Comp 210 – Introduction to Object-Oriented Programming EXAM #1 NAME: __SOLUTION KEY_____
Rice University - Instructors: Nguyen & Wong February 18, 2004

Instructions

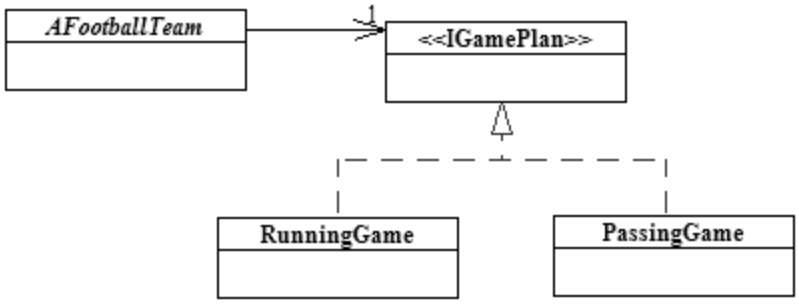
1. This exam is conducted under the Rice Honor Code. It is a closed-notes, closed-book exam.
2. Fill in your name on every page of the exam.
3. If you forget the name of a Java class or method, make up a name for it and write a *brief* explanation in the margin.
4. You are expected to know the syntax of defining a class with appropriate fields, methods, and inheritance hierarchy. You will not be penalized on trivial syntax errors, such as missing curly braces, missing semi-colons, etc, but do try to write Java code as syntactically correct as possible. We are more interested in your ability to show us that you understand the concepts than your memorization of syntax!
5. Write your code in the most object-oriented way possible, that is, with the fewest number of control statements and no checking of the states and class types of the objects involved.
6. For each algorithm you are asked to write, 90% of the grade will be for correctness, and 10% will be for efficiency and code clarity.
7. You have two hours and a half to complete the exam.

Please State and Sign your Pledge:

1a) 10	1b) 10	2a) 5	2b) 5	2c) 10	3) 20	4a) 10	4b) 10	5) 20	TOTAL 100

1. (20 pts total) In the spring, a young man's fancy turns to…life after the SuperBowl. But the memories of a Sunday in January linger. Football teams like the Panthers and the Patriots each have a specific "game plan" which they hope will enable them to win. For instance, the Panthers were noted during the season as a team whose game plan focused on running the ball, while the Patriots were noted for favoring game plans that emphasized passing the ball.

   a. (10 pts) Draw a UML class diagram that illustrates the basic relationships between a football team, a game plan and game plans that emphasize running or passing. Omit any methods or fields from your diagram.



   b. (10 pts) *In a couple of sentences*, describe all the design patterns used in your design for part a). Clearly explain your reasons for using them.

Union design pattern: `IGamePlan` is the union of its subclasses, `RunningGame` and `PassingGame`
Strategy design pattern: `AFootballTeam` uses (delegates to) `IGamePlan` as a strategy to win football games.

2. (20 pts total) Consider the following code:

```
public interface IStockbroker {
    public abstract String getAdvice();
}

public class Schwab implements IStockbroker {
    public String getAdvice() {
        return “buy Microsoft!”;
    }
}

public class MerrillLynch implements IStockbroker {
    public String getAdvice() {
        return “sell ImClone!”;
    }
}

public class Investor {
    private IStockbroker _sb;
    public Investor(IStockbroker sb) {
        _sb = sb;
    }

    public void setStockBroker(IStockbroker sb) {
        _sb = sb;
    }

    public String tradeStock() {
        return “I am going to ”+_sb.getAdvice();
    }
}
```

a) (5 pts) What design pattern(s) is(are) being implemented in the above system of classes?

Union design pattern: `IStockbroker` is the union of `MerrillLynch` and `Schwab`.
Strategy design pattern: the `Investor` uses (delegates to) the `IStockbroker` as a strategy for investing.

b) (5 pts) What will appear on the screen when the follow code is executed?

```
Investor aPerson = new Investor(new Schwab());
System.out.println(aPerson.tradeStock());
aPerson.setStockbroker(new MerrillLynch());
System.out.println(aPerson.tradeStock());
```

```
I am going to buy Microsoft!
I am going to sell ImClone!
```

c) (10 pts) What classes represent the invariant behaviors? What classes represent the variant behaviors? Explain your answers in a few short sentences.

`Investor` and ***`IStockbroker`*** represent the invariant behaviors because their code is independent of which concrete type of stockbroker is being used. `MerrillLynch` and `Schwab` represent the variant behaviors because their code each has a specific sort of investment strategy.

3. (20 pts) A function is an abstract mathematical computation that takes an input, performs some calculation and produces an output. In theoretical computer science, such a function is called a "lambda." In OOP, we can model abstract functions as an interface defined as follows.

```
public interface ILambda {
    /**
     * Takes an Object x as input, performs some computation
     * and returns an Object as output.
     */
    public Object apply(Object x);
}
```

For instance, the following are two concrete lambda functions:

```
public class Lambda1 implements ILambda {
    public Object apply(Object x) {
        return "Lambda1.apply with the input:" + x.toString());
    }

public class Lambda2 implements ILambda {
    public Object apply(Object x) {
        return "Lambda2.apply with the input:" + x.toString());
    }
```

Consider the following interface, ***`ILogical`***, that represents objects with the ability to apply one of two possible lambda's:

```
public interface ILogical {
    /**
    * Calls either the trueLambda's or the falseLambda's apply method
    * with the given x, and returns the result.
    */
    public Object select(ILambda trueLambda, ILambda falseLambda, Object x);
}
```

Continued on next page ➔

Define two concrete classes, called `True` and `False`, that implement ***`ILogical`*** and satisfy the following specification.

When the following is executed with a variable "`b`" of type ***`ILogical`***,

```
b.select(new Lambda1(), new Lambda2(), "host")
```

the return value will be

"`Lambda1.apply with the input: host`" when `b` is of type `True` and
"`Lambda2.apply with the input: host`" when the `b` is of type `False`.

Your code for `True` and `False` may **not** contain any `String` literals.

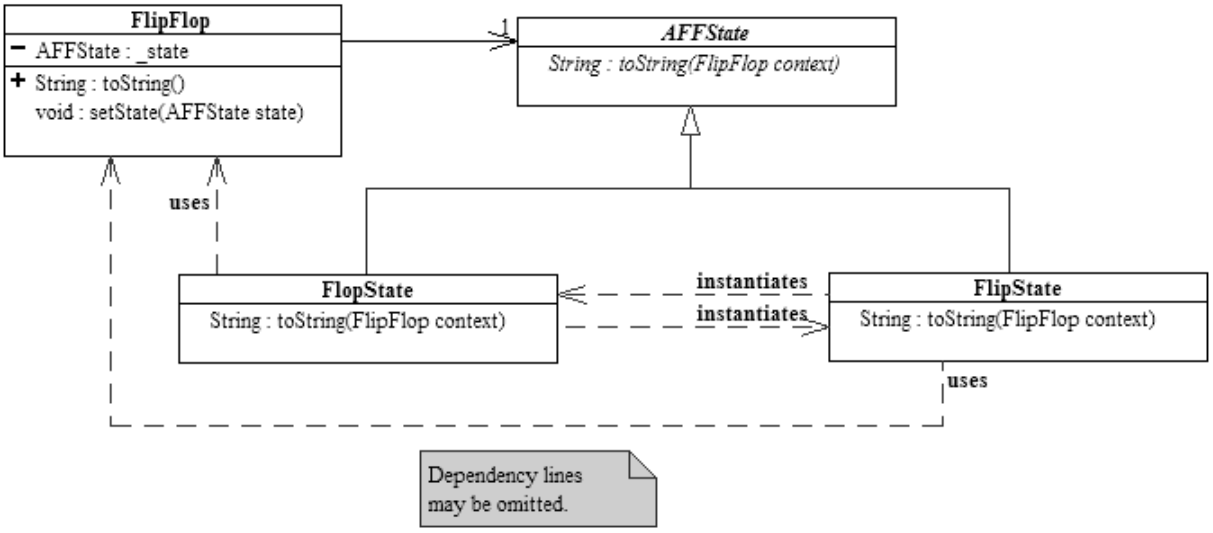
Aside: The behavior `True` and `False` corresponds to the way that conditional decision-making is performed in a pure "message-passing" OO language such as Smalltalk. In general, this technique is known as "double-dispatching" because the result is the effect of two consecutive method calls.

```
public class True implements ILogical {
    public Object select(ILambda trueLambda, ILambda falseLambda, Object x) {
        return trueLambda.apply(x);
    }
}

public class False implements ILogical {
    public Object select(ILambda trueLambda, ILambda falseLambda, Object x) {
        return falseLambda.apply(x);
    }
}
```

4. (20 pts total) Consider the following system of classes:

```
//---------------------------------------------------------------------------
package flipflop;

public class FlipFlop {
  private AFFState _state = new FlipState();

  public String toString() {
    return _state.toString(this);
  }

  void setState(AFFState state) {
    _state = state;
  }
}

//---------------------------------------------------------------------------
package flipflop;

abstract class AFFState {
  abstract String toString(FlipFlop context);
}

//---------------------------------------------------------------------------
package flipflop;

class FlipState extends AFFState{
  String toString(FlipFlop context) {
    context.setState(new FlopState());
    return "Flip";
  }
}

//---------------------------------------------------------------------------
package flipflop;

class FlopState extends AFFState{
  String toString(FlipFlop context) {
    context.setState(new FlipState());
    return "Flop";
  }
}

//---------------------------------------------------------------------------
```

a) (10 pts) Draw a complete UML class diagram of the above system, including all fields and methods for every class shown. Be sure to show the proper access specifiers.



b) (10 p[s) What output will the following code produce? Why? Be specific and complete!

Operations executed in the following order:	**Output and reasons for that output:**
`flipflop.FlipFlop ff = new flipflop.FlipFlop();`	No output to screen. New `FlipFlop` instance being created.
`System.out.println(ff.toString());`	Output: "`Flip`" `FlipFlop`'s `toString()` method always delegates to its `_state`'s `toString()` method. `_state` is initialized to be a `FlipState`, so it's `toString()` method returns "`Flip`". But `FlipState`'s `toString()` method also sets its host `FlipFlop`'s state to be a new `FlopState`.
`System.out.println(ff.toString());`	Output: "`Flop`" `FlipFlop`'s `toString()` method always delegates to its `_state`'s `toString()` method. `_state` is currently set to be a `FlopState`, so it's `toString()` method returns "`Flop`". But `FlopState`'s `toString()` method also sets its host `FlipFlop`'s state to be a new `FlipState`.
`System.out.println(ff.toString());`	Output: "`Flip`" `FlipFlop`'s `toString()` method always delegates to its `_state`'s `toString()` method. `_state` is currently set to be a `FlipState`, so it's `toString()` method returns "`Flip`". But `FlipState`'s `toString()` method also sets its host `FlipFlop`'s state to be a new `FlopState`.
`System.out.println(ff.toString());`	Output: "`Flop`" `FlipFlop`'s `toString()` method always delegates to its `_state`'s `toString()` method. `_state` is currently set to be a `FlopState`, so it's `toString()` method returns "`Flop`". But `FlopState`'s `toString()` method also sets its host `FlipFlop`'s state to be a new `FlipState`.

5. (20 pts) Consider the familiar composite design pattern for the immutable list structure. Assume the ***IList*** contains `Integer` objects. Write the code for `MTList` and `NEList` to compute the sum of the squares of the integers in the list as indicated below.

```
/**
 * Represents the abstract list structure.
 */
public interface IList {
    /**
     * Computes the sum of the squares of the integers in this IList,
     * assuming it contains Integer objects only.
     */
    Integer sumOfSquares();
}

/**
 * Represents the empty list.
 */
public class MTList implements IList {
    public static final MTList Singleton = new MTList();
    private MTList() {
    }

   /**
    *   Student to write appropriate method to compute the sum of
    *   the squares of integers in this IList.
    */

     public Integer sumOfSquares() {
         return new Integer(0);
     }




}

/**
 * Represents non-empty lists.
 */
public class NEList implements IList {
   private Object _first;
   private IList _rest;

   public NEList(Object f, IList r) {
       _first = f;
       _rest = r;
   }
   /**
    *   Student to write appropriate method to compute the sum of
    *   the squares of integers in this IList.
    */

     public Integer sumOfSquares() {
         return new Integer(
             ((Integer)_first).intValue() * ((Integer)_first).intValue()
             + ((Integer)_rest.sumOfSquares()).intValue() );
     }



}
```