McGill University

Faculty of Science

FINAL EXAMINATION

COMP 303 – Programming Techniques

Fall 2006

Examiner: Martin Robillard **Monday, December 11, 2006**
Associate Examiner: Jörg Kienzle **14:00-17:00**

Name (IN INK): ______________________________

Student Number (IN INK): ______________________________

Signature (IN INK): ______________________________

You have **180 minutes** to write this examination.

You must write your answers directly on the question booklet.

You can write in pencil or in pen, as you wish.

This question booklet has 14 pages.

Your answers must be **concise**, **clear**, and **precise**. Long-winded, vague, and/or unclear answers will not get full marks.

No notes, books, or any type of electronic equipment is allowed.

Good luck and happy holidays!

Question	Mark
1	/16
2	/10
3	/18
4	/15
5	/15
6	/16
7	/10
Total (/100)	

Student ID #:____________________________

Question 1: Very short answers [2 marks each, up to a total of 16]

Answer each sub-question using a maximum of **two sentences**. Make sure your answer is **clear** and **precise**.

(a) What is the difference between *subtype polymorphism* and *parametric polymorphism*?.

(b) ~~What is the difference between a *join point* and a *pointcut*~~?

(c) What is the advantage of following the Law of Demeter?

(d) What software engineering problem does the ConcernMapper tool address?

(e) What collection of mechanisms/features does the word "reflection" (as in Java reflection) refer to?

(f) What is the purpose of refactoring, as a software engineering activity?

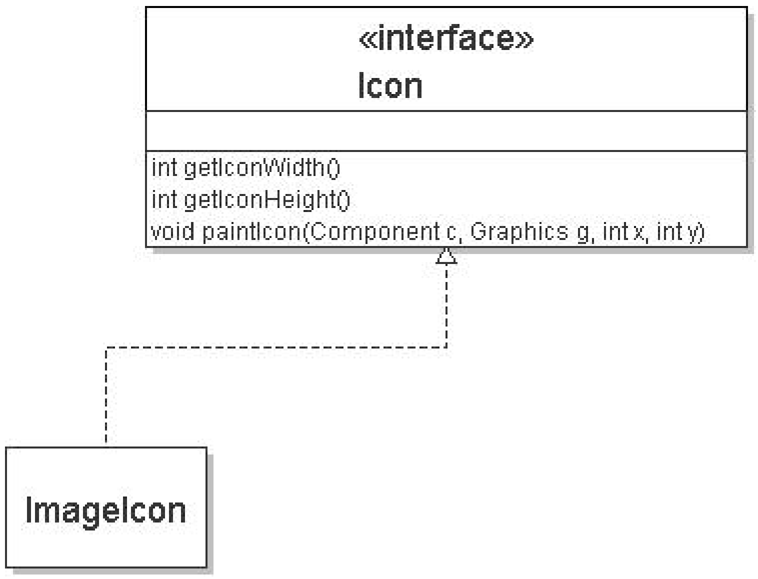
(g) ~~If you need to find out how often a specific method *m()* is called during a test case, will you use hprof or will you write an aspect? Justify your answer~~.

(h) ~~What is the difference between a fault and a failure~~?

Student ID #:__________________________

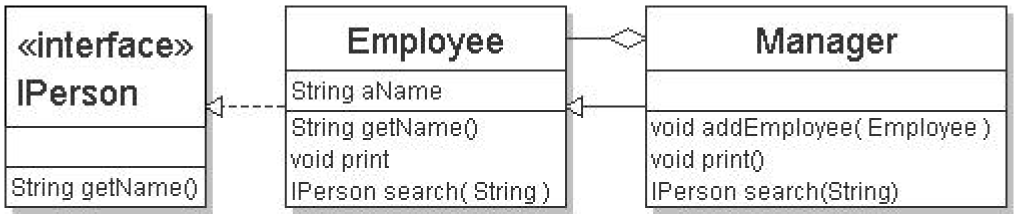
Question 2 [10 marks]

Design a `CompositeIcon` class that can contain multiple icons. Note that a standard application of the COMPOSITE design pattern will result in the composed icons being painted on top of each other. Solve this problem with a `ShiftedIcon` decorator that will support drawing an icon as shifted by (parametric) x and y values. Extend the following diagram to complete the design. You do not need to write the code but make sure you list all the methods (including constructors) that will be necessary to make this work. Add a few lines of explanation to clarify anything not obvious from the diagram.



Question 3 [18 marks]

An employee management system supports tracking employees through a management hierarchy (managers can have multiple employees), as well as recursively printing the name of each employee in the hierarchy, and searching the hierarchy for a person with a specific name. The diagram below shows the class hierarchy, and the code of the `search` method of the `Manager` class.



```
public IPerson search( String pName ) {
      IPerson lReturn = super.search( pName );
      if( lReturn == null ) {
            for( Employee e : aEmployees ) {
                  lReturn = e.search(pName);
                  if( lReturn != null ) return lReturn;
            }
      }
      return lReturn;
}
```

a) Revise the design to add VISITOR support to this class hierarchy to factor out the `print` and `search` operations. Draw the revised class diagram. [8]

b) Write the code of the new `search` method of class `Manager`. If you decided to move the recursion out of the method, include the code implementing the recursion (clearly identify in which method it is located). [4]

c) Assume a variable `bigBoss` of type `Manager` is the root of the hierarchy. Draw a sequence diagram representing the behavior resulting from a call to `bigBoss.search(“Bilbo”)`. Assume that `bigBoss` is not Bilbo himself. [6]

Question 4 [15 marks]

Design a system capable of storing the model for a weather pattern that can be viewed by a variety of clients. Your system must meet the following requirements.

1. Your design must realize the OBSERVER design pattern. However, the logic implementing the mechanism supporting the subject must **not** be in the same class as the model.
2. A basic weather model consists of a single array of 100 `Object`s. It should be possible to update the model by updating individual elements.
3. The client should have control over whether or not a state change triggers a notification of the observers.
4. Use the pull model.

a) Draw a UML class diagram for a design that will respect all the above requirements. Your design should include a `BasicModel` class and two extensions: `Model1` and `Model2`. You should also include two views: `View1` and `View2`. Note that you will need more classes and/or interfaces. Use the `private`, `protected`, `public`, `final`, and `abstract` keywords as appropriate to clarify your design. Only include the information that supports answering the question (for example, you don't need to design what each view actually does, besides the functionality needed to connect it to the model) Briefly explain how your design meets the requirements. [9]

b) Write the code of the class in (a) that encapsulates the implementation of the subject. [4]

c) Complete this code snippet in a client that would connect a view with the model. [2]

```
public class Client
{
   public void connect( BasicModel pModel, View1 pView )
   {
```

--- DO NOT WRITE BELOW THIS LINE ---

Question 5 [15 marks]

Consider the following (partial) implementation of a system allowing farmers to manage their flock of sheep. Note that a `HashSet` is an implementation of the `Set` interface that is backed by a hash table. Appendix 1 provides segments of the Java 1.5.0 API that may be useful for this question.

```
public class Flock
{
      Set<Sheep> aSheep = new HashSet<Sheep>();

      public void addSheep( Sheep pSheep ) { aSheep.add( pSheep ); }
}

class Sheep
{
      private String aName;
      private Date aBirth;    // Date/Time of birth of the sheep
      public Sheep( String pName ) { aName = pName; aBirth = new Date();}
      public void rename( String pName ) { aName = pName; }
      public String getName() { return aName; }
}
```

a) Given the current implementation, if you add the same `Sheep` object multiple times to a `Flock`, it will only be added once. Explain why. [2]

b) At some points scientists discover how to clone sheep. Consider that a cloned sheep is “born” at the time when it is cloned. The system is thus modified by making the `Sheep` class implement the `Cloneable` interface, and by adding the following method to class `Sheep`:

```
public Object clone() {
   try { return super.clone(); }
   catch( CloneNotSupportedException e ) { return null; }
}
```

Briefly explain what happens when this method is called, and why this behavior is probably incorrect with respect to the representation of cloned sheep. [3]

c) Rewrite the `clone` method so that the problem identified in (c) is fixed. [3]

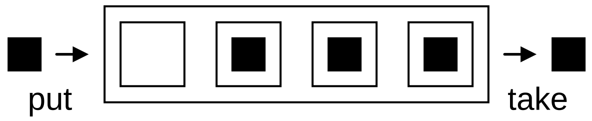
d) As it turns out, farmers freak out at the idea of having multiple copies of a sheep running around in their pasture and require modifications to the system so that it is not possible to have cloned sheep in a flock. How would you solve that problem? Describe (in English) the changes you would make to the system to implement this requirement. [4]

e) In a different scenario (assume the system does not implement the solution for (d), the users of the system find it too inconvenient to allow multiple sheep with the same name in a flock. The system is thus changed so an `equals` and `hashCode` methods are implemented in `Sheep` so that `equals` simply compares sheep names, and `hashCode` simply returns `aName.hashCode().` Why does this make the `rename` method problematic? [3]

Student ID #:_________________________

Question 6 [16 marks]

This question will focus on the implementation of a bounded buffer. A bounded buffer is a data structure that can store a fixed number of elements, and that supports a first-in, first-out (FIFO) behavior. On a bounded buffer, a `put` operation puts an object at the tail of the buffer; A `take` operation takes (and removes) an object from the head of the buffer. `put` is not allowed if the buffer is full, and `take` is not allowed if the buffer is empty. The figure below shows an example of a bounded buffer of capacity 4 filled with 3 elements.



a) Bounded buffers are typically used in multi-threaded designs, where one thread puts elements in the buffer while another thread takes elements from the buffer. In a multi-threaded environment, why is it important to synchronize the buffer? Describe an incorrect execution scenario involving an unsynchronized bounded buffer. [4]

b) Assume we synchronize our bounded buffer so that only one thread can access it at the same time. Furthermore, we define the `put` and `take` operations to be *blocking*. If a thread calls the `put` method on a full buffer, it blocks by yielding its time slice (by calling `Thread.yield()`) and then tries again, until the element is inserted. Similarly, if a thread calls the `take` method on an empty buffer, it blocks by yielding its time slice (by calling `Thread.yield`) and then tries again, until an element can be taken. See question (c) for the code. What is the potential problem with this design? Describe, and illustrate with a scenario of incorrect execution. [4]

c) The following code represents the (faulty) implementation described in b). Complete and modify the code to solve the problem (cross out unwanted statements, add missing statements). See Appendix 2 for details of the Lock API. [8]

```
public class BoundedBuffer
{
   final Lock lock = new ReentrantLock();




   final Object[] aItems = new Object[100];
   int aPutIndex, aTakeIndex, aCount;

   public void put(Object x) throws InterruptedException
   {
      lock.lock();


      try
      {

         while( aCount == aItems.length ) Thread.yield();


         aItems[ aPutIndex ] = x;
         if( ++aPutIndex == aItems.length ) aPutIndex = 0;
         ++aCount;



       }
      finally { lock.unlock(); }
   }

   public Object take() throws InterruptedException
   {
      lock.lock();


      try
      {

         while( aCount == 0 ) Thread.yield();


         Object x = aItems[ aTakeIndex ];
         if( ++aTakeIndex == aItems.length ) aTakeIndex = 0;
         --aCount;



         return x;
      }
      finally { lock.unlock(); }
   }
}
```

Student ID #:___________________________

Question 7 [10 marks]

You are asked to test the `Math.abs(int)` function. The description of this function is the following:

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned. Note that if the argument is equal to the value of Integer.MIN_VALUE, the most negative representable int value, the result is that same value, which is negative. The range of Java integer is contained between the values Integer.MIN_VALUE and Integer.MAX_VALUE.

a) Determine equivalence partitions and boundaries for this functions. The equivalence partitions must respect the coverage and disjointedness criteria. You should include all possible boundary cases. Provide your answer by completing the following table: [4]

Partition/Boundary Description	**Input value**	**Expected Output**

b) Complete the jUnit test method below to implement your equivalence partitions and boundaries. You should use the jUnit `assertEquals(int pExpected, int pValue)` method, which will automatically verify that `pExpected` is equal to `pValue` and take the appropriate steps if it is not. [3]

```
public void testAbs()
{
```

c) You're in charge of purchasing a test coverage analysis tool. Tool A offers statement coverage analysis for 200$ and tool B offers branch coverage analysis for 100$. Everything else being equal, which tool would you purchase to analyze the test coverage for your crib project. Justify your decision. [3]

Appendix 1: Excerpts from the Java 1.5.0 API – Set and Object

Set<E>.add

`public boolean add(E o)`

Adds the specified element to this set if it is not already present (optional operation). More formally, adds the specified element, `o`, to this set if this set contains no element `e` such that `(o==null ? e==null : o.equals(e))`. If this set already contains the specified element, the call leaves this set unchanged and returns `false`. In combination with the restriction on constructors, this ensures that sets never contain duplicate elements.

Object.equals

`public boolean equals(Object obj)`

Indicates whether some other object is "equal to" this one.

The `equals` method implements an equivalence relation:

- It is *reflexive*: for any reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is *transitive*: for any reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is *consistent*: for any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the object is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x==y` has the value `true`).

Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes.

hashCode

`public int hashCode()`

Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by `java.util.Hashtable`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

Appendix 2: Excerpts from the Java 1.5.0 API - Threads

public interface **Lock**

`Lock` implementations provide more extensive locking operations than can be obtained using `synchronized` methods and statements. They allow more flexible structuring, may have quite different properties, and may support multiple associated `Condition` objects.

Method Summary	
`void`	**`lock`**`()` *Acquires the lock.*
`void`	**`lockInterruptibly`**`()` *Acquires the lock unless the current thread is `interrupted`.*
`Condition`	**`newCondition`**`()` *Returns a new `Condition` instance that is bound to this `Lock` instance.*
`boolean`	**`tryLock`**`()` *Acquires the lock only if it is free at the time of invocation.*
`boolean`	**`tryLock`**`(long time, TimeUnit unit)` *Acquires the lock if it is free within the given waiting time and the current thread has not been `interrupted`.*
`void`	**`unlock`**`()` *Releases the lock.*

public interface **Condition**

`Condition` factors out the `Object` monitor methods (`wait`, `notify` and `notifyAll`) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary `Lock` implementations. Where a `Lock` replaces the use of `synchronized` methods and statements, a `Condition` replaces the use of the Object monitor methods.

Conditions (also known as *condition queues* or *condition variables*) provide a means for one thread to suspend execution (to "wait") until notified by another thread that some state condition may now be true. Because access to this shared state information occurs in different threads, it must be protected, so a lock of some form is associated with the condition. The key property that waiting for a condition provides is that it *atomically* releases the associated lock and suspends the current thread, just like `Object.wait`.

A `Condition` instance is intrinsically bound to a lock. To obtain a `Condition` instance for a particular `Lock` instance use its `newCondition()` method.

Method Summary	
`void`	**`await`**`()` *Causes the current thread to wait until it is signalled or `interrupted`.*
`boolean`	**`await`**`(long time, TimeUnit unit)` *Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.*
`long`	**`awaitNanos`**`(long nanosTimeout)` *Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.*
`void`	**`awaitUninterruptibly`**`()` *Causes the current thread to wait until it is signalled.*
`boolean`	**`awaitUntil`**`(Date deadline)` *Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses.*
`void`	**`signal`**`()` *Wakes up one waiting thread.*
`void`	**`signalAll`**`()` *Wakes up all waiting threads.*