

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
6.170 LABORATORY IN SOFTWARE ENGINEERING

FALL 2002

Quiz 1

October 16, 2002

ANSWERS

ANSWER SHEET / 6170 FALL 2002 / QUIZ 1

NAME

MIT ID NUMBER

EMAIL

TA

<i>solution template</i>			
---------------------------------	--	--	--

*Enter 0 or 1 in each box
All parts are worth one point*

A.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	1	1	0	1	1	1	0	0	1	1	1	0	1	1	1	0	1	1
20	21	22	23	24	25	26	27											
0	1	0	0	1	0	1	0											

B.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	1	0	0	0	1	0	1	0	0	0	0	1	1	1	1

C.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	1	1	1	1	0	1	0	1	0	1	0	1	0

D.

1	2	3	4	5	6
1	1	0	0	0	0

E.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	1	0	1	0	1	0	0	1	0	1	1

<i>question</i>	<i>score</i>
A	
B	
C	
D	
E	
<i>total</i>	

A Shorties

Unchecked exceptions:

- A.1 cannot be caught
False.
- A.2 need not be listed in the throws clause of a method
True. Java doesn't require it, and common convention is not to do it.
- A.3 are objects
True. All exceptions are objects.
- A.4 should be used for returning special results from a method
False. Checked exceptions are better for this purpose.
- A.5 should be used for unexpected failures
True.

Which of the following can be tested by assertions (albeit not always efficiently)?

- A.6 precondition
True.
- A.7 postcondition
True.
- A.8 modifies clause
False. Checking the modifies clause would require looking at every object in the heap.
- A.9 abstraction function
False. An abstraction function maps rep values to abstract values; 'testing it' is meaningless.
- A.10 rep invariant
True.

Which of the following kinds of methods may be found on an immutable abstract data type?

- A.11 creator
True.
- A.12 producer
True.
- A.13 mutator
False.
- A.14 observer
True.
- A.15 inherited
True. An immutable type may be a subclass of another type, some of whose methods might be inherited.

A correctly-written observer method:

- A.16 may modify the rep
True, by benevolent side-effects.
- A.17 may expose the rep
False. Such an observer would be badly written; no class implementing an abstract data type should expose its rep.

- A.18 may return an iterator
True. An iterator is one way to observe the contents of a collection.
- A.19 may lack an explicit modifies clause
True. The default value for the modifies clause is “nothing”; since an observer method should modify nothing, it can omit the modifies clause.

Which of the following methods will return true for objects that are behaviourally equivalent, and false otherwise? (Hint: this is very tricky! Think very carefully before answering.)

- A.20 the equals method inherited from Object
False! Imagine writing an immutable class for strings of characters, MyString, which has the same methods as String but inherits Object.equals instead of overriding it. Then two distinct MyString objects with the same sequence of characters will be behaviorally equivalent, but the inherited equals method will return false.
- A.21 String.equals
True. String.equals returns true exactly when two strings consist of exactly the same characters, and so cannot be distinguished by calls to other methods of String.
- A.22 the equals method of the LinkedList class of the Java library
False. Java collection classes implement observational equivalence, not behavioral equivalence. According to LinkedList.equals, two empty lists are equal, but they are not behaviorally equivalent because you can distinguish them by mutating one of them.
- A.23 this method:

```
public boolean equals (Object o) {
    return this.hashCode () == o.hashCode ();
}
```

False. Two objects with the same hash code are not necessarily behaviorally equivalent.

The Object contract includes the following requirements (among others):

- A.24 equals must be transitive
True.
- A.25 two objects with the same hashcode must be equal
False. Two equal objects must have the same hashcode, but the converse is not required to be true (and usually isn't).
- A.26 a.equals(null) must be false for all objects a
True. This follows from symmetry.
- A.27 objects must be cloneable
False. The clone method is not required to be implemented.

B Abstract Types

Suppose you're developing software to help automate the grading of multiple-choice quizzes. Your design uses two abstract types, `Template`, for representing the structure of a quiz (the number of options per question, for example), and `Solution`, for representing a particular submitted solution for grading, or a solution against which other solutions are graded. Study these skeletons for the two classes:

```
class Template {
    // A <Template> is a set of questions, each consisting of a name and the number of multiple-choice
    // options available. Names are numbers: consecutive integers starting at 1. Templates are immutable.
    ...
    private int [] counts;

    Template ()
    // returns: constructs an empty template with no questions

    Template addQuestion (int numOptions)
    // requires: numOptions > 0
    // returns: a new template with a new question with <numOptions> options, whose number
    // is one larger than the highest numbered question in this template, or 1 if this template is empty

    int getNumOptions (int questionNumber)
    // returns: number of options in question numbered <questionNumber>
    // throws: IndexOutOfBoundsException if no such question
    ...
}

class Solution {
    // A Solution is a set of answers to a quiz. Abstractly, it consists of a Template and a map that associates
    // with each question number in the Template a set of options, represented as integers
    // numbered consecutively from 1. Solutions are mutable.
    ...
    private boolean [] answers;
    private Template template;

    Solution (Template template)
    // requires: <template> is not null
    // returns: constructs a new Solution from <template> in which
    // all questions have the empty set of options selected

    void select (int question, int option)
    // modifies: this
    // effects: adds <option> to the set of options associated with the
    // question numbered <question>
    // throws: NoSuchOption if, in the template of this, there is no such <question> or <option>

    int grade (Solution goldStandard)
    // requires: <goldStandard> is not null
    // returns: grade of this with respect to <goldStandard>
    // computed by scoring 1 for each option correctly included or excluded
    ...
}
```

Which of the following constraints are reasonable to include in the rep invariant of Solution?

- B.1 template is non-null
True.
- B.2 the elements of answers are non-null
False; null is not a possible value of a primitive type such as boolean.
- B.3 the total number of options for all questions in template equals the length of answers
True.
- B.4 template.counts is non-null
False. The rep invariant should not mention the rep of another class.

The method grade is not properly specified, because it does not handle the (unexpected but possible) case in which this and goldStandard have different templates. This case could be handled in different ways. Which is the best solution for the client? (Choose one).

- B.5 Adding a constraint that the templates are equal as a precondition
- B.6 Specifying in the postcondition that when the templates differ, a checked exception is thrown
- B.7 Specifying in the postcondition that when the templates differ, an unchecked exception is thrown
This is the best for the client. It provides an unambiguous outcome, and the use of an unchecked exception allows the client to omit exception handling code when she knows the call will not throw the exception.
- B.8 Specifying in the postcondition that -1 is returned when the templates differ.

Which is easiest for the implementor of Solution? (Choose one).

- B.9 Adding a constraint that the templates are equal as a precondition
This is the easiest for the implementor, because it allows him to ignore the issue completely.
- B.10 Specifying in the postcondition that when the templates differ, a checked exception is thrown
- B.11 Specifying in the postcondition that when the templates differ, an unchecked exception is thrown
- B.12 Specifying in the postcondition that -1 is returned when the templates differ.

If an approach is chosen that requires a test that two template references are equivalent, how should it be implemented?

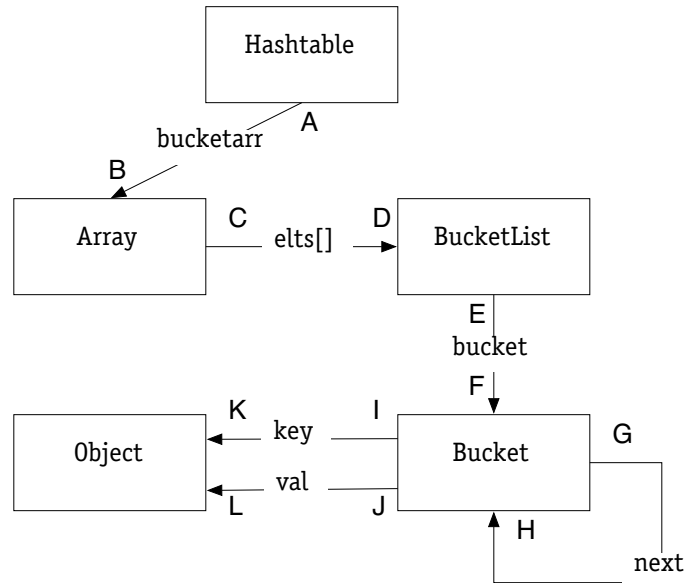
- B.13 with ==
- B.14 with Template.equals
This was a badly phrased question, so we gave credit whatever you said. We meant to ask this: If an approach is chosen that requires a test that two templates are equivalent, how should it be implemented? The answer would then be to use Template.equals. This encapsulates the equality test properly, allows an equality of contents for immutable templates, and is barely less efficient if, within the Template.equals method, a short-circuit test for reference equality is first performed.

Notice that Template is immutable. Which of the following concerns might have been motivations for this design decision?

- B.15 Allowing sharing of Template objects between Solution objects
True. This is crucial to the efficiency of the design.
- B.16 Making it easier to reason about code that uses Template objects
True – a property of all immutable types.

B.17 Avoiding rep exposure in Solution

True, because a Template is taken as an argument of the constructor.



C Object Models

Consider the object model of the representation of a hashtable shown above. The choice of multiplicities can have a major impact. In this diagram, they have been omitted and the letters A through L stand in their place. For each of the design and implementation considerations below, indicate which multiplicities you would need to know to answer the question. You should assume that equality of keys is determined according to observational and not behavioural equivalence.

Is it possible to store null as a value against a key?

- C.1 F
- C.2 J
- C.3 K
- C.4 L

L alone, since this determines whether the val field can be null.

Can the put method be implemented in a uniform way (avoiding, for example, checking whether fields are null and performing special actions accordingly)?

- C.5 B
- C.6 D
- C.7 F
- C.8 G
- C.9 H
- C.10 L

B, D, F and H are the relevant ones. Making B, D and F 'exactly one' (ie, non-null) will ensure that there is no need to create a new array, list or bucket. Ensuring that H is 'exactly one' means that when a bucket is to be spliced in, its successor will be readily available.

Can the rep invariant be compromised by sharing?

C.11 A

C.12 B

C.13 C

C.14 D

C.15 I

C.16 J

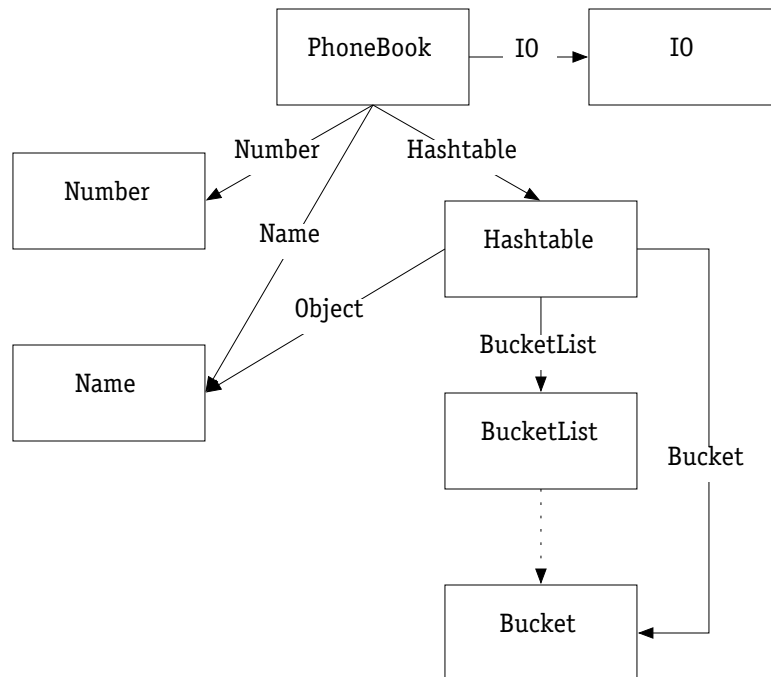
A, C, I. The first two follow from the simple principle of not sharing internal rep objects. The last is much more subtle: sharing the key will actually be an exposure unless observational equivalence is used, since, if it is mutable, a mutation may break the invariant that keys are in their appropriate buckets according to their hash code.

D Module Dependences

You are reviewing an implementation of a phonebook program that supports the storage and retrieval of phone numbers against names. You are shown the module dependence diagram reproduced below. The dotted edge is a *name dependence*. The edge from Hashtable to Name represents a dependence on the object contract of Name.

Which of the following inferences can you draw from the dependence diagram?

- D.1 The BucketList type used within Hashtable is not an abstract data type in its own right.
True. BucketList has only a name dependence on Bucket, so it doesn't access its fields. The diagram also shows that Bucket is accessed directly by Hashtable.
- D.2 In the absence of additional data structures not shown, or a perverse use of the data structures already present, the program does not support reverse lookup (from numbers to names).
True; otherwise there would be an Object Contract dependence of Hashtable on Number as well as Name.
- D.3 The Hashtable class cannot be reused in a program that does not also include the Name class.
False. The dependence is only a dependence on the Object Contract of Name.
- D.4 A Bucket object cannot point to a Name object.
False. Object references do not necessarily induce dependences.
- D.5 Since there is no edge from Hashtable to Number, there must be a bug, or the diagram must be wrong, since the lack of an edge means that Number objects cannot be stored in Hashtable objects.
False, for the same reason.
- D.6 The edge from PhoneBook to IO suggests that the PhoneBook class has a field that holds an IO object.
False, for the same reason.



E Testing

Consider the following method and answer the questions about it that follow:

```
static String[] search (String[] words, String s) {
    // throws: NullPointerException if <s> or any element of <words> is null
    // returns: an array consisting of every string in <words> that contains <s>
    //          as a substring

    // initially allocate space for 10 elements
    List result = new ArrayList (10);

    if (words != null) {
        for (int i = 0; i < words.length; ++i) {
            String w = words[i];
            // test whether w contains s as a substring;
            // throws NullPointerException if either w or s is null
            if (w.indexOf (s) != -1) {
                result.add (w);
            }
        }
    }
    return (String[]) result.toArray (new String[result.size ()]);
}
```

Select a minimal subset of the test cases below that achieves 100% statement coverage for the search function.

- E.1 search ([], null)
False.
- E.2 search (["beeblebrox", "dent", "trillian"], "frodo")
False.
- E.3 search (null, "frodo")
False.
- E.4 search (["apple", "banana", "grape"], "a")
True. This test case is required for 100% statement coverage, because it's the only one that executes result.add(w). It is also sufficient for 100% coverage, because it causes every statement to be executed.
- E.5 search ([null, null, null], "a")
False.

Select a minimal subset that achieves 100% decision coverage.

- E.6 search (null, "frodo")
True. This test case is required to take the decision words!=null in the false direction.
- E.7 search (["apple", "banana", "grape"], "a")
False. If you included this test case, you'd also need E.10 to test the w.indexOf(s) decision in the other direction, which isn't minimal.
- E.8 search (["french", "english", "thai"], "en")
True. This test case handles both directions of the w.indexOf(s) decision, so using it instead of E.7 and E.10 gives a minimal subset.

- E.9 search ([null, null, null], "a")
False. The search procedure has no decision point for null words. This decision is made deeper, inside indexOf.
- E.10 search (["beebro", "dent", "trillian"], "frodo")
False. See E.7 and E.8 for explanation.

Which of the following subdomains might be suggested by black-box testing?

- E.11 s is null
True.
- E.12 words has more than 10 elements
False. The magic number 10 appears in the code, not the spec, so this subdomain would be found by glass-box, not black-box, testing.
- E.13 words includes a null element
True.
- E.14 words has no element containing s as a substring
True.