

Name:

TA's name:

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
6.170 LABORATORY IN SOFTWARE ENGINEERING
FALL 2005
Quiz
November 1, 2005

This is a CLOSED-BOOK quiz.

Before you start, write your name and your TA's name at the top of every sheet.

There are 6 questions (labeled A through F), each with several numbered parts. Please check your copy of the quiz before you start to make sure it is complete: you should have 13 pages on 7 sheets.

You have 110 minutes and should attempt to answer all questions. The table below shows their relative value. Note that, even though sometimes more space is given (for pagination purposes), every question can be answered in a few sentences at most.

<i>Part</i>	<i>Score</i>	<i>Max possible</i>
A		30
B		10
C		20
D		15
E		15
F		10
<i>Total</i>		100

Name:

TA's name:

A Shorties

True or false? Please answer by writing true or false to the left of the question number.

If a class C' is declared in Java to extend class C , and compiles without errors, then

1. C' is a subclass of C
2. C' is a behavioral subtype of C
3. Unless overridden, every method of C is implicitly a method of C'
4. Any code expecting an object of type C will behave as expected when passed an object of type C'
5. If C has a method with name m and one argument of type T , then if a method named m appears in C' , it must have an argument of the same type T
6. If C has a method with name m and one argument of type T , then if a method named m appears in C' , it must have an argument whose type is T , or a supertype of T
7. If C has a method with name m and one argument of type T , then if a method named m appears in C' with the same argument type, the return types of the two methods must be the same
8. If C has a method with name m and one argument of type T , then if a method named m appears in C' with the same argument type, the return type of the method in C' must be the same as the return type of the method in C , or a subtype of it

To ensure that a program does not suffer from problems related to subtyping, it is sufficient (but perhaps not necessary) to

9. Make sure the code compiles without errors
10. Mark every class in the program as final
11. Check that every class is a true (behavioral) subtype of its superclass and the interfaces it implements

The representation invariant of an abstract data type

12. Is a predicate over a single instance of a representation object, and cannot capture sharing properties involving multiple representation objects
13. Should always be executed as a runtime assertion
14. Should hold at the start and end of all public methods of the type
15. Should hold at the start and end of all methods of the type
16. Should be efficiently computable
17. Allows methods to be reasoned about independently
18. Is useful only when an abstraction function is also recorded

Name:

TA's name:

19. Cannot be non-deterministic
20. Can never be the constant predicate *false*
21. Is called an 'invariant' because its value must not change during the execution of a method
22. Applies only to immutable datatypes

The precondition of a method

23. Must be a predicate that can be efficiently computed
24. Should always be checked explicitly by the client of the method
25. Should always be checked explicitly by the method itself
26. Should never be violated by a client
27. Should never be the constant predicate *true*
28. Should never be the constant predicate *false*
29. Should generally be weak, if the method is to be easily used

A hashCode method

30. Will result in inefficient code if it always returns the same value
31. Must be coded carefully to satisfy the Object contract
32. Must always be provided explicitly, in preference to the inherited method from Object
33. Should never be a mutator
34. Should be deterministic

Cognitive delay

35. Between sensing and muscle response is typically about 240ms
36. In perceptual processing is typically 10ms
37. Provides the parameters for Fitt's Law, which can guide the design of pointing tasks
38. If shorter, would suggest greater use of progress bars and delay indications

Java exceptions

39. Are objects themselves
40. Are lightweight and should be used extensively
41. Should always be used in preference to boolean return values
42. Should be used to signal any violation of a precondition

Name:

TA's name:

- 43. Can violate 'failure atomicity', if thrown too late
- 44. Can violate 'representation independence', if thrown too late
- 45. May be propagated without an explicit *throw*

The object model of a program

- 46. Expresses an invariant over program states
- 47. Can be derived trivially from the code
- 48. Only shows abstract data types
- 49. Is a subgraph of the program's module dependence diagram
- 50. Is strictly less expressive than a collection of rep invariants
- 51. Does not show methods
- 52. Is invalid when subclasses are not subtypes
- 53. Is a graph that is always acyclic

Runtime assertions

- 54. Should never have visible side effects
- 55. Are useful in testing
- 56. Should always be turned off when the code is deployed

Testing

- 57. Requires the creation of stubs when a program is constructed bottom-up
- 58. Should be automated to the greatest extent possible
- 59. Is harder to do for non-deterministic code
- 60. Can be undermined by rep exposure

Name:

TA's name:

B Substitutability

A browser's cache holds a collection of web pages indexed on their URL's. Suppose the cache is implemented as a Java class, with a method `add` that takes a URL and a page, and attempts to add them to the cache. The client of the class is expected to call another method `flush` to free up space if needed, so a call to the `add` method might fail to make the addition if there is inadequate room.

Assume the cache has a maximum size `MAXSIZE`, and that `size(c)` gives the current space consumption of a cache `c`, and `size(p)` gives the space required for storing a page `p`.

Consider the following variant specifications of the `add` method:

```
1  void add1 (URL u, Page p)
2  modifies this
3  effects adds p to this cache under the index u
4
5  void add2 (URL u, Page p)
6  modifies this
7  requires size(this) + size(p) < MAXSIZE
8  effects adds p to this cache under the index u
9
10 void add3 (URL u, Page p)
11 modifies this
12 effects
13   if size(this) + size(p) < MAXSIZE
14     adds p to this cache under the index u
15   else
16     does nothing
17
18 void add4 (URL u, Page p)
19 modifies this
20 effects
21   if size(this) + size(p) < MAXSIZE
22     adds p to this cache under the index u
23   else
24     throws IllegalArgumentException
```

If a client is expecting a method satisfying one of these specifications, S say, it may be reasonable to provide instead with a method satisfying a different specification, S' say. In these circumstances, we would say that S' refines S .

1. Draw a graph to the right of the specs showing the refinement relation for these 4 specifications, with an arrow from S' to S when S' refines S . Please lay the graph out so that the most weakest specifications are at the top, and the specifications are lexically ordered from left to right.

Name:

TA's name:

C Abstraction Functions and Representation Invariants

Here is a (possibly defective) fragment of an implementation of an abstract data type for a set, with a method `choose` that removes an arbitrary element from the set and returns it, and a method `add` that adds an element to the set:

```
25 class Set {
26   private Object [] elements;
27   private int size;
28
29   Set (int capacity) {
30     elements = new Object [capacity];
31     size = 0;
32   }
33
34   Object choose () {
35     if (size == 0) return null;
36     size = size - 1;
37     return elements [size];
38   }
39
40   void add (Object o) {
41     this.elements [size] = o;
42     size = size + 1;
43   }
44   ...
45 }
```

Consider the following candidate representation invariants, and for each, say whether it is preserved by `choose`, `add`, both or neither:

1. $\text{size} \geq 0$
2. $\text{size} \leq \text{elements.length}$
3. $\forall i: \text{int} \mid i \geq 0 \wedge i < \text{size} \Rightarrow \text{elements}[i] \neq \text{null}$
4. $\forall i, j: \text{int} \mid i < j \wedge i \geq 0 \wedge j < \text{size} \Rightarrow \text{elements}[i].\text{compareTo}(\text{elements}[j]) < 0$
5. $\forall i: \text{int} \mid i \geq \text{size} \wedge i < \text{elements.length} \Rightarrow \text{elements}[i] == \text{null}$

Name:

TA's name:

Answer the following questions briefly and precisely:

6. The last representation invariant might not appear at first to be necessary. Explain why it's a reasonable invariant to include.

7. If the two statements in the add method were exchanged, the implementation would be faulty, even if the array index were changed to $\text{size}-1$. Explain why.

8. Give a plausible abstraction function for this representation.

D Datatype Theory

Here is a (defective) fragment of an implementation of an abstract data type for a map, with a method `put` to associate a value with a key, and a method `getEntry` that obtains an entry whose key matches the key passed as an argument. The value associated with the key can then be updated easily by modifying the entry object; this method can be used internally (as in `put`) or by a client.

```

46 public class ArrayMap <K,V> {
47     private List <Entry<K,V>> entries = new LinkedList <Entry<K,V>>();
48     ...
49     public void put (K k, V v) {
50         Entry<K,V> e = getEntry (k);
51         if (e != null)
52             e.val = v;
53         else
54             entries.add (new Entry<K,V> (k, v));
55     }
56
57     public Entry <K,V> getEntry (K k) {
58         for (Entry<K,V> e: entries) {
59             if (e.key.equals (k)) return e;
60         };
61         return null;
62     }
63 }
64
65 public class Entry <K,V> {
66     Entry(K k, V v) { key = k; val = v; }
67     K key;
68     V val;
69 }

```

1. What representation invariant does `getEntry` assume?
2. Returning `null` in `getEntry` is controversial. Give one reason for, and one reason against, this decision.

Name:

TA's name:

E Equality

Suppose you're implementing a program that looks for matches between images, and that each pixel value is represented by an object of the type Pixel:

```
70 final class Pixel {
71   private byte red;
72   private byte green;
73   private byte blue;
74
75   Pixel (byte r, byte g, byte b) {
76     red = r; green = g; blue = b;
77   }
78
79   byte getRed () {return red;}
80   byte getGreen () {return green;}
81   byte getBlue () {return blue;}
82
83   boolean equals (Pixel p) {
84     return (approx (p.red, red) && approx (p.green, green) && approx (p.blue, blue));
85   }
86   private boolean approx (byte x, byte y) {
87     return (Math.abs (x - y) < 10);
88   }
89 }
```

The equal method has been designed to return true when two pixel values are close to each other, so that a small color shift will not make two images distinct.

1. What is wrong with the declared type of the method equals, and what consequence will this have?
2. The implementation of the equals method has two serious flaws. What are they?

F Object Models

The following is an excerpt from the class `java.util.TreeMap` in the Java collections framework, which provides a red–black tree implementation of a mapping. You don't need to understand red–black trees to answer this question, but you do need to know that they are ordered trees with a complex balancing operation that involves executing methods such as `rotateLeft`. The implementation establishes the ordering using the private method `compare`, which either uses a comparator object provided or the constructor (not shown), or the `compareTo` method of the key.

```

1  public class TreeMap<K,V> ...{
2      private Comparator<? super K> comparator = null;
3      private transient Entry<K,V> root = null;
4
5      static class Entry<K,V> implements Map.Entry<K,V> {
6          K key;
7          V value;
8          Entry<K,V> left = null;
9          Entry<K,V> right = null;
10         Entry<K,V> parent;
11
12         Entry(K key, V value, Entry<K,V> parent) {
13             this.key = key;
14             this.value = value;
15             this.parent = parent;
16         }
17         ...
18     }
19
20     public TreeMap() {}
21     ...
22     private void rotateLeft(Entry<K,V> p) {
23         Entry<K,V> r = p.right;
24         p.right = r.left;
25         if (r.left != null)
26             r.left.parent = p;
27         ...
28         r.left = p;
29         p.parent = r;
30     }
31
32     public V put(K key, V value) {
33         Entry<K,V> t = root;
34         if (t == null) {
35             incrementSize();
36             root = new Entry<K,V>(key, value, null);
37             return null;

```

Name:

TA's name:

```
38     }
39     while (true) {
40         int cmp = compare(key, t.key);
41         if (cmp == 0) {
42             return t.setValue(value);
43         }
44         ...
45     }
46 }
47
48 private int compare(K k1, K k2) {
49     return (comparator==null ? ((Comparable<K>)k1).compareTo(k2)
50         : comparator.compare((K)k1, (K)k2));
51 }
52 ...
53 }
```

1. Draw an object model in the space to the right of the code that shows the essential structure of this representation. Include all multiplicity markings., but ignore final markings for now.
2. Two of the fields should have final markings on their edges. Mark them on your object model, and explain their significance very briefly.