SE 450 Winter 2002
Midterm Exam

March 19, 2002

Name: _____

Directions: Write your name in the blank at the top of this page.
You have **two hours and thirty minutes** (2:30) to take this exam.
It is **closed book and closed notes**.
Write your answers **on the exam**.
There are 7 questions, some with mutiple parts.
If I can not read your answer clearly, then it will be marked as incorrect.

Question	Value	Score
1	8	
2	8	
3	7	
4	7	
5	7	
6	7	
7	7	
8	7	
9	7	
10	7	
11	7	
12	7	
13	7	
14	7	
Total	100	

Question 1
Draw a single UML class diagram that describes the following situation. Make your diagram as precise as possible.

(a) Interface `I` defines method `f`.

(b) Interface `J` defines method `g`.

(c) Class `C` implements `I` and `J`; `C` implements `f`; `C` does *not* implement `g`.

(d) Class `D` extends `C` and implements `g`.

(e) Each instance of class `X` maintains references to between zero and twenty instances of `I`.

Question 2
Consider the following definitions.

```
interface Stack {
  bool isEmpty();
  void push(Object i);
  Object pop();
}
class Queue {
  public bool isEmpty() { /* ... */ }
  public void enqueue(Object i) { /* put item at rear */ }
  public Object dequeue() { /* remove & return front item */ }
  // some implementation
}
class StackImpl extends Queue implements Stack {
  public void push(Object i) { /* put item at front */ }
  public Object pop() { return this.dequeue(); }
}
```

For each question, answer true or false, with a short ONE sentence justification that fits in the space provided.

(a) Is `StackImpl` a subclass of `Queue` in Java? In other words, does Java allow you to use the code for `Queue` when defining the code for `StackImpl`?

(b) Is `StackImpl` conceptually a subtype of `Queue`? For example, would `StackImpl` work well in a supermarket?

(c) Is `StackImpl` a subtype of `Queue` in Java? In other words, does Java allow you to substitute an instance of `StackImpl` where an instance of `Queue` is required (e.g. `Queue q = new StackImpl();`)?

(d) Is `StackImpl` conceptually a subtype of `Stack`? For example, would `StackImpl` work well as a stack of plates?

(e) Is `StackImpl` a subtype of `Stack` in Java? In other words, does Java allow you to substitute an instance of `StackImpl` where an instance of `Stack` is required?

Question 3
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();
    }
  }
  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 4
Consider the following code:

```
interface SummingDice {
  int roll();  // roll two dice; return the value of the roll
  int sum();   // return the sum of all rolls
}
interface Player {
  int takeTurn (SummingDice d);
}

// A generic player that rolls until he goes bust, or decides to stop...
abstract class AbstractPlayer implements Player {
  public int takeTurn (SummingDice d) {
    while (true) {
      if (d.roll() == 2)
        return (0);      // player looses
      if (!rollAgain(d))
        break;
    }
    return d.sum(); // player gets points
  }
  // Should we roll again?
  protected abstract boolean rollAgain (SummingDice d);
}

// A player that never rolls more than once.
class CautiousPlayer extends AbstractPlayer {
  protected boolean rollAgain (SummingDice d) {
    return false;
  }
}

// A player that rolls until he accumulates 1000 points (or goes bust trying).
class GreedyPlayer extends AbstractPlayer {
  protected boolean rollAgain (SummingDice d) {
    return d.sum() < 1000;
  }
}
```

(a) What pattern does this code use?

(b) Rewrite the code to use the Strategy Pattern. (You may change all of the class names; you must maintain only the interface.)

Question 5

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 */}
}
```

(a) 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
  - Dean Top
  - Gregory (office manager)
  - Arts department
    - Chair One
    - Jamie
    - KJaggu
  - Music department
    - Chair Mozart
    - Eminem

(b) 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() {









  }
}

class OU implements Node { // ...
  public int size() {









  }
}
```

Question 6
Consider the interface `Predicate` defined as follows.

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

(a) Write a class `IsEven` so that the following code:

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

produces output

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

```
class IsEven implements Predicate {
  public boolean evaluate(int j) {









   }
}
```

(b) Write a class `Alternate` so that the following code

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

for (int k=0; k<6; k++){
  int j = in.readInt();  // read a number from the user
  if (p.evaluate(j)) {
    System.out.println("true");
  } else {
    System.out.println("false");
  }
}
```

produces the following output, no matter what input the user gives:

```
true
false
true
false
true
false
```

Thus, the `evaluate` method in class `Alternate` alternates between `true` and `false`: it returns `true` the first time around, `false` the second time, `true` again the third time etc.

You may add fields if necessary.

```
class Alternate implements Predicate {



  boolean evaluate(int j){









  }
}
```

(c) Write a class `Not` implementing `Predicate`, so that the following code

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

produces output

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

Notice the extra negation in the guards of the conditionals in the above program relative to question a.

As another example, the following code:

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

for (int k=0; k<6; k++){
  int j = in.readInt();  // read a number from the user
  if (q.evaluate(j)) {
    System.out.println("true");
  } else {
    System.out.println("false");
  }
}
```

produces the following output, no matter what input the user gives:

```
false
true
false
true
false
true
```

Notice that the outputs are just flipped, from true to false and false to true, in comparing with question b.

Finish the code in the following class:

```
class Not implements Predicate {
  Predicate _p;

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









  }
}
```

Question 7

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
```

(a) 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 evaluate(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() {









  }
}
```

(b) Consider the following classes:

```
class NotDivn implements Predicate {
  final private int _n;
  NotDivn(int n) {
    _n = n;
  }
  public boolean evaluate(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 = ((Integer) _it.next()).intValue();
    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);

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

(intentionally blank)

(intentionally blank) 16

(intentionally blank) 17