Final Exam

CMSC 433, Section 0201
Programming Language Technologies and Paradigms
Spring 2003

May 22, 2003

Instructions

This exam contains 14 pages, including this one. Make sure you have all the pages. Put your name and class account number (last 3 digits only) on each page before starting the exam.

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 a question, your will earn 1/5 of the points for that question (rounded down). You may punt on any number of lettered question, but not on any smaller piece of a question. Punting on a question with multiple parts is scored the same as punting on each individual part. Thus, for example, punting on question 1 will earn you no points.

<table>
<thead>
<tr>
<th>Question</th>
<th>Score</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>
Question 1. Short Answer on old-ish stuff (20 points).

a. (4 points) The use of stubs in RMI is an example of what design pattern we talked about in class? Explain.

b. (4 points) Does a method need to declare all exceptions it may throw? Explain.

c. (4 points) What is the difference between black box and white/glass box testing?
d. (4 points) What is the advantage of using parameterized types in GJ such as List<A> versus regular types such as List?

Answer:

With regular types such as Lists, the programmer constantly has to cast back and forth between Object and the class C that the List actually contains. These casts may fail at run-time. Using GJ programmers can avoid most of the casts on parameterized types. Most importantly, GJ is able to check that parameterized containers are used correctly at compile-time.

e. (4 points) Suppose that class B extends class A, and that in B we declare a method void foo() that overrides method void foo() of class A. Can B’s foo() method be declared to throw a subset, equal set, or superset of the exceptions A’s foo() method is declared to throw?

Answer:

B’s foo() method can be declared to throw a subset of the exceptions of A’s foo() method. (Because of the way the question was worded, “equal” was accepted for partial credit.)
Question 2. Futures (20 points). In this question, you will implement the future design pattern for using separate threads to run tasks that return a result. The tasks to be carried out are objects that implement the Callable interface:

```java
interface Callable {
    Object call(Object arg) throws Exception;
}
```

You must fill in the code for the class Future, shown on the next page. The constructor takes a Callable c and its argument arg. The constructor starts a new thread T, and thread T invokes c.call(arg)—thus the computation of c.call(arg), which may take a while, is running in its own thread. When the user is ready for the result of c.call(arg), the user invokes the get() method of the Future. The get() method waits for thread T to finish computing c.call(arg) and then returns the result. If c.call(arg) raised an exception, then get() also raises the same exception.

*Hint:* If t is a thread, you can invoke t.join() to wait for t to finish.
class Future {

    /** Starts a new thread to compute c.call(arg) */
    Future(Callable c, Object arg) {

    }

    /** Returns result from thread that computed c.call(arg),
     or throws an exception if c.call(arg) did. This method blocks
     until the result is ready (or until an exception is thrown) */
    public Object get() throws Exception {

    }
}
Question 3. Synchronization (10 points). For each of these examples, determine whether it has any potential deadlocks, race conditions, or other synchronization-related problems, and justify your answer. (There may be more than one kind of error in each example.)

a. Silly sum (5 points). Calling doCount() in the following class attempts to add the numbers 0 through 99.

```java
class Parent {
    int n = 0, sum = 0;

    class Child extends Thread {
        synchronized public void run() {
            while (true) {
                while (n == 0)
                    try { wait(); } catch (InterruptedException e) {}
                sum += n;
                n = 0;
                notifyAll();
            }
        }
    }

    synchronized public void doCount() throws Exception {
        (new Child()).start();
        for (int i=0; i<100; i++) {
            n = i;
            notifyAll();
            while (n != 0)
                try { wait(); } catch (InterruptedException e) {}}
        System.out.println("Sum is " + sum);
    }
}
```

Answer: doCount() synchronizes, waits, and calls notifyAll() on the parent object, but run() synchronizes, waits, and calls notifyAll() on the child object. Thus this code will simply hang when run, since once doCount() and run() enter wait(), they’ll never be notified. There’s also a race condition, since different children don’t synchronize on the same lock when changing sum.
b. Back and Forth (5 points). In this example, assume that produce() and consume() may be run by many different threads.

class ProdCons {
    Object buffer = null;

    void produce(Object o) {
        try {
            synchronized (buffer) {
                while (buffer != null) buffer.wait();
                buffer = o;
                buffer.notifyAll();
            }
        } catch (Exception e) {
        }
    }

    Object consume() {
        try {
            synchronized(buffer) {
                while (buffer == null) buffer.wait();
                Object o = buffer;
                buffer = null;
                buffer.notifyAll();
                return o;
            }
        } catch (Exception e) {
            return null;
        }
    }
}

Answer: This code is seriously broken. The problem is that we’re trying to synchronize on the object stored in buffer—but we keep changing that with our own code. And we keep trying to synchronize on null, which raises an exception. But we’ve wrapped all the code in a big try-catch block, so we won’t even see the NullPointerExceptions, and produce() will just mysteriously die and consume() will always return null. (The try-catch placement is also bad because if produce sets buffer to o and then an exception is raised before notification happens, then buffer will be non-null but notifyAll won’t be called, which means things would just freeze.) Needless to say, this is really bad code.
Question 4. Big Messages (20 points). In projects 5 and 6, you sent subclasses of Message over the peer-to-peer network. But what if want to use the network to distribute something like large files? Then we'd like to be able to split such an object up into many different pieces and distribute the pieces over the network in order to distribute the burden on the network.

In this question you will add a method to Portal and implement two new Message classes: FindMessage, which will be broadcast over the network to find the components of a big object (similar to the outgoing message in project 6), and ComponentMessage, which will use forwardBackToSource to return the pieces of a large object back to the requester.

- You need to add a findMessage(String objectName) method to Portal to start the process of finding the components of a big object.

- You can assume that findMessage(String objectName) will only be invoked if at least one of the components of objectName is not available on the LocalPortal.

- You can assume that whoever wants to distribute a big object will make all of its Components (see below) available on various LocalPortals. So you don’t need to worry about the initial distribution of big objects.

- When all of the components of a big object have reached a LocalPortal, then one of the Components' processAll() method (see below) must be invoked. You may assume that the processAll() methods of all components of the same big object are the same.

- processAll() must be invoked only once per initial FindMessage request for a big object.

- As ComponentMessages forward themselves back to the requesting portal, they store their components on the intermediate LocalPortals (so that future requests for this object may be faster).

- In order to reduce network demands, a ComponentMessage containing Component c must only call forwardBackToSource until it reaches a portal that already had c stored locally. It’s safe to stop in this case because we know that the FindMessage that went out followed this path, and thus the component c must have already gone back to the source.

We will not address the issue of splitting objects into components. We will assume each piece of a big object implements the Component interface:

```java
public interface Component extends Serializable {
    /** Returns the index of the current component */
    public int componentId();

    /** This component is part of a big object with component ids 0..numComponents()-1 */
    public int numComponents();

    /** Invoked when all the components of a big object have arrived at a portal. components contains all the components of the message in componentId() order, starting at 0, i.e., for all i, components[i].componentId() == i. */
    public void processAll(LocalPortal portal, Component[] components);
}
```

You may or may not need to use the componentId() and numComponents() fields.
We’ve extended the LocalPortal interface to allow you to save and retrieve components:

```java
public interface LocalPortal {
    public void forward(Message msg);
    public void forwardBackToSource(Long id, Message msg);
    public boolean initiatePeering(String portalAddress) throws Exception;
    public void shutdown();
    public String getAddress();

    /** Stores a component c of object name on the local portal; c may not be null.
     * Returns previously stored component with the same componentId(), or null if none.
     * Thread safe. */
    public Component setComp(String name, Component c);

    /** Returns an array of all the components available locally for object name.
     * Let x be the return value. Then either x[i] is null if component i is not stored
     * on the LocalPortal, or x[i].componentId() == i.
     * Also, the length of the array is x[i].numComponents() for x[i] != null
     * Returns null if there are no components for name available locally.
     * Thread safe. */
    public Component[] getComps(String name);
}
```

For your reference, here is a skeleton of the Message class.

```java
public abstract class Message implements Serializable {
    public final Long id;
    public String from;
    public Message(String from) { ... }
    public Message(String from, int ttl) { ... }
    public abstract void process(LocalPortal portal);
}
```
class Portal ... { 
    public void findMessage(String objectName) {

    }
}

class FindMessage extends Message {

    /** The name of the big object whose components we're trying to find */
    final String objectName;

    public FindMessage(String from, String objectName) {
        super(from);
        this.objectName = objectName;
    }

    public void process(LocalPortal portal) {

    }
}
class ComponentMessage extends Message {

    Long origId;
    Component c;
    String requester;
    String objectName;

    public ComponentMessage(String from, String requester, String objectName,
                              Long origId, Component c) {
        super(from);
        this.origId = origId;
        this.c = c;
        this.requester = requester;
        this.objectName = objectName;
    }

    public void process(LocalPortal portal) {
    }
}
Question 5. Java RMI (15 points).

a. (5 points) Explain the difference between using Remote and Serializable objects for RMI. In particular, suppose in project 5 we had made the Message class (and all subclasses) Remote instead of Serializable. What would have gone wrong with our network?

b. (5 points) Determine when the following statements are true or false. Circle your answer.

- In a class that implements a Remote interface, all the methods must be declared as throwing RemoteException.

  TRUE  FALSE

- In order for you to be able to download code via RMI, you must set your java.rmi.server.codebase.

  TRUE  FALSE

- In order for you to be able to download code via RMI, you must install a security manager.

  TRUE  FALSE
• A remote method invocation may raise RemoteException to signal that there was some network problem that prevented the call from being completed.

TRUE  FALSE

• In order for an object to be used remotely, it must be registered in an RMI registry.

TRUE  FALSE

c. (5 points) Java RMI is only one way to communicate with remote machines. In project 1, for example, you used the http protocol to communicate with a remote web browser. Describe some of the tradeoffs between using RMI for remote communication and using lower-level sockets directly.
Question 6. Special Topics (15 points).

a. **Reflection** (5 points)  Discuss the disadvantages of calling a method via `Method.invoke(Object, Object[])` instead of in the usual way.

b. **Garbage collection** (5 points)  Is it ever the case that, when a garbage collector runs, it may fail to deallocate an object \( o \) even though \( o \) will no longer be used? Explain.

c. **XML** (5 points)  Explain what Document Type Definitions and XML Schemas are used for.