Explanation

Some classes/interfaces from the j2sdk and junit:

```
public interface Iterator {
  public boolean hasNext();
  public Object next();
  public void remove();
}
public interface Collection {
  public int size();
  public boolean isEmpty();
  public boolean contains(Object o);
  public Iterator iterator();
  public boolean add(Object o);
  public boolean remove(Object o);
  public void clear();
  ...
}
public interface List extends Collection {
  public Object get(int index);
  public Object set(int index, Object element);
  public void add(int index, Object element);
  public Object remove(int index);
  ...
}
public interface Observer {
  public void update(Observable o, Object arg);
}
public class Observable {
  public synchronized void addObserver(Observer o);
  public void notifyObservers(Object arg);
  protected synchronized void setChanged();
  protected synchronized void clearChanged();
  public synchronized boolean hasChanged();
  ...
}
public class Assert {
  static public void assertTrue(boolean condition);
  static public void assertFalse(boolean condition);
  static public void fail();
  static public void assertEquals(Object expected, Object actual);
  static public void assertEquals(int expected, int actual);
  static public void assertNull(Object object);
  static public void assertNotNull(Object object);
  static public void assertSame(Object expected, Object actual);
  static public void assertNotSame(Object expected, Object actual);
  ...
}
```

Question 1

This question covers material which is not covered by your midterm (abstract classes)

Draw a single UML class diagram that describes the following situation. Make your diagram as precise as possible.

1. Interface `I` defines method `f`.
2. Interface `J` defines method `g`.
3. Class `C` implements `I` and `J`; `C` implements `f`; `C` does *not* implement `g`.
4. Class `D` extends `C` and implements `g`.
5. Each instance of class `X` maintains references to between zero and twenty instances of `I`.

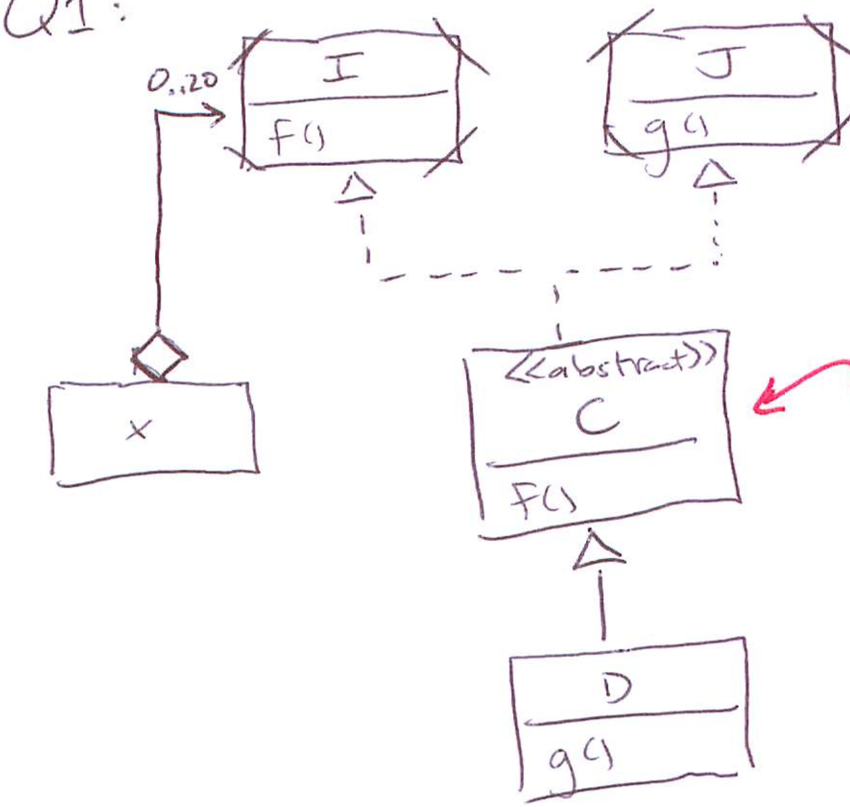
Question 2

This question covers material which is not covered by your midterm (old homework, with abstract classes)

Consider the following code from the homework assignments:

```
abstract public class AbstractUndoableCommand implements UndoableCommand {
  private CommandHistory _history;

  protected AbstractUndoableCommand(CommandHistory history) { _history = history; }
  abstract protected boolean setup();
  abstract protected void dodo();
  abstract public void undo();
  final public void execute() {
    if (setup()) {
      dodo();
      _history.addCommand(this);
    }
  }
  final public void redo() {
    dodo();
  }
}
public class CompositeCommand extends AbstractUndoableCommand {
  private final List _list;

  public CompositeCommand(CommandHistory history) {
     super(history);
    _list = new ArrayList();
  }
  public void addCommand(AbstractUndoableCommand cmd) { _list.add(cmd); }
  public boolean setup() { return true; }
  public void dodo() {
    Iterator i = _list.iterator();
    while (i.hasNext()) {
      AbstractUndoableCommand cmd = (AbstractUndoableCommand) i.next();
      cmd.dodo();
    }
  }
```

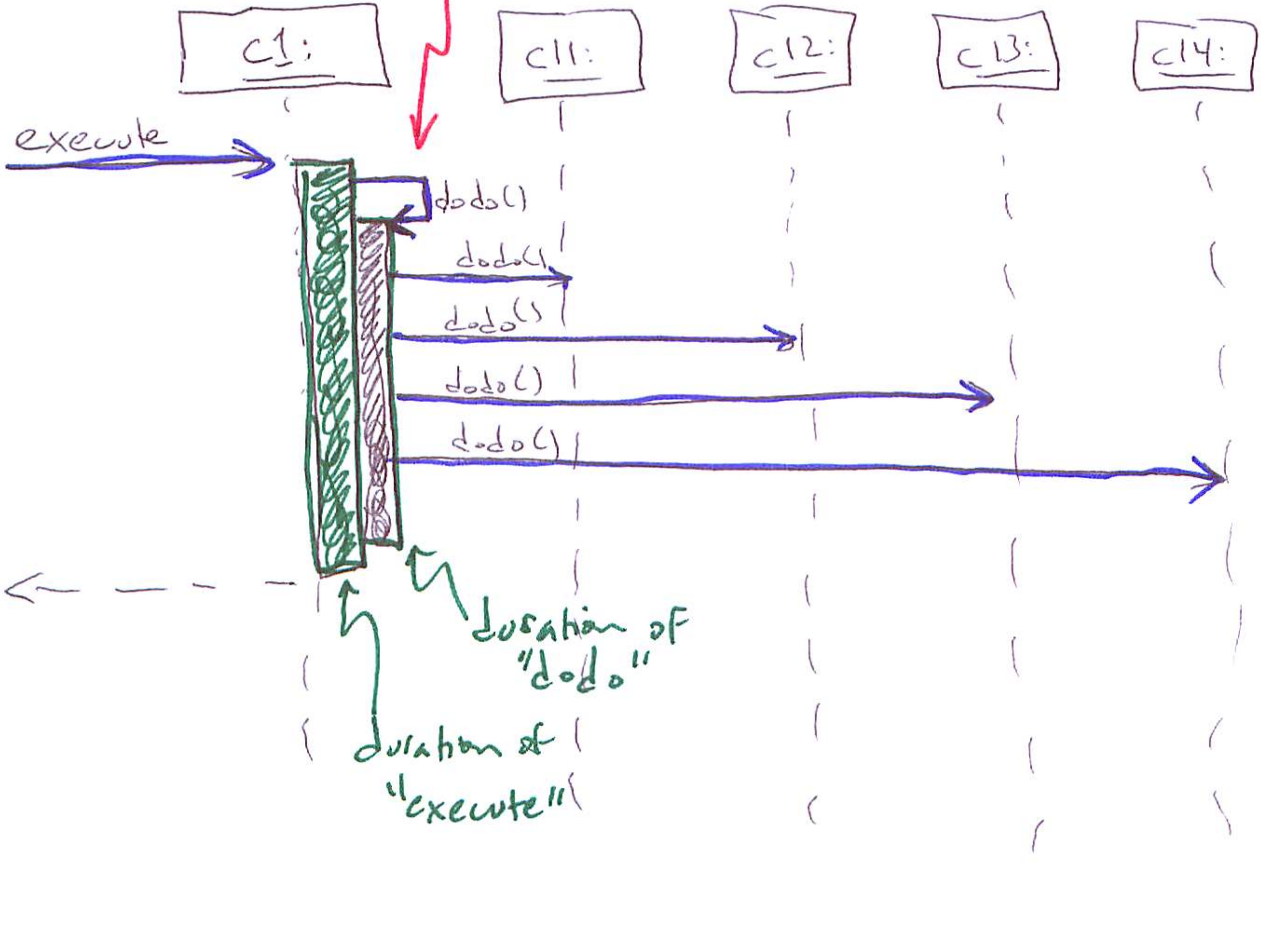
Q1:



C is abstract b/c does not implent g()

Q2

this is a call to self (or to "this")



duration of "dodo"

duration of "execute"

```
  public void undo() {
    ListIterator i = _list.listIterator(_list.size());
    while (i.hasPrevious()) {
      AbstractUndoableCommand cmdObject = (AbstractUndoableCommand)  i.previous();
      cmdObject.undo();
    }
  }
}
public class AddCommand extends AbstractUndoableCommand {
  private Database _db;       private int _year;
  private Video _newVideo;    private String _category;
  private String _title;      private int _numCopies;

  public AddCommand(Database db, String title, int year, String category,
  int numCopies) {
    super(db);                _year = year;
    _db = db;                 _category = category;
    _title = title;           _numCopies = numCopies;
  }
  protected boolean setup() {
    return (_db.findVideo(_title) == null);
  }
  public void dodo() {
    _newVideo = new VideoImpl(_title, _year, _category, _numCopies);
    _db.addVideo(_newVideo);
  }
  public void undo() {
    _db.removeVideo(_title);
  }
}
```

Given the following main program fragment

```
  AbstractUndoableCommand c11 = new AddCommand(db, "Vanishing Point", 1973, "Drama", 1);
  AbstractUndoableCommand c12 = new AddCommand(db, "American Graffiti", 1975, "Comedy", 3);
  AbstractUndoableCommand c13 = new AddCommand(db, "El Mariachi", 1996, "Drama", 2);
  AbstractUndoableCommand c14 = new AddCommand(db, "Play it again, Sam", 1978, "Comedy", 4);
  CompositeCommand c1 = new CompositeCommand(db);
  c1.addCommand(c11);
  c1.addCommand(c12);
  c1.addCommand(c13);
  c1.addCommand(c14);
  c1.execute();
```

draw an object interaction diagram tracing the method executions for the call `c1.execute()`. Your diagram should have columns corresponding to objects `c1`, `c11`, `c12`, `c13`, `c14`, with the leftmost column being `c1`. Do not include any other objects.

Question 3

In the following code, circle the uses of `x0`, `x1`, `x2` and `x3` that are *not* allowed by the java compiler; that is, circle the field usages that *do not compile*.

note the packages! declaration are ok, just circling to highlight them.

```
package one;
public class A {
    private int x0;
            int x1;
     public int x3;

  int f1(A that) {
    return this.x0 + this.x1  + this.x3
         + that.x0 + that.x1  + that.x3;
  }
}
```

all ok.

```
package one;
class B {
  int g1(A that) {
    return that.x0 + that.x1 + that.x3;
  }
}
```

not ok, private

```
package two;
class D {
  int g2(one.A that) {
    return that.x0 + that.x1 + that.x3;
  }
}
```

not ok. private or package private

Question 4

In this problem, we will look at a data structure for representing simple organizational charts (org charts).

```
interface Node { int size(); }

class P implements Node { // a node consisting of a single p(erson)
  private String _name;
  public P(String name) { _name = name }
  public int size() {/* TODO */}
}

class OU implements Node { // an organizational unit: a node with zero or more children
  private String _name;     // (any type of nodes including P and OU)
  private List _children = new LinkedList();
  public OU(String name) { _name = name; }
  public Iterator getChildren() { _children.iterator(); }
  public void addChild(Node node) { _children.add(node); }
  public int size() {/* TODO */}
}
```

1. Using the classes P and OU, write code that builds an org chart for the following portion of a College (you may use the space to the right of the following tree):
   - The college itself
     - Malcolm (dean)
     - Arts department

  - Nancy (Chairman)
  - Xiaoping (Professor)
- Music department
  - Srinivas (Chairman)
  - Bob (Professor)

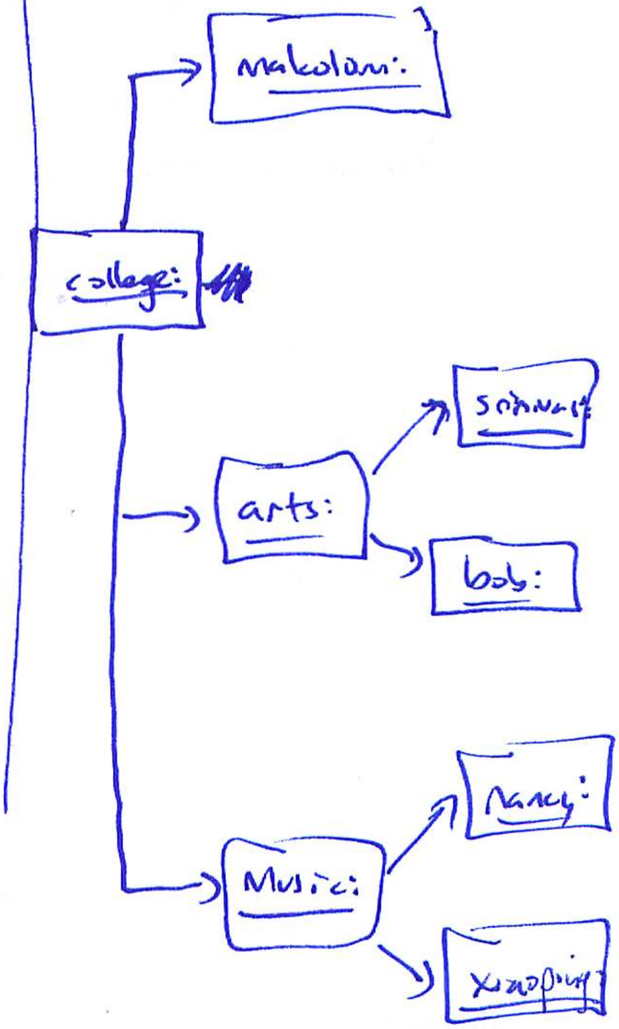
2. Recall that you can iterate over the elements in a list `x` as follows:

```
Iterator i = x.iterator();
while (i.hasNext()) {
  Node current = (Node) i.next();
  // do something with current
}
```

Assuming that each person is in at most one organizational unit, complete the `size()` method in the classes `P` and `OU`. Write your answers below.

```
class P implements Node { // ...
  public int size() {

       return 1

  }
}

class OU implements Node { // ...
  public int size() {
       int result = 0
       For (Node n : _children)
            result += n.size()

  }
}
```

Part 1:



Question 5

Consider the interface `Predicate` defined as follows.

```
interface Predicate {
  boolean eval(int j);
}
```

1. Recall that you can check if an integer "`i`" is even by using the expression "`i%2 == 0`". Write a class

`IsEven` that determines whether a number is even:

```
class IsEven implements Predicate {
  public boolean eval(int j) {
          return  j%2 == 0

   }
}
```

For example, the following code

```
  Predicate p = new IsEven();
  if (  p.eval(2)) { System.out.println("2 is even"); }
  if (! p.eval(3)) { System.out.println("3 is not even");
```

should produce the output

```
  2 is even
  3 is not even
```

2. Write a class `Alternate` that alternates between `true` and `false`, starting with `true`. You may add fields if necessary.

```
class Alternate implements Predicate {
    boolean  state = false

  boolean eval(int j){
          state  =  !state
          return  state

  }
}
```

For example, the following code

```
  Predicate p = new Alternate();
  in = new DataInputStream(System.in);

  for (int k=0; k<4; k++){
    int j = in.readInt();   // read a number from the user
    if (p.eval(j)) {
      System.out.println("true");
    } else {
      System.out.println("false");
    }
  }
```

should produce the following output, no matter what input is given:

```
true
false
true
false
```

3. Write a class `Not` that implements logical negation:

```
class Not implements Predicate {
  Predicate _p;

  Not(Predicate p) { _p =p; }
  boolean eval(int j){

          return ! p.eval(j)


  }
}
```

For example,

```
Predicate p = new IsEven();
Predicate q = new Not(p);
if (! q.eval(2)) { System.out.println("2 is even"); }
if (  q.eval(3)) { System.out.println("3 is not even");
```

should produce the same output as before, even though the negation operator (`!`) is moved with respect to question 1:

```
2 is even
3 is not even
```

As another example, the code

```
Predicate p = new Alternate();
Predicate q = new Not(p);
in = new DataInputStream(System.in);

for (int k=0; k<4; k++){
  int j = in.readInt();  // read a number from the user
  if (q.eval(j)) {
    System.out.println("true");
  } else {
    System.out.println("false");
  }
}
```

should produce the following output, no matter what input the user gives:

```
false
true
false
true
```

Notice that the outputs are just flipped, from `true` to `false` and from `false` to `true`, in comparison with question 2.

Question 6

In this problem, we will look at the following "`Stream`" iterator.

```
abstract class Stream implements Iterator {
  public final boolean hasNext() { return true; }
}

class IntegerStream extends Stream {
  private int _i = -1;
  public Object next() { _i++; return new Integer(_i);}
}
```

See the comments for the output of the following code:

```
  IntegerStream I = new IntegerStream();
  System.out.println(I.next());   // prints 0 on the screen
  System.out.println(I.next());   // prints 1 on the screen
  System.out.println(I.next());   // prints 2 on the screen
  System.out.println(I.next());   // prints 3 on the screen
  System.out.println(I.next());   // prints 4 on the screen
```

1. Finish the code of class `FilteredStream` implementing `Iterator` so that the following code fragments work as indicated:

   ```
   IntegerStream I = new IntegerStream();
   FilteredStream F = new FilteredStream(I, new IsEven());
   System.out.println(F.next());   // prints 0 on the screen
   System.out.println(F.next());   // prints 2 on the screen
   System.out.println(F.next());   // prints 4 on the screen
   System.out.println(F.next());   // prints 6 on the screen
   System.out.println(F.next());   // prints 8 on the screen

   IntegerStream J = new IntegerStream();
   J.next();                       // move forward one item in J
   FilteredStream G = new FilteredStream(J, new IsEven());
   System.out.println(G.next());   // prints 2 on the screen
   System.out.println(G.next());   // prints 4 on the screen
   System.out.println(G.next());   // prints 6 on the screen
   System.out.println(G.next());   // prints 8 on the screen

   IntegerStream K = new IntegerStream();
   class Div3 implements Predicate {
     public boolean eval(int n) { return (n%3) == 0; }
   }
   FilteredStream H = new FilteredStream(K, new Div3());
   System.out.println(H.next());   // prints 0 on the screen
   System.out.println(H.next());   // prints 3 on the screen
   System.out.println(H.next());   // prints 6 on the screen
   System.out.println(H.next());   // prints 9 on the screen
   ```

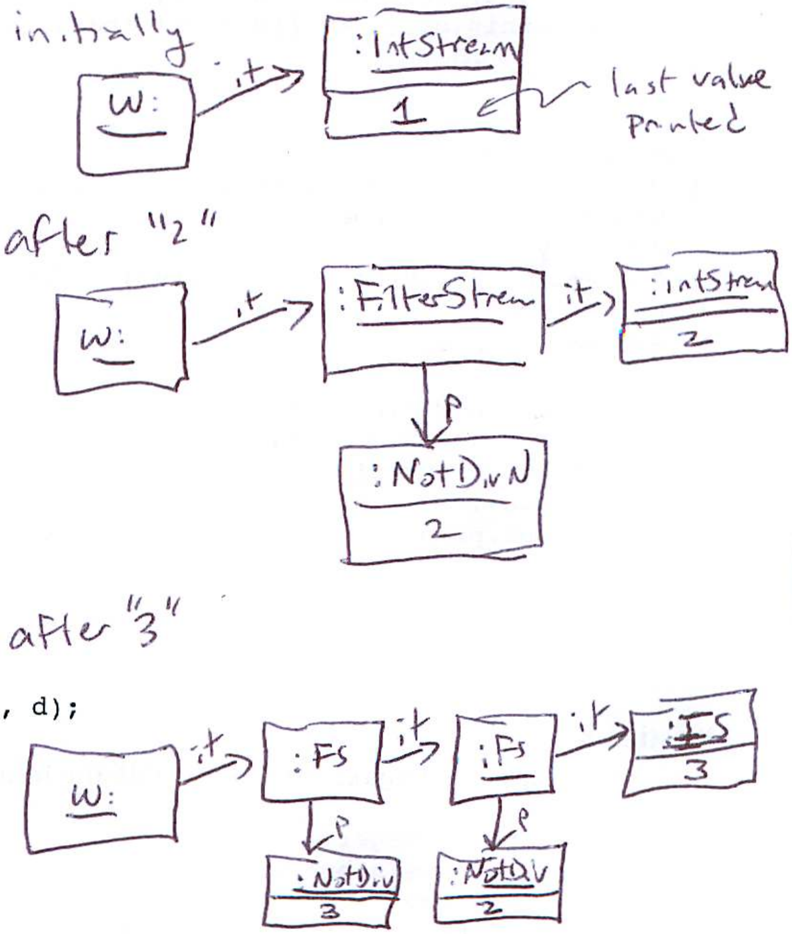
   Your job is to write the method `next()` in the following code.

```
class FilteredStream extends Stream {
  private Stream _it;
  private Predicate _p;

  public FilteredStream(Stream it, Predicate p) {
    _it = it;
    _p =p;
  }

  public Object next() {
      int result;
      do { result = it.next()
       } while (! p.eval(result))
      return result;
  }
}
```

[Ignoring conversion between Object & int]

2. Consider the following classes:

```
class NotDivn implements Predicate {
  final private int _n;
  NotDivn(int n) {
    _n = n;
  }
  public boolean eval(int m) {
    return (m%_n) != 0;
  }
}
class WhatAPain extends Stream {
  private Stream _it;

  public WhatAPain(Stream it) {
    _it = it;
  }

  public Object next() {
    final int n = _it.next();
    final Predicate d = new NotDivn(n);
    Stream newit = new FilteredStream(_it, d);
    _it = newit;
    return (new Integer(n));
  }
}
```



What does the following code print?

```
IntegerStream I = new IntegerStream();
System.out.println(I.next());   // prints 0 on the screen
System.out.println(I.next());   // prints 1 on the screen

WhatAPain w = new WhatAPain(I);
```

etc...

```
    System.out.println(w.next());
    System.out.println(w.next());
    System.out.println(w.next());
    System.out.println(w.next());
    System.out.println(w.next());
    System.out.println(w.next());
```

2
3
5
7
11
13

not 4 b/c divisible by 2
not 6 b/c divisible by 3
not 8, 9, 10 b/c divisible by 2 or 3
not 12 b/c divisible by 3

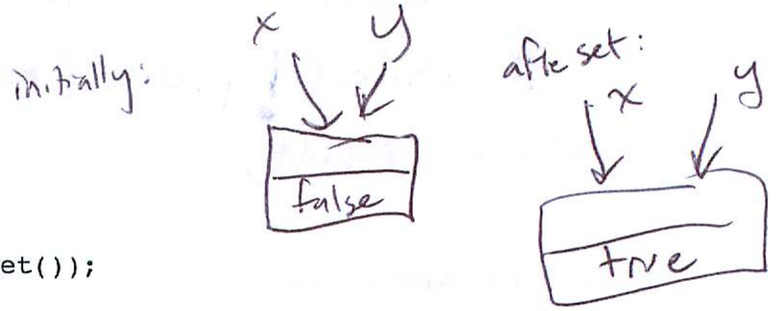
Question 7

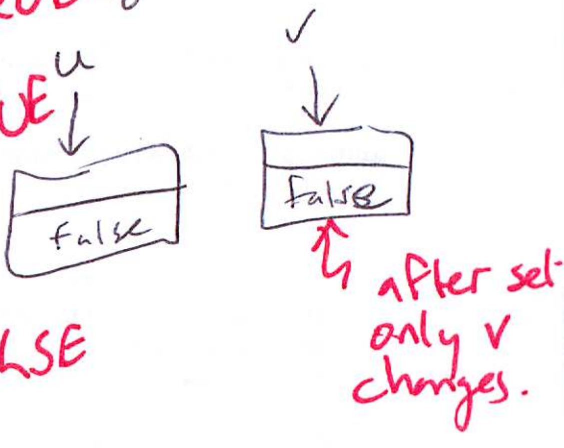
What is the output of the following Java code?

Briefly justify your answer.

```
class MyBool {
  private boolean _v;
  public MyBool(boolean v) { set(v); }
  public void set(boolean v) { _v = v; }
  public boolean get() { return _v; }
  public boolean equals(Object that) {
    return (that instanceof MyBool)
        && (this.get() == ((MyBool) that).get());
  }
}

public class Main {
  public static void main(String[] args) {
    MyBool x = new MyBool(false);
    MyBool y = x;
    System.out.println( (x.equals(y)) + "," + (x == y) );
    y.set(true);
    System.out.println( (x.equals(y)) + "," + (x == y) );

    MyBool u = new MyBool(false);
    MyBool v = new MyBool(false);
    System.out.println( (u.equals(v)) + "," + (u == v) );
    v.set(true);
    System.out.println( (u.equals(v)) + "," + (u == v) );
  }
}
```

TRUE
TRUE
TRUE
FALSE





after set only v changes.

Question 8

Consider the class `NonNegativeInteger` with the following interface:

```
class NonNegativeInteger {
  public NonNegativeInteger();
  public boolean equals (Object that);
  public void set(int v) throws IllegalArgumentException;
  public int get();
}
```

The invariants for the class are:

Q8:

```
public void test1 ():

        NNI x = new NNI ()
        NNI y = new NNI ()

        AssertEquals (0, x.get())
        ~~assertEquals (0, y.get())~~

        x.set (5)
        assertEquals (5, x.get())
cait set
negative
        try {
            x.set (-1) ; Fail ()
        } catch (IllegalArgumentException e) { }

        x.set (6)
        assertEquals (6, x.get())
        ~~assertNotEquals (x.get(), y~~
        assertFalse (x.equals (y))
        y.set(6)
        assertTrue (x.equals (y))
        assertFalse (x.equals (new Object()))
        assertFalse (x.equals (null))
```

- if set has not been called, get should return 0
- if set has been called, get should return the value of the last set
- get should never return a negative value

In this question you must write tests for NonNegativeInteger.

- Write tests to check that this class obeys all of its invariants.
- Write a test to check the equals operation. (It should return true exactly when this.get() is the same as that.get().)

You may use methods from the Assert class summarized on the last page of the exam.

```
public class NonNegativeIntegerTEST extends TestCase {
  public void test1() {
```

Question 9

This question covers material which is not covered by your midterm (observer)

Consider the following class:

```
class MutableInteger {
  private int _v;
  public void set(int v) { _v = v; }
  public int get() { return _v; }
}
```

Suppose that you are required to modify MutableInteger to perform some function each time a particular MutableInteger is set. However, you do not know the exact function to be performed.

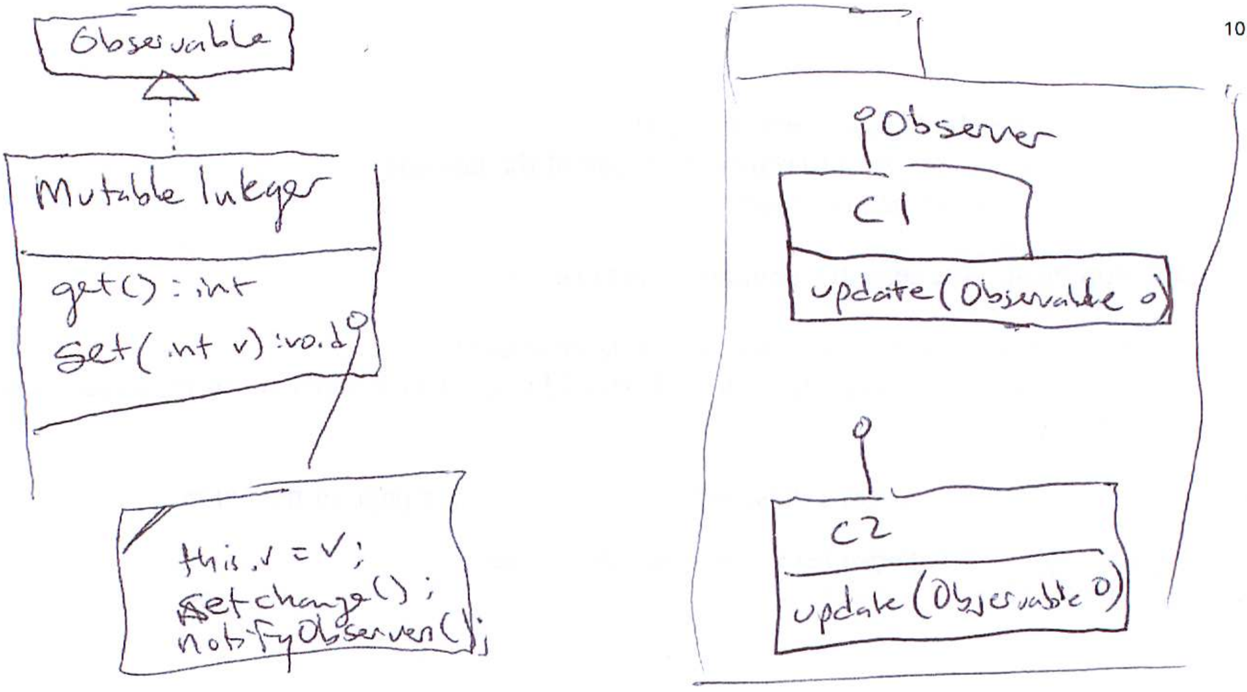
(For example, it might be neccesary to print some changes to the screen, or to keep count of the number of times set has been invoked.)

1. What pattern can help you re-write MutableInteger to suit these requirements? Observer
2. Draw a UML class diagram sketching your solution. Draw your classes within a package.

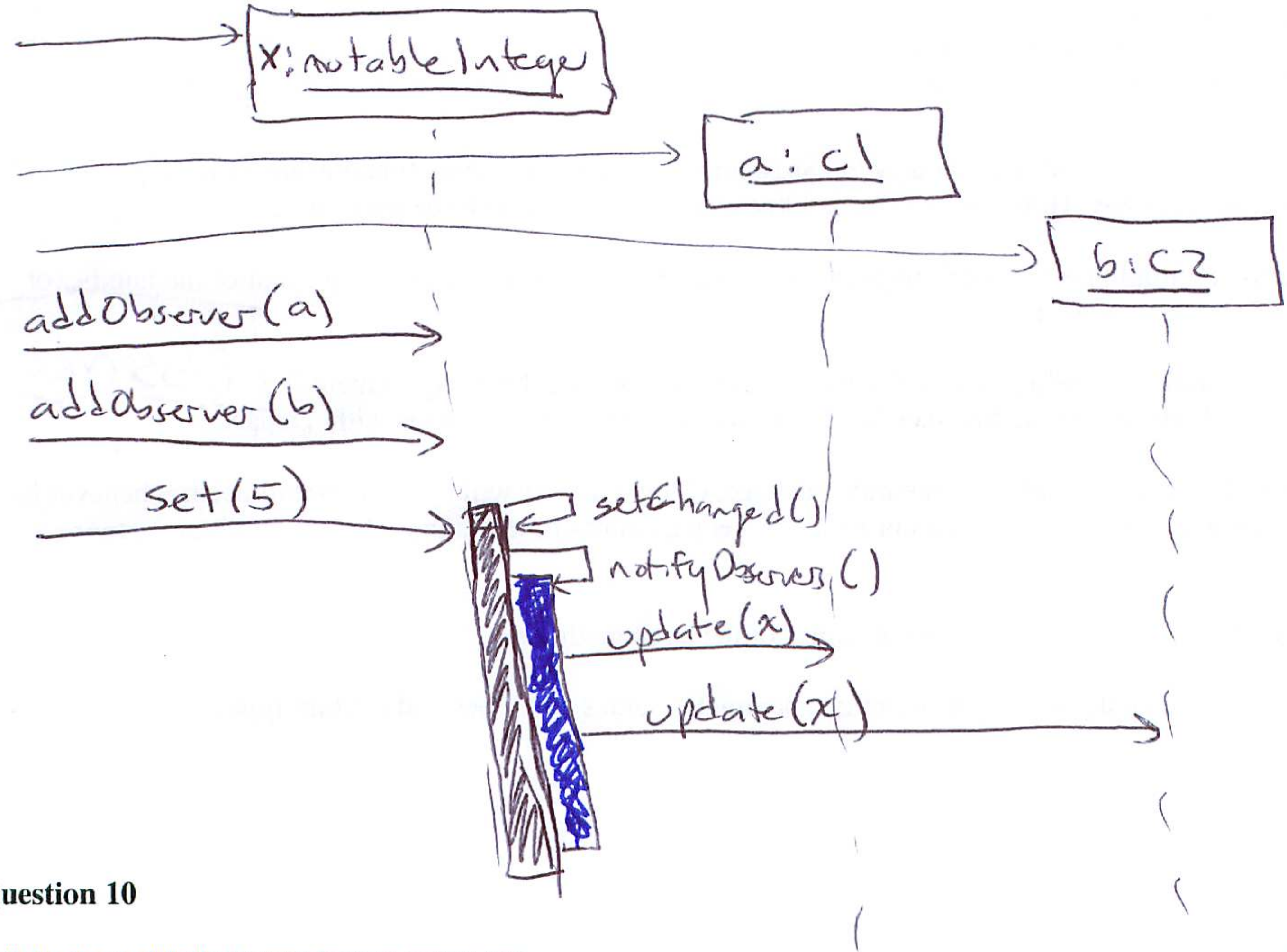
   Show two client classes in a separate package. Client class C1 will cause a print to occur whenever its MutableInteger is set; Client class C2 will keep a count of the number of times its MutableInteger is set.

   In each class you *must include* the names of declared methods.

   For each method you *must also* include parameter names and types and a return type.



3. Draw a UML sequence diagram showing the interactions of a program that:
   - creates one instance each of `MutableInteger`, `C1` and `C2`, informing `C1` and `C2` to keep track of `set` methods on the `MutableInteger`.
   - calls `set` on the `MutableInteger`.



Question 10

For the purposes of this problem, assume that you are given classes `MyStack` and `MyQueue` to implement interface `MyContainer` in the expected way.

```
interface MyContainer {
  public Object get();       // return item in container
  public void add(Object x); // add item into container
  public void remove();      // remove item from container
  public boolean isEmpty();  // check if container is empty
}
```

You are given the following classes to establish a tree. The method `children()` returns an iterator that is empty for leaves, and returns first the left, then right child for internal nodes.

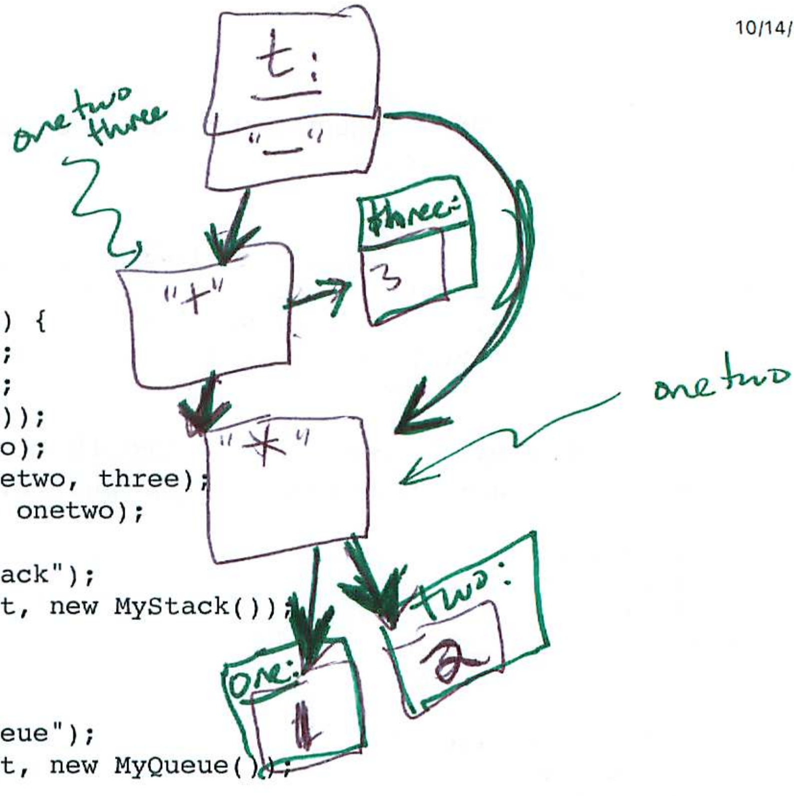
```
import java.util.ArrayList;
import java.util.Iterator;
interface Tree {
  public void print();
  public Iterator children();
}
class Node implements Tree {
  private String _v;
  private Tree _l, _r;
  public Node(String v, Tree l, Tree r) { _v = v; _l =l; _r =r; }
  public void print() { System.out.println(_v); }
  public Iterator children() {
    ArrayList A = new ArrayList();
    A.add(_l); A.add(_r);
    return A.iterator();
  }
}
class Leaf implements Tree {
  private Integer _v;
  public Leaf(Integer v) { _v = v; }
  public void print() { System.out.println(_v); }
  public Iterator children() {
    return new ArrayList().iterator();
  }
}
```

You must write the output of the main program on the following page.

Consider the following implementation of `Iterator` and class `Main`.

```
import java.util.Iterator;
class TreeIterator implements Iterator {
  private MyContainer _c;
  public TreeIterator(Tree t, MyContainer c) {
    _c = c; _c.add(t);
  }
  public boolean hasNext() { return ! _c.isEmpty(); }
  public void remove() { throw new UnsupportedOperationException(); }
  public Object next() {
    Tree top = (Tree) _c.get();
    _c.remove();
    Iterator i = top.children();
    while (i.hasNext())
```

```
      _c.add(i.next());
    return top;
  }
}

import java.util.Iterator;
class Main {
  public static void main(String[] argv) {
    Tree one = new Leaf(new Integer(1));
    Tree two = new Leaf(new Integer(2));
    Tree three = new Leaf(new Integer(3));
    Tree onetwo = new Node("*", one, two);
    Tree onetwothree = new Node("+", onetwo, three);
    Tree t = new Node("-", onetwothree, onetwo);

    System.out.println("Iterate with Stack");
    TreeIterator i1 = new TreeIterator(t, new MyStack());
    while (i1.hasNext())
      ((Tree) (i1.next())).print();

    System.out.println("Iterate with Queue");
    TreeIterator i2 = new TreeIterator(t, new MyQueue());
    while (i2.hasNext())
      ((Tree) (i2.next())).print();
  }
}
```



What is the output generated by `Main`?

Stack:
− * 2 1 + 3 * 2 1

queue:
− + * * 3 1 2 1 2

Question 11

Consider the following interfaces for functions on integers and integer arrays.

```
interface IntFun { public int   exec(int   x); }
interface ArrFun { public int[] exec(int[] x); }
```

Here are two classes, one implements an absolute value function, the other a cube function:

```
class Abs implements IntFun {
  public int exec(int x) { return (x < 0) ? -x : x; }
}
class Cube implements IntFun {
  public int exec(int x) { return x*x*x; }
}
```

Complete the following two questions. There is an example with output on the next page.

1. Complete the following definition of `Comp` which composes two functions. For example, `new Comp(new Abs(), new Cube()).exec(-3)` should return 27 (computing the absolute value of -3 and the cubing it).

```
class Comp implements IntFun {
  IntFun _f, _g;
  public Comp(IntFun f, IntFun g) { _f = f; _g = g; }
  public int exec(int x) {
```

```
return g.exec ( f.exec ( x ))
```

2. Complete the following definition of `Map` which performs a integer function `f` on every element of an array, returning a new array.

```
class Map implements ArrFun {
  IntFun _f;
  public Map(IntFun f) { _f = f; }
  public int[] exec(int[] x) {
      int[] result = new int[ x.length]
      For (int i = 0; i< x.length ; i++)
            result[i] = f. exec( x [i])
      return result.
```

As an example, the following code prints:

```
[ 0 1000 8000 27000 64000 125000 216000 343000 512000 729000 ]

class Main {
  public static void print(int[] x) {
    System.out.print("[ ");
    for (int i=0; i<x.length; i++)
      System.out.print(x[i]+ " ");
    System.out.println("]");
  }
  public static void main(String[] argv) {
    int[] a = new int[10];
    for (int i=0; i<a.length; i++)
      a[i] = -i*10;

    ArrFun mabs = new Map(new Comp(new Abs(), new Cube()));
    print(mabs.exec(a));
  }
}
```

Question 12

This problem is about the construction of a simple music library. You are given the following class to start off things:

```
class Music {
  static void play(int duration) {/*...*/}
    // play note at the current pitch for the given duration
    // in milliseconds (the initial pitch is A = 440 Hz)
  static void rest(int duration) {/*...*/}
    // rest for given duration
  static void scalePitch(double factor) {/*...*/}
    // multiply the pitch frequency by the given factor
    // (a factor less than one will lower the pitch)
  static void reset() {/*...*/}
    // reset the pitch to note A = 440 Hz
}
```

For example:

```
Music.reset();               // initialize pitch to middle A (440 Hz)
Music.play(500);             // play a middle A for half a second
Music.rest(1000);            // rest for one second
Music.scalePitch(2.0);       // set pitch an octave higher (880 Hz)
Music.play(500);             // play a high A for half a second
Music.rest(250);             // rest for a quarter of a second
Music.scalePitch(1.0/2.0);   // reset pitch to middle A (440 Hz)
Music.play(500);             // play a low A for half a second
```

In this problem, we will write code to build and manipulate complex musical objects that are built out of notes and rests. The basic interface is `Event`, and one implementing class is `Note`:

```
interface Event {
  public void play();
}
class Note implements Event {
  int _d;
  double _f;
  public Note(int duration, double factor) {
    _d = duration;
    _f = factor;
  }
  public void play() {
    Music.scalePitch(_f);
    Music.play(_d);
    Music.scalePitch(1.0/_f);
  }
}
```

Now we look into mechanisms for building rests and more complex musical objects. Answer the following questions.

1. Following `Note`, write a class `Rest` that, when played, will rest for the given duration. Make the duration

a parameter of the constructor.

```
class Rest implements Event {
   int d
   public Note (int duration) { d = duration; }
   public void play() { Music.rest(d);}
```

2. Finish the following code for the `EventGroup` class:

```
class EventGroup implements Event {
  List _events = new LinkedList();
  public void add(Event e) {
        events.add(e)
  }
  public void play() {
          For (Event e : events)
              e.play()
```

3. Name the pattern most closely associated with `EventGroup`. COMPOSITE
4. Write a class `Transpose` that, when played, will play an event at a scaled pitch. Make the event to be transposed and the scaling factor both parameters of the constructor.

   This question covers material which is not covered by your midterm (decorator)

```
class Transpose implements Event {
   Event it;
   double Factor;
    public Transpose (Event it, double Factor){
            this.it = it;   this.factor = Factor;
    }
    public void play() { Music.salePitch(F)
                         it.play()
                         Music.scalePitch(1.0/F) }
                                                 }
```

5. Name the pattern most closely associated with `Transpose`. DECORATOR

   This question covers material which is not covered by your midterm (decorator)

6. Using these classes, write Java code that declares a variable `e1` and construct an object such that "`Music.reset(); e1.play();`" does the following:
   - play for 0.25 second a note at 880.0 Hz.
   - rest for 0.25 second
   - play for 0.5 second a note at 440.0 Hz.
   - rest for 0.5 second
   - play for 1.0 second a note at 220.0 Hz.

```
EventGroup e1 = new EventGroup ()
e1. add (new Note (250, 2.0))
e1 .add (new Rest (250))
e1. add ( new Note (500,1.0))
e1.add (new Rest (500))
e1.add (new Note (1000, 0.5))
```

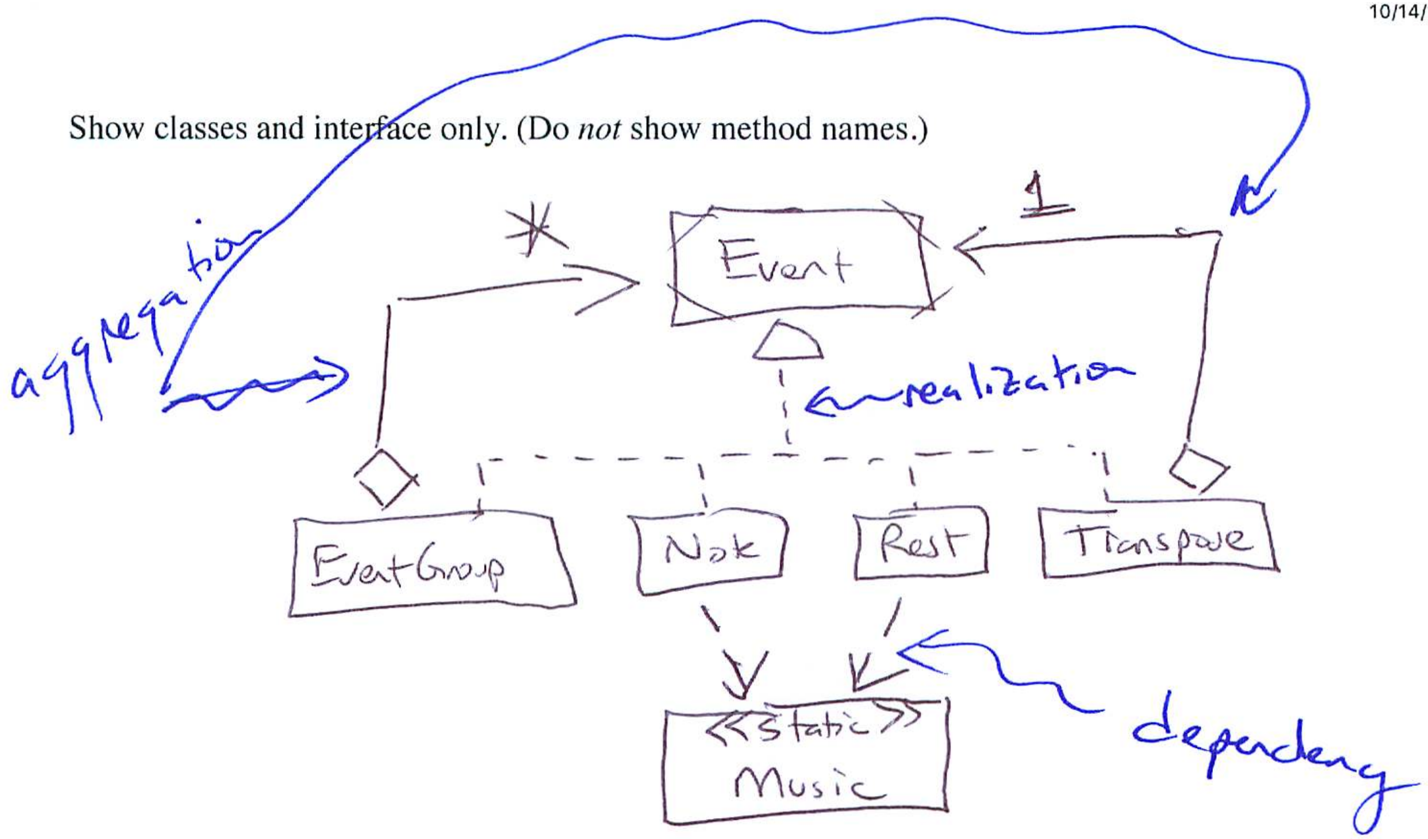
7. Using these classes and `e1`, write Java code that declares a variable `e2` and construct an object such that "`Music.reset(); e2.play();`" does the following:
   - play the events of `e1`
   - rest for 1.0 second
   - play the events of `e1` transposed by a factor of 2.0

   This question covers material which is not covered by your midterm (decorator)

```
EventGroup e2 = new EventGroup ()
 e2. add (e1)
 e2. add ( new Rest (1000))
e2. add (e1)
```

8. Draw a UML class diagram showing the relationships between these classes.

Show classes and interface only. (Do *not* show method names.)



This document was translated from LATEX by HEVEA.