# Midterm #1

CMSC 433
Programming Language Technologies and Paradigms
Fall 2006

October 26, 2006

## Guidelines

This exam has 8 pages (including this one); make sure you have them all. Put your name on each page before starting the exam. Write your answers directly on the exam sheets, using the back of the page as necessary. Bring your exam to the front when you are finished. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

*You may avail yourself of the punt rule.* If you write down *punt* for any part of a question, you will earn 1/5 of the points for that question (rounded to nearest integer).

*Use good test-taking strategy*: read through the whole exam first, and first answer the questions that are easiest for you and are worth the most points.

| Question | Points | Score |
|---|---|---|
| 1 | 24 | |
| 2 | 19 | |
| 3 | 19 | |
| 4 | 19 | |
| 5 | 19 | |
| Total | 100 | |

1. (24 pt, Short answer)

   (a) What is the information hiding principle?

   (b) Why use information hiding?

   (c) Say you have a class whose data members are all private. Is this enough to ensure that the design decision hidden in this module remains hidden? If it is, explain why; if not, write a few lines of Java showing how information hiding can still be violated.

   (d) Say you have some base classes that are likely to change and want to avoid the fragile base class problem. What are two alternatives to inheritance that you can use? (One word each)

(e) What is the principle of low coupling and how does it facilitate ease of change?

(f) Good or bad pair programming practice? Discuss. While the driver is implementing a class, the navigator is writing down test cases for the class.

2. (19 pts) List the line numbers of all violations of the Law of Demeter in the following code.

```
 1  public class MainFrame extends JFrame
 2  {
 3    private static final String appName    = "MainFrame" ;
 4    private static final String appVer    = "0.6" ;
 5    private Color        bgColor    = Color.black ;
 6    private boolean        fullScreen  = false ;
 7    private Dimension      screenSize ;
 8
 9    /* Make a new Frame with an ImageCollage as the content pane */
10    MainFrame(ImageCollage collage)
11    {
12      // Set the initial window size, title, background color, close on exit
13      setTitle(appName) ;
14      setDefaultCloseOperation(EXIT_ON_CLOSE) ;
15      setBackground(bgColor) ;
16      Toolkit toolkit = Toolkit.getDefaultToolkit() ;
17      screenSize = toolkit.getScreenSize() ;        // Get (primary) screen resolution
18      setSize(screenSize) ;
19      toolkit.setDynamicLayout(true) ;          // Tell Swing to update the window as it is being resized
20      Image icon = toolkit.getImage("icon.gif") ;
21      setIconImage(icon) ;
22      collage.setCanvasSize(screenSize) ;
23      collage.setBackgroundColor(bgColor) ;
24      collage.clearCanvas() ;
25      setContentPane(collage) ;
26      boolean fullScreen = GraphicsEnvironment.getLocalGraphicsEnvironment()
27                               .getDefaultScreenDevice().isFullScreenSupported() ;
28      Debug.d("Screen size " + (int) screenSize.getWidth() + "x" + (int) screenSize.getHeight()
29        + ", full screen " + (fullScreen ? "" : "not ") + "supported.") ;
30    }
31   }
```

3. (19 pt) Design pattern

   (a) You are writing a diagnostic utility to test the various components of a computer system to ensure that they will work with your big, expensive enterprise application. You want to write one class for each computer component you diagnose: CPUTest, MemoryTest, and DiskTest. However, your utility needs to be cross-platform, and each class contains some platform-specific code. So for each type of test, you will have several implementations, one for each platform: PCMemoryTest, MacMemoryTest, LinuxMemoryTest, etc. You'd like your diagnostic utility to create and use the right test classes for the platform its running on, without having to change a lot of code each time you add a new platform. You recall that there is a design pattern to solve this problem. What design pattern would you use?

   (b) Draw a UML class diagram showing the relationships among participants in your design pattern. Include the three types of tests above, for two platforms: PC and Mac.

4. (19 pt) Visitor / Double dispatch

   (a) You have an interface A, implemented by classes B and C and an interface X, implemented by classes Y and Z.

   Define a method Test.doubleDispatch

```
class Test {
        static void doubleDispatch(A a, X x) { ... }
        }
```

   Use the visitor or double dispatch design pattern to use dynamic dispatch so that a method chosen to be executed based on the runtime type of `a` and `x`. In other words, there should be one method that is selected if `a` is a `B` and `x` is a `Y`, another method selected if `a` is a `B` and `x` is a `Z`, and so on. You may not use `instanceof` or any conditional expressions; instead, use the visitor pattern or double dispatch to achieve this. You may add any methods you need to add to the interfaces and classes. Label the methods that select for each combination of runtime types (e.g., label one method as the `BY` method). Use space on the following page if needed.

(b) Given the implementation you chose, what would be easier: adding an additional class that implemented `A`, or an additional class that implemented `X` (in either case, the code would now need to select one of six different methods, depending upon the runtime type of `a` and `x`).

5. (19 pts) Iterator pattern

   The `3n+1` problem is a simple but open problem in computer science. [1] Given the following method, a sample call to threeNPlusOne(22) will print the sequence 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1.

```
void threeNPlusOne(int n) {
   while (true) {
     System.out.println(n);
     if (n == 1) return;
     if (n%2 == 0) n = n/2;
     else n = 3*n+1;
   }
}
```

   Your job is to define a method:

```
Iterator<Integer> threeNPlusOneIterator(int n) {...}
```

   that returns an iterator over integer values. Thus,

```
for(int i : threeNPlusOneIterator(22)) System.out.println(i)
```

   will print the same sequence as the code above.

   Note that due to autoboxing, you can pretty much ignore the difference between `int` and `Integer` for this project. Also, the Iterator interface is declared as:

```
interface Iterator<T> {
  boolean hasNext();
  T next();
  void remove();
  }
```

[1] It is believed that this sequence will terminate for any positive integer n, but that conjecture is unproven (fame if not fortune to the first person who proves it). It is known to terminate for all positive integer values that can be represented with a `int`. To win the fame, you need to prove that it terminates for all positive integer values.