

Your name: \_\_\_\_\_ **SOLUTIONS** \_\_\_\_\_

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
6.170 LABORATORY IN SOFTWARE ENGINEERING  
SPRING 2000

Quiz 1  
March 1, 2000

There are 4 sections (labeled A through D) and 9 pages. Please check your copy of the quiz before you start to make sure it is complete.

**You have 50 minutes. Note that there are a total of 50 points on the exam.**

Write your name at the top of **every** page of this exam before you start.

**Name:** \_\_\_\_\_ **SOLUTIONS** \_\_\_\_\_.

**TA Name:** \_\_\_\_\_

<b>Section</b>	<b>Max</b>	<b>Score</b>	<b>Grader</b>
A	10		
B	10		
C	15		
D	15		
<b>TOTAL</b>	<b>50</b>		

**A. True/False (10 points, 10 questions)**

Write *TRUE* or *FALSE* in the blank to the left of each question.

- False** 1. It is easier to implement a specification that requires true than to implement one that requires false.
- False** 2. The purpose of a transition relation is to describe exactly what changes occur to an object when it is modified.
- True** 3. The value 5 is not an object in Java.
- True** 4. The only way to detect aliasing of immutable objects is to use the == operator.
- True** 5. There may be an infinite number of implementations of a given specification.
- True** 6. Checked exceptions are preferable to unchecked exceptions in cases where the using program would normally catch the exception.
- False** 7. The revealing subdomains for a given procedure are determined by the specification of that procedure.
- F and T** 8. It is possible to design both immutable and mutable ADTs to represent the same abstract concept.
- False** 9. The purpose of testing is to determine whether or not a program is correct.
- True** 10. A procedure specification that requires something other than true or false indicates that the procedure is a partial procedure.

**B. Short Answer (10 points, 5 questions)**

11. Write the precise definition of what we mean when we say that a procedure P satisfies a specification S.

Full credit was given for answering this either in terms of transition relations, or in terms of requires/modifies/effects. Here is the T.R. answer.

For all  $x$  in the domain of the transition relation corresponding to S,

1. Exists some  $(x,z)$  in the relation corresponding to P
2. For all  $(x,z)$  in P,  $(x,z)$  is in S

12. Write a code fragment for which a test suite could achieve 100% coverage as measured by statement coverage but not 100% coverage as measured by decision coverage.

```
if (x > 0) {  
    x = y;  
}  
z = z/x;
```

13. What is the primary reason that it is desirable to write an underdetermined specification, in cases where this is acceptable?

Gives more flexibility to the implementer.

14. Describe a situation that shows the difference between `similar` and `equals`, assuming that they behave as defined in the Liskov textbook. In other words, describe a type `T` (including whether it is mutable or immutable), and values of the variables `a` and `b` of type `T`, in which `a.similar(b) != a.equals(b)`. What is the value of `a.equals(b)` in this situation?

```

Mutable type: Vector.
Vector a = new Vector();      value: ["hello", "goodbye"]
Vector b = new Vector();      value: ["hello", "goodbye"]

a.similar(b) => true
a.equals(b)  => false

```

15. Here are two specifications for the same procedural abstraction.

```

int version1(int[] a) throws EmptyArray, NullPointerException
// requires: true
// effects: returns the least i such that a[i] >= a[j], 0 <= j < a.length
//          unless
//          a is null => throws NullPointerException
//          a.length = 0 => throws EmptyArray

```

```

int version2(int[] a) throws EmptyArray, NullPointerException
// requires: true
// effects: returns some i such that a[i] >= a[j], 0 <= j < a.length
//          unless
//          a is null => throws NullPointerException
//          a.length = 0 => throws EmptyArray

```

Which version is weaker, and what precisely do we mean when we say that it is weaker?

```

Version 2 is weaker.
It is weaker because any implementation of version 1 will also
satisfy version 2.

```

**C. ADTs (15 points, 5 problems)**

Consider the following ADT specification (some parts have been omitted).

```
class IString {
    // An IString is an immutable string of characters.
    // ...

    public void IString();
    public void IString(char c);
    public IString concatenate(IString s);
    public int length();
    public IString substring(int start, int length) throws ArgOutOfBounds;
    public char charAt(int index) throws ArgOutOfBounds;
    public int find(char c, int start_index)
        throws NotFound, ArgOutOfBounds;
}
```

- To *concatenate* is to put two strings together. So if IString **a** represents "abc" and IString **b** represents "def", then **a.concatenate(b)** returns an IString that represents "abcdef".
- The *length* is the number of characters in the string.
- To *substring* is to extract a piece of a string. The characters in a string are indexed from 0, so if IString **a** represents "abcdef", then **a.substring(2, 1)** returns an IString that represents "c", while **a.substring(3, 3)** returns an IString that represents "def".
- **a.charAt(1)** returns the character 'b'.
- *Find* looks for a character in the string, starting from the index **start\_index**, and returns the index at which that character occurs if it is present.

16. Describe a possible representation for IString. Your answer will consist of the following: write the field declarations of the class (a field declaration is something like "int x;"), write a short comment explaining the function of each field, and provide an implementation of the **length()** operation.

Several representations were acceptable for this problem. Here is one.

```
char[] chars; // stores the characters in the string

public int length() {
    return chars.length;
}
```

17. Write the omitted part of the overview section (shown as ... in the code).

Missing from the overview was a description of the abstract state and an example.

```
// Abstractly, a string is an indexed sequence of characters,
// [c_0, c_1, c_2, ..., c_n]
// An example string is ['h', 'e', 'l', 'l', 'o'], also denoted
// "hello".
```

18. Write a specification for the substring method that matches the behavior described on the previous page. Your specification should make **substring** a complete function.

```
// effects: If length is zero, returns an empty IString.
// If length is nonzero, returns the
// IString [c_start, c_start + 1, ..., c_(start + len - 1)]
// unless,
// start < 0, start >= (length of this),
// or length < 0, or start + len > (length of this)
// => throws ArgOutOfBounds
```

Here is a part of an implementation of this ADT.

```
private char found_char;
private int search_start; // initialized to -1
private int search_found;

public int find(char c, int start_index)
    throws NotFound, ArgOutOfBounds
{
    if (start_index < 0) {
        throw new ArgOutOfBounds();
    }
    if ( (c == found_char) && (start_index == search_start) ) {
        return search_found;
    } else {
        found_char = c;
        search_start = start_index;
        search_found = InternalFind(c, start_index);
        return search_found;
    }
}

// same specification as find()
private int InternalFind(char c, int start_index)
    throws NotFound, ArgOutOfBounds
{ ... }
```

19. This implementation of `find` modifies the object, even though the ADT is declared to be immutable. What do we call this kind of a modification? Explain why it is acceptable to modify an immutable object in this way.

This type of modification is called a benevolent side effect. It is acceptable because the abstract state of the object is unchanged. That is, the user can discern no change by calling the object's observers.

20. This implementation of `find` has a bug. Write a test case that exposes it and say what the incorrect output of this implementation would be.

`Find()` stores the result of the last `find` operation, on character `<found_char>` starting from character `<search_start>` in `<search_found>`.

The bug occurs when `InternalFind` cannot find the character and throws a `ArgOutOfBounds` exception. This call is propagated to `find`'s caller. `<search_start>` is set to `<start_index>`, and `<found_char>` is set to `<c>`. `<search_found>` is unset, however, since `InternalFind` throws an exception.

```
IString is = new IString('a');
try {
    int ix = is.find('b', 0); // exception thrown by first call
    System.err.println("fail!");
} catch(ArgOutOfBoundsException e) {
    try {
        int ix2 = is.find('b'); // no exception thrown by this call
        System.err.println("fail!"); // "fail!" is printed here
    } catch(ArgOutOfBoundsException e2) {
        System.err.println("pass!");
    }
}
```

**D. Testing (15 points, 3 problems)**

Consider the following function.

```
int indexOfMax(int[] a) throws EmptyArray, NullPointerException
// requires: true
// effects: returns the least i such that a[i] >= a[j],
//           0 <= j < a.length
//           unless
//           a is null => throws NullPointerException
//           a.length = 0 => throws EmptyArray
{
    if (a.length = 0) throw new EmptyArray();
    int max = a[0];
    int answer = 0;
    for (int i=1; i<a.length; i++) {
        if (a[i] > max) {
            max = a[i];
            answer = i;
        }
    }
    return answer;
}
```

21. Write a minimal complete black box test suite. Include boundary cases. Write a brief comment explaining why you included each test case.

- You may use the back of this page if you run out of room.
- You do *not* need to write out each test case as a piece of driver code. It is sufficient to show the input and expected output in a table with one row per test case.

**Black Box cases (1 point each, total 5 points):**

```
1) null => NullPointerException // test throwing
                               NullPointerException
2) [] => EmptyArray           // test throwing EmptyArray
3) [3] => 0                   // test 1-element array
4) [2, 3] => 1                // test many-element array
5) [3, 3] => 0                // test returning the least index
```

**Boundary cases (2 points for any 2 reasonable cases):**

```
6) [3, 2] => 0                // test many-element array with max at 0th
                               element
7) [1, 3, 2] => 1             // test many-element array with max not the
                               0th element nor the last element
8) [3, 4, 4, 0] => 1          // test many-element array with repeated max
                               in the middle
9) [3, 0, 4, 4] => 2          // test many-element array with repeated max
                               at the end
10) [4, 3, 0, 4] => 0         // test many-element array with repeated max
                               not next to each other
```

22. Write additional test cases, if any, needed to make the test suite from problem 1 a decision-complete glass-box test suite. Explain why each case is needed. You may use the back of this page if you run out of room.

No additional tests are needed. Case 2, case 3, case 4, and case 5 above provide decision-complete coverage.

23. Write additional test cases, if any, needed to make the test suite from problem 1 a path-complete glass-box test suite, using the same definition of path-complete that was used in the problem sets. Explain why each case is needed. You may use the back of this page if you run out of room.

The path that ends with throwing EmptyArray exception was covered before.

Same for the paths in which the for loop is executed 0 time, 1 time in which the body of the if statement is executed, and 1 time in which the body of the if statement is not executed.

So, need 4 more cases to cover the paths that go through the for loop 2 times in which the body of the if statement is (a) not executed both times, (b) executed both times, (c) not executed the first time and executed the second time, and (d) executed the first time and not executed the second time.

(d) is covered by case 7 above. So, 3 additional cases:

For (a), [3, 2, 1] => 0  
For (b), [1, 2, 3] => 2  
For (c), [2, 1, 3] => 2