

Name:

TA's name:

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
6.170 LABORATORY IN SOFTWARE ENGINEERING

FALL 2005

Quiz [SOLUTIONS]

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 [30 points total]

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 [T]
2. C' is a behavioral subtype of C [F]
3. Unless overridden, every method of C is implicitly a method of C' [T]
4. Any code expecting an object of type C will behave as expected when passed an object of type C' [F]
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 [F]
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 [F]
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 [F]
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 [T]

Question 8 assumed methods were public. We gave credit to students who argued that the answer should be False because of private methods.

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 [F]
10. Mark every class in the program as final [F]
11. Check that every class is a true (behavioral) subtype of its superclass and the interfaces it implements [T]

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 [T]
13. Should always be executed as a runtime assertion [F]
14. Should hold at the start and end of all public methods of the type [T]
15. Should hold at the start and end of all methods of the type [F]
16. Should be efficiently computable [F]

Name:

TA's name:

17. Allows methods to be reasoned about independently [T]
18. Is useful only when an abstraction function is also recorded [F]
19. Cannot be non-deterministic [T]
20. Can never be the constant predicate *false* [T]
21. Is called an 'invariant' because its value must not change during the execution of a method [F]
22. Applies only to immutable datatypes [F]

The precondition of a method

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

A hashCode method

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

Cognitive delay

35. Between sensing and muscle response is typically about 240ms [T]
36. In perceptual processing is typically 10ms [F]
37. Provides the parameters for Fitt's Law, which can guide the design of pointing tasks [T]
38. If shorter, would suggest greater use of progress bars and delay indications [T]

Java exceptions

39. Are objects themselves [T]
40. Are lightweight and should be used extensively [F]

Name:

TA's name:

- 41. Should always be used in preference to boolean return values [F]
- 42. Should be used to signal any violation of a precondition [F]
- 43. Can violate 'failure atomicity', if thrown too late [T]
- 44. Can violate 'representation independence', if thrown too late [F]
- 45. May be propagated without an explicit *throw* [T]

The object model of a program

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

Runtime assertions

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

Testing

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

B Substitutability [10 points total]

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.

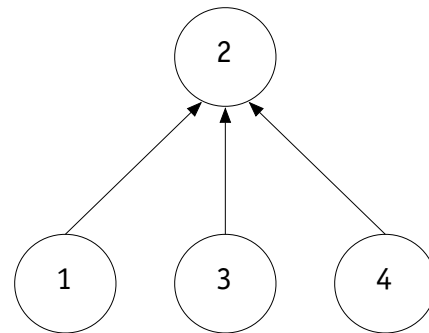
Assume the cache has a maximum size `MAXSIZE`, and that `size(c)` gives the current space consumption of a cache `c`, and `size(p)` gives the space required for storing a page `p`.

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.

add₁, add₃, add₄ all refine add₂. Many of you thought that add₄ refines add₃, because throwing an exception is 'doing more' than doing nothing. But for a client expecting no exception to be thrown, a method that throws one is no good. Some of you noted that the postcondition of add₁ was inconsistent with the

Name:

TA's name:

informal comment about maximum size. Even if that were so, a method that satisfied the spec would have to magically overcome the problem.

Grading: out of 10; 2 points for putting add_2 at the top; 2 points for each edge present; 2 points for no additional edges.

Name:

TA's name:

C Abstraction Functions and Representation Invariants [20 points total]

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:

2 points each

1. $size \geq 0$ [both: `add` only increments `size`, and `choose` does not decrement it when zero]
2. $size \leq elements.length$ [both: neither changes the length of the array, only `add` increments `size`, and if `size` is equal to length in the prestate, the first statement will throw an `ArrayOutOfBoundsException`, and the increment won't happen].
3. $\forall i: int \mid i \geq 0 \wedge i < size \Rightarrow elements[i] \neq null$ [`choose` alone, because it sets no array elements, and only reduces the range of the quantifier; `add` does not check that the arg is non-null]
4. $\forall i, j: int \mid i < j \wedge i \geq 0 \wedge j < size \Rightarrow elements[i].compareTo (elements[j]) \leq 0$ [`choose` alone, because it doesn't mutate the array, and reduces the range of the quantifier; `add` inserts a new element, and doesn't compare it to existing ones]
5. $\forall i: int \mid i \geq size \wedge i < elements.length \Rightarrow elements[i] == null$ [`add` alone, because it reduces the range of the quantifier; `choose` fails to set `elements[size]` to null. If `add` is invoked with a null argument, and `size` equal to `elements.length`, the first statement will throw an exception and the

Name:

TA's name:

second statement will not be executed, but the array will not be mutated, so the invariant is still preserved.]

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.

2 points

Elements of the array at and above the size index are not abstractly part of the set. Holding on to their references prevents garbage collection.

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

3 points

If the array index is out of bounds on the array setter, an exception will be thrown and the array will not have been modified. But the increment of size will have already occurred. So rep invariant 2 is violated; the method is not failure atomic. Some students claimed the modified code would throw an exception when calling add on an empty set: perhaps they forgot the increment of the index, which would cause the array to be accessed correctly at index 0.

8. Give a plausible abstraction function for this representation.

5 points: 1 for correct type (function from rep to set); 1 for referring to the elements of the array; 2 for a range involving size; 1 for getting it right, including appropriate scoping of variables.

$A(s) = \{s.elements[i] \mid 0 \leq i < s.size\}$

D Datatype Theory [15 points total]

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 points

all e: entries | e != null and e.key != null

Some of you also included an invariant that the keys of different entries are distinct. This might be suggested by the spec, but it isn't necessary.

2. Returning null in getEntry is controversial. Give one reason for, and one reason against, this decision.

2 points

For: Easy and simple way to handle special case; alternatives (such as extra observer method or throwing exception) are more work for client and implementor. Against: Client isn't forced to check that result returned is non-null, so might lead to obscure NullPointerException later.

Name:

TA's name:

A common error was to think that the null value returned might be confused with a null key. This can't happen because the object returned is the entry, and not the key.

3. What's a more serious problem with the `getEntry` method?

2 points

It exposes the rep of `ArrayMap`; can break the rep invariant by setting the key field of the returned entry to null.

Several students pointed out that the implementation doesn't handle null keys properly, and their answers to this and subsequent parts followed this line of reasoning, for which we gave credit.

4. Give an example of a fragment of client code that will fail because of this problem.

4 points

```
ArrayMap <String, String> m = new ArrayMap <String, String> ();  
m.put ("a", "b");  
Entry <String, String> e = m.getEntry ("a");  
e.key = null;  
m.getEntry ("a"); // throws NPE
```

A common mistake was to show some code in which mutating the key of the returned entry caused a subsequent lookup to behave differently. This might seem wrong, but it could plausibly be allowed by the spec. In contrast, the failure in this sample code breaks the rep invariant that `getEntry` depends on.

5. Explain how to fix the problem, while still making the method `getEntry` available to clients.

5 points

Make key field of `Entry` final, or make fields of `Entry` private and provide get/set methods, but not setKey. Could also use Decorator pattern by creating a wrapper around an entry with setKey not implemented, but this is overkill here. Note that returning a copy of the `Entry` object is no good, as it prevents the mutation from being reflected back in the map itself – the reason for returning `Entry` objects in the first place. Nor is making `Entry` immutable, which also defeats the point.

Name:

TA's name:

E Equality [15 points total]

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?

3 points

Argument type should be Object. Consequence is that it doesn't override Object.equals, but instead overloads the method name equals. Overloads are resolved by compile-time type, so if you call equals on a Pixel object bound to a variable of declared type Object, the wrong method will be called. (Should also have been public; gave credit for this too.)

2. The implementation of the equals method has two serious flaws. What are they?

4 points

Not transitive, and doesn't return false when argument is null. Several of you also noted problems in the implementation of approx: for example that because byte is signed, the subtraction might overflow. In fact, this is a hopeless way of comparing colors anyway, since very different RGB values can represent colors that appear to the eye to be identical.

3. Assuming that this class is complete, what other fundamental problem does it have?

2 points

Name:

TA's name:

Doesn't override hashCode, so violates Object Contract.

4. Suppose you decide to represent pixels instead with the RGB components drawn from a small set, each representing a range of actual values. A friend suggests that you use the design pattern *Factory Method* in its implementation. What does using this pattern involve, and what might the benefit be?

6 points

Involves making constructor private, and providing for clients instead a static method that returns an object of the type (or a subtype). The benefit is that the implementation could keep a pool of Pixel objects and return an existing one, thus reducing memory usage.

F Object Models [10 points total]

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.

*See the diagram. Some common things students got wrong: the target multiplicity of root is ? because it may be null (see the constructor for the empty TreeMap); the source multiplicity of root cannot be !, because that would mean every entry is the root of some tree; the left and right fields have multiplicity ?-? because they can have null values (see the initialization and the test in rotateLeft), and the root is not the left or right value of any other node; the parent field is *-? because the left and right children share a parent, and the leaf nodes have no children; the key field is ?-! because keys can't be shared across entries (because it's a map), and keys can't be null (or the compare method would fail); the value field is *-? however, because values can be shared by entries, and can be null.*

2. Two of the fields should have final markings on their edges. Mark them on your object model, and explain their significance very briefly.

10 points total: 1 point for getting the right boxes; 1 points for the edges; 4 points for the multiplicities (take one off for each error); 1 point for each final marking; 1 point for explaining each. Since the boxes and edges follow very directly from the declarations, most of the points here were for the more subtle aspects.

Final markings are on target ends of key and comparator. Significance: (1) if key was not target-final, then modifications would likely violate the ordering rep invariant of the tree (or it might suggest that balancing was being achieved by moving keys between entries rather than whole entries, which would be unnecessarily complex); (2) if comparator were not target-final, this would suggest that the basis of the ordering of an existing tree could be changed mid-way, which would require rebalancing, and has no useful purpose. Note that parent, left and right can't be target-final, because of the rebalancing evident in rotateLeft.

Name:

TA's name:

