

Massachusetts Institute of Technology  
6.170 Laboratory in Software Engineering  
Fall 2003  
Quiz 2  
Tuesday, 28 October 2003

Name: SOLUTIONS \_\_\_\_\_

Athena User Name: \_\_\_\_\_

TA (circle one):

- |                        |               |                  |
|------------------------|---------------|------------------|
| 1. Matthew Notowidigdo | 2. Lee Lin    | 3. David Saff    |
| 4. Brian Dunagan       | 5. Thomer Gil | 6. Sameer Ajmani |
| 7. Roger Moh           |               |                  |

Instructions

This quiz is 50 minutes long. It contains 30 questions in 13 pages (including this page). Please check your copy to make sure that it is complete before you start. Turn in all pages, together, when you finish. Write your name and recitation section number on the top of every page. Please write neatly. No credits will be given if we cannot read what you write.

Remember to email your TA with project assignments by 5pm today.

Question	Total Score
1 – 14	
15 – 18	
19	
20 – 22	
23 – 24	
25 – 26	
27	
28 – 30	
Total	

Section: \_\_\_\_\_ Name: \_\_\_\_\_

### **True/False [28 points]**

Circle the correct answer:

1. **T / F**  
Private variables can be directly accessed from subclasses.
2. **T / F**  
Fitt's Law says larger targets are easier to use, so controls should be placed at the edge of the screen.
3. **T / F**  
According the Model Human Processor, the Visual Image Store decays faster than the Auditory Image Store.
4. **T / F**  
The equals method applied to a class from the Java Collections Framework tests observational, not behavioral, equivalence.
5. **T / F**  
An implementation of hashCode() that always returns 1 can lead to performance degradation in some programs.
6. **T / F**  
One advantage of a factory method over a constructor is that the factory method can return a subclass of the promised return type while a constructor must always provide the declared type.
7. **T / F**  
One can use the Strategy design pattern to encapsulate an algorithm as an object.

Section: \_\_\_\_\_ Name: \_\_\_\_\_

8. **T/F**  
One disadvantage of using the Observer pattern is that the number of observers for a subject must be decided on during design time.

```
Set s1 = new HashSet();  
  
Set s2 = s1;  
  
s1.add("Zeeb");  
s2.add("ZEEB");
```

9. **T/F**  
s1.equals(s2)

10. **T/F**  
s1.equals(null)

For each of the following questions, suppose the given line of code was inserted at the BEGINNING of a valid equals method. Could this cause the method to have different behavior or violate the Java equals contract? If so, then circle "T"; else circle "F". Assume that the hashCode() method is implemented correctly.

11. **T/F**  
if (this == obj) return true;

12. **T/F**  
if (this != obj) return false;

13. **T/F**  
if (this.hashCode() == obj.hashCode())  
return true;

14. **T/F**  
if (this.hashCode() != obj.hashCode())  
return false;

## Equality [18 points]

For each of the following implementations for `ModernPerson.equals`, say whether it is reflexive, symmetric, and/or transitive. Write “none” if the implementation is none of the above.

```
class Person {
    private String name;
    /** @effects returns true iff that is a Person
        whose name equals this.name */
    public boolean equals(Object that);
}

class ModernPerson extends Person {
    String email;
    public boolean equals(Object that) {
        //SEE CODE BELOW!
    }
}
```

15. [3 points]

```
return (((ModernPerson)that).email.equals(this.email));
```

*reflexive*

16. [3 points]

```
return (((ModernPerson)that).email.equals(this.email) &&
        ((ModernPerson)that).name.equals(this.name));
```

*reflexive, transitive*

17. [3 points]

```
return (((ModernPerson)that).name.equals(this.name));
```

*reflexive, transitive*

18. [3 points] `return (false);`

*transitive*

Section: \_\_\_\_\_ Name: \_\_\_\_\_

19. **[6 points]** Ben is inspired by the 6.170 lecture on “The Equals Method” and seeks to solve the problem of breaking symmetry and transitivity in `equals()` when comparing a `Point` and `ColorPoint`.

Ben decides to keep `ColorPoint.equals()` exactly the same by requiring that the `x`, `y`, and `color` fields to be equal in order for two `ColorPoints` to be equal. He makes the following modification to the `Point.equals()` method:

```
public boolean equals (Object obj) {  
    if (! (obj instanceof Point)) return false;  
    Point pt = (Point) obj;  
    return pt.x == x && pt.y == y && pt.equals(this);  
}
```

Ben claims the additional clause `pt.equals(this)` solves the symmetry and transitivity problems. Why is Ben incorrect? Explain using no more than forty words.

*infinite loop*

Section: \_\_\_\_\_ Name: \_\_\_\_\_

## Testing [6 points]

Consider the code fragment below.

```
boolean a, b;

if (a || b) {
    Statement_1;
    if (a && b)
        Statement_2;
    else
        Statement_3;
}
else {
    Statement_4;
}
```

- A test corresponds to a single combination of true or false values for a and b.
20. **T / F**  
100% (complete) statement coverage can be obtained for the above code fragment using all four possible tests.
21. **T / F**  
100% (complete) branch coverage can be obtained for the above code fragment using 2 tests.
22. **T / F**  
100% (complete) path coverage can be obtained for the above code fragment using 3 tests.

Section: \_\_\_\_\_ Name: \_\_\_\_\_

## Subtyping [20 points]

Consider the classes below.

```
// Coding is a true subtype of project
class Coding extends Project {...}

//HourlyWages is a true subtype of Wages.
class HourlyWages extends Wages {...}

class ProjectPartner {
    // requires: hours <= 25
    // effects: complete hours amount of work and
               throws SlippageException if the work
               can't be completed ontime
    Project work(int hours, Wages w) { ... }
}

class Slacker {
    // requires: good night's sleep
    // effects: complete hours amount of work and
               throws SlippageException if the work
               can't be completed ontime
    Project work(int hours, Wages w) { ... }
}

class Workhorse {
    // requires: hours <= 27
    // effects: complete hours amount of work and
               throws SickException when falling sick
    Coding work(int hours, Wages w) { ... }
}

class Hacker {
    // requires: hours <= 30
    // effects: complete hours amount of work
    // throws: none
    Coding work(int hours, HourlyWages w) { ... }
}

class Exterminator {
    // requires: none
    // effects: complete 20 <= hours <= 30
               amount of work depending on mood
    int mood; // modifiable level of mood
    Project work(int hours, Wages w) { ... }
}
```

Section: \_\_\_\_\_ Name: \_\_\_\_\_

23. **[12 points]** For each of the following classes, determine whether the class can be a true subtype of `ProjectPartner` according to the substitution principle and give a **one sentence** explanation to support your answer.

a) `Slacker`

*No, the requires clauses are incomparable.*

b) `Workhorse`

*No, the exceptions are different.*

c) `Hacker`

*No, `HourlyWages` is a subtype.*

d) `Exterminator`

*No, the effects clauses are different.*

24. **[8 points]** Which classes can be a Java subclass of `ProjectPartner`? State any assumptions you are making.

a) `Slacker`

*Yes, `work()` is simply overridden*

b) `Workhorse`

*No, compiler will not let return type for `work()` change*

c) `Hacker`

*Yes, `work` is overloaded, not overridden, so return type can change*

d) `Exterminator`

*Yes, `work()` is overridden*

## Inheritance [8 points]

Sometimes even designers of popular languages make mistakes. Consider the following classes in the Java Collections API:

```
/**
 * This class implements a hashtable, which maps keys to
 * values. Null objects cannot be used as a key or as a
 * value... **/
class Hashtable {
    ...
    public Object get(Object key) {... }
    public Object put(Object key, Object value) {...}
}

/**
 * The Properties class represents a persistent set of
 * properties. The Properties can be saved to a stream or
 * loaded from a stream. Each key and its corresponding
 * value in the property list is a string... **/
class Properties extends Hashtable {
    . . .
    public String getProperty(String key) {...}
    public Object setProperty(String key,
                               String value) {...}
}
```

25. [4 points] How can a client violate the representation invariant of `Properties`? Explain using no more than forty words.

*Use a non-string and do a `put()`.*

26. [4 points] How could the designer of the `Properties` class have avoided this problem? Explain using no more than forty words.

*Use composition.*

Section: \_\_\_\_\_ Name: \_\_\_\_\_

## Design Patterns [20 points]

27. [9 points] Recall that a Singleton design pattern is a creational pattern that ensures at most one instance of a class can ever exist. Fill in the blanks in the code below to implement the Singleton design pattern.

```
/** Universe follows the Singleton design pattern */
class Universe {
    private int numStars;

    _____private static_____ Universe singletonInstance;

    /** Constructs a new Universe */

    _____private_____ Universe() {
        numStars = 1000000;
    }

    /** Returns an object of type Universe */

    _____public static Universe_____ getInstance() {

        if (singletonInstance == null) {
            singletonInstance = new Universe();
        }
        return singletonInstance;
    }
}
```

Section: \_\_\_\_\_ Name: \_\_\_\_\_

Ben Bitdiddle is implementing his MapQuick GUI for PS6. He is an ambitious student and has decided to extend his GUI to include a visual drawing of the shortest path and surrounding streets. Originally, Ben used a direct approach that simply forwarded the result of the shortest path finder to the text area to display. Now Ben modifies his class to also forward the result to the visual display module:

```
class MapQuickGUI{
    ...

    public void displayShortestPath(Path p) {
        showPathText(p);
        drawShortestPathOnScreen(p);
    }
}
```

Alyssa P. Hacker suggests that Ben instead write his code like this:

```
interface PathListener {
    public displayPath(Path p);
}

class TextAreaListener implements PathListener {
    public displayPath(Path p) {
        //show path text on screen
    }
}

class GraphicalDisplayListener implements PathListener
{
    public displayPath(Path p) {
        //draw Path on screen
    }
}

class MapQuickGUI {
    List listeners = new ArrayList();
    ...
    public void addPathListener(PathListener p) {
        listeners.add(p);
    }
    public void displayShortestPath(Path p) {
        for (int i=0; i<listeners.size(); i++)
            ((PathListener)listeners.get(i)).displayPath(p);
    }
}
```

Section: \_\_\_\_\_ Name: \_\_\_\_\_

28. **[3 points]** What pattern has Alyssa applied?

*Observer pattern.*

29. **[4 points]** What are the advantages of Alyssa's code? Limit your answer to forty words.

*Different displays can be easily added in the future.*

30. **[4 points]** What are disadvantages with Alyssa's code? Limit your answer to forty words.

*It adds code complexity and reduces scalability.*

Section: \_\_\_\_\_ Name: \_\_\_\_\_

- END OF QUIZ -