433 Practice Final Questions - COM, JavaBeans, Design Patterns

The final exam is cumulative, covering topics from the entire semester.
These questions mainly cover topics since the 2nd midterm. Look at the two midterms and their practice exams for questions on topics covered earlier in the course.

1. In C++, COM and Java, compare how it is determined that objects are no longer in use and their resources can be recycled. How can a programmer specify actions to be taken when an object is recycled? Try to find something nice to say about all three.

   - In C++, stack allocated objects are collected when they go out of scope. Heap allocated objects must be explicitly deallocated via a call to delete.

     A destructor can specify arbitrary code to be executed when an object is collected.
   - In COM, reference counts are used to determine when components or interfaces can be collected. In languages such as C++, the programmer is responsible for incrementing and decrementing reference counts (which can be very error prone). In other languages, such as Visual Basic, reference count maintenance is automatic. In any case, circular links can make create big problems.

     A developer of a component can provide code to be executed when the reference count goes to zero.
   - In Java, the JVM performs garbage collection to determine when an object can no longer be accessed, and is therefore garbage. When it is detected that an object is garbage, the finalizer for the object is invoked. However, this may come an arbitrary amount of time after the object becomes garbage.

2. Compare, briefly, the relative advantages of using RMI or sockets for a distributed computation.

   - RMI - Using RMI avoids the need to figure out how to encode and decode a request and response as a bitstream. Many other things are easier using RMI, such as passing a remote reference to another service and serialization. However, these can be done with Sockets.
   - Sockets - The primary advantages of Sockets for communication is that it is not specific to Java, and they allow for very large amounts of data to be streamed efficiently

3. For the function `Ticket register(Person p)`, the following are possible pre and post conditions (t is used to refer to the return value).

   (a) Pre: p is not already registered,
   Post: p is registered and t is a Ticket for p

   (b) Pre: p is not already registered,
   Post: (returns normally, p is registered and t is a Ticket for p)
   or (throws SoldOutException)

   (c) Pre: true,
   Post: (returns normally, p is registered and t is a FirstClassTicket for p)
   or (throws AlreadyRegisteredException) Ticket for p) AlreadyRegisteredException)

   (d) Pre: Event not sold out,
   Post: (returns normally, p is registered and t is a Ticket for p)
   or (throws AlreadyRegisteredException)

   Assume that $\texttt{f}_a$ returns to `f` with pre and post conditions given in a, etc. Note that FirstClassTicket is a subtype of Ticket, and that AlreadyRegisteredException and SoldOutException do not have a subtype relationship.

- Show a table that describes whether or not it is legal to substitute a function with the described pre/post conditions for each of the other functions. In other words, which pairs of pre/post conditions could represent legal relationships between a client function's expectations/requirements and the function's interface contract? (The same pairs should also represent legal relationships between the interface contract and implementation guarantees made by the function provider)

Client Expectations	Contract a	b	c	d
a	OK[1]	No[2]	No[2]	No[2,3]
b	OK	OK[1]	No[2]	No[2,3]
c	No[3,4]	No[2,3,4]	OK[1]	No[3,4]
d	No[3]	No[3]	OK	OK[1]

1 - Same

2 - Client not expecting exceptions declared in contract

3 - Client doesn't guarantee contracts preconditions

4 - Ticket returned isn't guaranteed to be a FirstClassTicket

4. (short answers - no more than 3 sentences)

   (a) What is the QueryInterface function, and how does it typically work?
   It is a function in Com's IUnknown interface. It allows you to find out if a COM object supports a particular interface and if so, return a pointer of that type to the COM object.
   In C++, one typical implementation is to have a series of if-statements, checking for each interface the COM object supports, and doing a dynamic_cast to the appropriate type. It must also increment the reference count on the pointer being returned.

   (b) What are the security implications of declaring that a Java class implements Serializable?
   It allows other code to get at and to modify the bit patterns of the serialization of your object. These may reveal private data, and allow them to forge bogus new instances of your class.

   (c) How is Leasing used in distributed systems, and why is it useful?
   When partial failures occur, distributed clients don't logoff. Leasing fixes that by forcing clients to re-establish contact on a regular basis. If failure does occur, after some time period clients that are no longer reachable can be removed from a service listing.

   (d) Give an example where the Singleton design pattern would be useful.
   In any system where exactly one instance of a service is required, for example a print spooler that controls access to a set of printers. In the examples we've seen, the idea is that having more than one of the service would create a resource contention problem, with multiple services/objects vying for the same physical or virtual (e.g., a window manager) resources.

5. Write a JavaBean that implements a Rectangle interface, with properties of the following types:

   - Color color;
   - Point upperLeft;
   - Point bottomRight;
   - boolean visible;

   where Point has interface:

```
interface Point {
  int getX();
  int getY();
  void setX(int);
  void setY(int);
}
```

Include functions that allow other Beans to register as listeners on bound properties, and make the color and visible properties bound so that they trigger events in listeners.

```
import java.beans.*

class Rect implements Rectangle {
private Color color;
private Point upperLeft;
private Point bottomRight;
private Color fill;
private boolean visible;

/* instantiate convenience class object */
private PropertyChangeSupport changes = new PropertyChangeSupport(this);

/* add a listener */
public void addPropertyChangeListener(PropertyChangeListener l) {
  changes.addPropertyChangeListener(l);
}

public void removePropertyChangeListener(PropertyChangeListener l) {
  changes.removePropertyChangeListener(l);
}

void setColor(Color c) {
  Color oldColor = color;
  color = c;
  /* now fire event */
  changes.firePropertyChange("color", oldColor, color);
}

Color getColor() {
  return color;
}

void setUpperLeft(Point ul) {
  upperLeft = ul;
}

Point getUpperleft() {
  return upperLeft;
}

void setBottomRight(Point br) {
  bottomRight = br;
}

Point getBottomRight() {
  return bottomRight;
}

void setVisible(boolean vis) {
  boolean oldVis = visible;
  visible = vis;
```

```
  changes.firePropertyChange("visible", oldVis, visible);
}

boolean isVisible() {
  return visible;
}
```

6. Fill in code where indicated so that the visitAll method properly applies the visitor v to all nodes in the set s. Note that the additional code should not use instanceof, casts or reflection.

```
interface Visitor {
        visitA(A a);
        visitB(B b);
        }

abstract Class Node {
        static void visitAll(Set s, Visitor v) {
                for(Iterator i = s.iterator(); i.hasNext(); ) {
                        Node n = (Node) i.next();
                        n.accept(v);
                        }
                }
        abstract void accept(Visitor v);
        }

Class A extends Node {
        void accept(Visitor v) {
                v.visitA(this);
                }
        }
Class B extends Node {
        void accept(Visitor v) {
                v.visitB(this);
                }
        }
```