object key) {
        int hash = hash(key);
        synchronized (locks[hash % N_LOCKS]) {
            //code for removing the object is skipped here.
        }
        return null;
    }

    //...
}
```

Method `size` returns the number of elements in the map. One of the potential problems with its implementation is that if there are a large number of buckets, line 4 will be executed many times and the same object in `locks` will be locked many times.

(a) Modify `size()` such that each object in `locks` is locked exactly once. *(12 points)*

```
    public int size() {









    }
```

(b) Describe as precise as you could what is the precondition and postcondition of your implementation. *(8 points)*

END OF PAPER