McGill University
COMP 303 – Programming Techniques
Midterm Examination – Fall 2006

Name (IN INK): _____

Student Number (IN INK): _____

Signature (IN INK): _____

You have **80 minutes** to write this examination.

Do not start until you are informed to.

Do not continue after you are informed to stop.

If you wish to have your midterm exam returned to you, write in blue or black **ink only**. If you write your exam in pencil you will only be able to view your copy during office hours.

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.

You may **not ask any questions** of the invigilators except in the case where you believe that there is an error in a question. If you believe the requirements stated in a question are ambiguous, you are required to state the ambiguity, state the assumption that you are going to make and then proceed to answer the question.

Good luck!

Question	Mark
1	/16
2	/30
3	/16
4	/18
5	/20
Total (/100)	

Academic Integrity

McGill University values academic integrity. Therefore all students must understand the meaning and consequences of cheating, plagiarism and other academic offenses under the Code of Student Conduct and Disciplinary Procedures.

Question 1: [16 marks]

A programmer in desperate need of retraining was asked to produce a building management application and came up with the following mess.

```
public class Floor {
   private List<Office> aOffices = new ArrayList<Office>();
   public String aName;
   public List<Office> getOffices() { return aOffices; }
   public Floor( String pName ) { aName = pName; }
   public String toString() { return aName; }
}

public class Office {
   private String aOccupant;
   public String getOccupant() { return aOccupant; }
   public Office( String pOccupant ) { aOccupant = pOccupant; }
   public Floor addOrRemoveToFromFloor( Floor pFloor, boolean pAdd )
   {
      if( pAdd ) pFloor.getOffices().add( this );
      else pFloor.getOffices().remove( this );
      return pFloor;
   }
}

public class Building {
   private List<Floor> aFloors = new ArrayList<Floor>();
   public List<Floor> getFloors() { return aFloors; }
}


public class Application {
   public static void mcgill()
   {
      Building lMcConnell = new Building();
      lMcConnell.getFloors().add( new Floor("2"));
      Floor lFloor = new Floor("1");
      lFloor.getOffices().add( new Office("Martin"));
      new Office( "Claude" ).addOrRemoveToFromFloor( lFloor, true );
      lMcConnell.getFloors().add( new Floor("1"));
      int lFloorNb = 1;
      System.out.println( lMcConnell.getFloors().
         get(lFloorNb-1).getOffices().get(0).getOccupant() +
            "has and office on floor " + lMcConnell.getFloors().
               get(lFloorNb-1).aName);
   }
}
```

Comment on the quality of the *encapsulation* and *interfaces* offered by the `Building`, `Floor`, and `Office` classes. Your comments must be organized in terms of main points, which should be discussed by referring to the code. Use specific terminology as seen in class. Discuss at least **four** main points. Do not discuss the textual format of the code, or the `Application` class, which is provided only to illustrate how the other classes can be abused. Note that this code compiles.

Question 2: [30 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 `Objects`. It should be possible to update the model by updating individual elements. The model should be extensible through subclassing so that additional operations can be added (e.g., making some calculations prior to an update).
3. Every change to the state of the model *must* trigger a notification of the observers. This should be true for both the basic model *and* any extension. Use the TEMPLATE METHOD design pattern to ensure that this requirement will be respected by all specializations of the 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)
[13 marks]

b) Briefly explain how your design meets the requirements. [3 marks]

c) Does your realization of the Observer implement the push or the pull model. Why? [4 marks]

d) Write the code of the class in a) that encapsulates the implementation of the Subject. [8 marks]

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

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

Question 3: [16 marks]

The following method returns a number that forms a special code used for some performance optimization purpose. The code below compiles, and actually implements the desired behavior. Unfortunately, the interface of `calculate` is problematic.

```
public class SpecialCode
{
   private Integer aCode;

   public SpecialCode( Integer pCode )
   { aCode = pCode; }

   public double calculate( Integer pVal1, Integer pVal2, Integer pVal3 )
   {
      if( pVal1 == pVal2 ) return 0;
      if( pVal1 == null || pVal2 == null )
            return aCode / pVal3;
      else
            return aCode / (pVal1 - pVal2);
   }
```

a) What would be good preconditions for this method? List all appropriate precondition in the form of Java `assert` statements that you would insert before the normal method code [4 marks].

b) Let's say we change the interface specification for `calculate`. Now `null` arguments should no longer be accepted. Instead, a checked exception of type `NullArgException` should be raised when null arguments are passed. Complete the code below to implement this new policy. You do not need to copy the asserts written in a). [3 marks]

```
public double calculate( Integer pVal1, Integer pVal2, Integer pVal3 )


{




}
```

c) Write a test case that will exercise all the branches and the exceptional behavior of your second version of `calculate` (the one you wrote in b). Only test for valid input (i.e., for input respecting the preconditions). [9 marks]

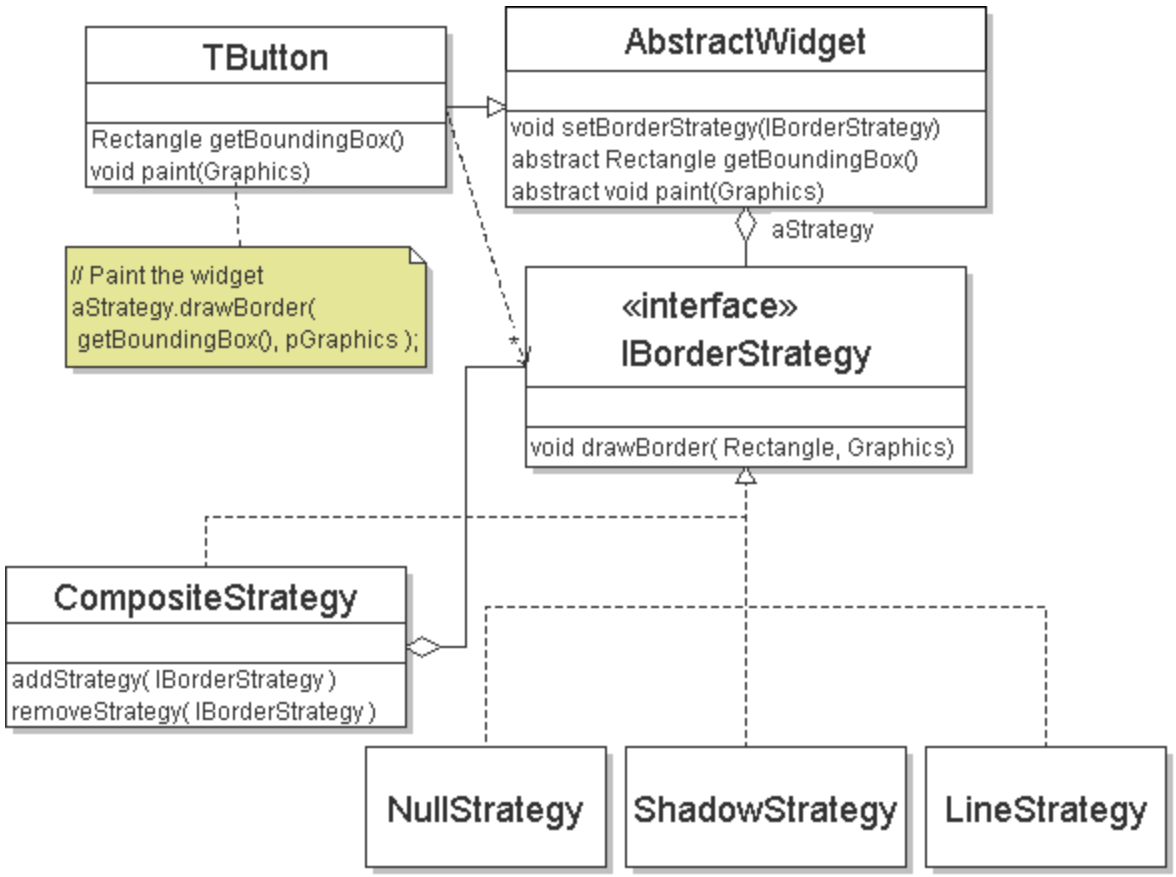
Remember that in subclasses of `TestCase` you have access to the following methods:

```
assertEquals( T pExpected, T pValue ) // T is any primitive or reference type
assertTrue( boolean pPredicate )
fail()
```

```
public class TestSpecialCode extends TestCase
{
   public void testCalculate()
   {









   }
}
```

Question 4 [18 marks]

Tremblay and Sons Software Solutions Inc. received a contract for building a GUI widget toolkit, the Tremblay Widget Innovation Toolkit (TWIT). Part of the design of a TWIT widget is illustrated below.



The software developers at Tremblay's are design pattern enthusiasts. A TWIT widget (such as a `TButton`) can be rendered with different types of borders (e.g., no border, line border, shadow border). To give a border to a widget, the client code creates the appropriate STRATEGY and sets the strategy in the widget. The strategy is then used by the paint method to draw the border. The strategy objects are decoupled from widget since the strategy method only takes in a bounding box argument (of type `Rectangle`) and a `Graphics` (on which drawing primitives are called). Finally, the COMPOSITE design pattern is used to allow client to create multiple effects (e.g., a line with a shadow).

a) Describe two weaknesses of this design. Illustrate your points with Java client code. [8 marks]

b) Redo the design so that it supports the same requirement (flexible composition of borders), but this time using the DECORATOR design pattern. Draw the UML class diagram. [10 marks]

Question 5: Very short answers [3 marks each, up to a maximum of 20]

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

(a) What is represented by a typical sequence diagram? Be as precise as possible (i.e., answering "a sequence" won't cut it).

(b) Is it a good idea for a getter to return a field reference to a mutable object? Why or why not?

(c) What problem does the COMPOSITE design pattern solve?

(d) What's the Law of Demeter?

(e) What is the difference between a subtype and a subclass?

(f) What is the basic idea of the Liskov Substitutability Principle?

(g) Why does it make no sense to have both the static and final modifiers in front of a Java method?