

Massachusetts Institute of Technology
6.170 Laboratory in Software Engineering
Spring 2001

Quiz 1

Wednesday, March 7, 2001

Name: *Solutions* _____

Athena username: _____

Section (circle one):

Section 1, Felix Klock	Section 5, Jeffrey Sheldon	Section 9, Kenneth Lu
Section 2, Allison Waingold	Section 6, Andreas Hofman	Section 10, David Maze
Section 3, Matt Deeds	Section 7, Jeremy Nimmer	Section 11, Delphine Nain
Section 4, Sameer Ajmani	Section 8, John Holmes	

This quiz is 50 minutes long. It contains 39 questions (you will answer 32 questions on this sheet, and will answer the remainder elsewhere) and 7 pages (excluding this one). Please check your copy to make sure it is complete before you start. You may separate this sheet from the rest of the test, but turn in all pages, together, when you are finished. Write your username and section on the top of pages 5–7.

Please write neatly; we cannot give credit for what we cannot understand.

The problems are weighted as follows:

true-false : multiple-choice : short-answer :: 1 : 2 : 6

Expect to spend (on average; some problems are easier, some are harder) about 30 seconds per true-false question, 1 minute per multiple-choice question, and 3 minutes per short-answer question.

Good luck!

True/false:

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

13	14	15	16	17	18	19	20	21	22	23	24

Multiple choice:

25	26	27	28	29	30	31	32

True/False [1 point each]

1. Class type hierarchies (classes and subclasses, no interfaces) in Java form tree structures. **True**
2. Interface type hierarchies in Java form tree structures. **False**
3. A good test suite proves the absence of errors in a program. **False**
4. The revealing subdomains for a given procedure are determined by the specification of that procedure. **False**
5. Two inputs are in the same revealing subdomain if a program has the same behavior on both inputs. **False**
6. Regression testing tests all modules together, to validate the overall program. **False**
7. It is often possible to test individual methods of an ADT in isolation using unit testing. **False**
8. The goal of debugging is to eliminate the current bug as quickly as possible. **False**
9. Black-box testing of a procedure should usually include cases where the requires clause is not met. **False**
10. Derived specification fields may change from implementation to implementation, so clients should not depend on them. **False**
11. It is possible that the code of `A.foo` is executed during evaluation of `(new B()).foo()` in the following?

```
class A {  
    ...  
    public foo() { ... }  
}  
class B extends A {  
    ...  
    public foo() { ... }  
}
```

True. `B.foo` might contain an instance of “`super()`”.

12. Java differs from Scheme in that Scheme objects are heap-allocated, whereas Java’s are declared. **False. Both have heap-allocated objects.**
13. Java differs from Scheme in that Scheme objects have no runtime type, whereas Java’s do. **False. Both have objects with runtime types.**

Write your answers to these questions on the answer page, not on this page.

14. After a program accesses an object for the last time, the object will (eventually) be garbage-collected. **False. There might still be references to the object. (Also, the program might not ever terminate.)**
15. The Java compiler alerts programmers of potential errors by enforcing that compile-time types and run-time types are identical. **False.**
16. The Java compiler ensures the absence of array bounds errors at runtime. **False. Programmers can still write code with this error, which will cause an exception to be thrown.**
17. Performance-critical code usually should not declare exceptions, for reasons of efficiency. **False. Declaring exceptions has little to no effect on performance. Throwing exceptions is expensive in some implementations (but not in others), so it is sometimes avoided in performance-critical code.**
18. When a module is replaced by another module with the identical specification, integration tests should be repeated. **True**
19. A class that overrides `equals` should also override `hashCode`. **True**
20. An ADT does not always guarantee deterministic behavior. **True**

For questions 21–24, assume

- B extends A,
 - a refers to an object whose compile-time and run-time type is A, and
 - b refers to an object whose compile-time and run-time type is B.
21. `b.apply(a)` can behave differently than `((B) b).apply(a)`. **False.**
 22. `b.apply(a)` can behave differently than `((A) b).apply(a)`. **False.**
 23. `b.apply(a)` can behave differently than `b.apply((B) a)`. **True. This throws a `ClassCastException`.**
 24. `b.apply(a)` can behave differently than `b.apply((A) a)`. **False.**

Multiple choice [2 points each]

25. Which of the following is the easiest requires clause to satisfy?

- (a) `requires: arguments are non-null`
- (b) `requires: false`
- (c) `requires: true`
- (d) none of the above

C. It makes no demands on the caller whatsoever. It only makes sense for a client (caller) to satisfy a requires clause — that is, to make it true. The implementation (callee) gets to assume that the requires clause is already satisfied. (Both the callee and the caller are required to satisfy the specification as a whole; each one has different obligations in that case.)

26. Suppose that the specification of `Quux.foo()` is stronger than the specification of `Bar.foo()`. A program that works when it calls `Bar.foo()`

- (a) will work if it calls `Quux.foo()`
- (b) will not work if it calls `Quux.foo()`
- (c) may work if it calls `Quux.foo()`

A

27. Suppose that the specification of `Quux.foo()` is stronger than the specification of `Bar.foo()`. A program that works when it calls `Quux.foo()`

- (a) will work if it calls `Bar.foo()`
- (b) will not work if it calls `Bar.foo()`
- (c) may work if it calls `Bar.foo()`

C

28. Specification fields should usually be

- (a) Java classes such as `Vector`
- (b) Java interfaces such as `List`
- (c) mathematical types such as “sequence”
- (d) none of the above
- (e) a or b
- (f) a, b, or c

C

29. Which of the following procedural specifications is stronger?

- (a) requires: $x \geq 0$
- (b) requires: $x > 0$
- (c) equally strong
- (d) incomparable

A: A is stronger than B because A requires less than B does, so A can be used wherever B is. Recall $(x \geq 0) == ((x > 0) \vee (x == 0))$.

30. Which of the following procedural specifications is stronger?

- (a) modifies: nothing
- (b) modifies: x
- (c) equally strong
- (d) incomparable

A: A is stronger than B because A modifies less than B does, so A can be used wherever B is.

31. Which of the following procedural specifications is stronger?

- (a) returns: 5
- (b) returns: 5 if $x \geq 0$
throws: `NegativeException` if $x < 0$
- (c) equally strong
- (d) incomparable

D: Neither specification satisfies the other when $x < 0$.

32. Which of the following procedural specifications is stronger?

- (a) requires: $x < 0$
returns: a value > 0
- (b) requires: $x \leq 0$
returns: a value ≥ 0
- (c) equally strong
- (d) incomparable

D: Incomparable

Short answer [approximately 6 points each]

33. Can a deterministic procedure satisfy an underdetermined spec? If so, give an example. If not, explain (in one sentence) why not. [6 points]

Sidebar 3.2 of the Liskov text states, “A procedure is underdetermined if for certain inputs its specification allows more than one possible result.” and “An implementation of a procedure is deterministic if, for the same inputs, it always produces the same result.”

Even though the specification allows more than one possible result, the procedure only needs to produce one of them. For example:

```
// returns: an arbitrary int
int foo() { return 3; }
```

34. Write a code fragment for which it is possible to write a test suite that achieves 100% statement coverage but not 100% decision (edge) coverage. (You need not write down the test suite.) [5 points]

```
if (p) {
    foo();
}
bar();
```

A test suite for which p is true would execute every line of the code, but would not exercise the p-is-false condition for the if condition — nor would it exercise the path in which bar(); is executed but foo(); is not. Decision coverage is sometimes called edge coverage because execution flows along each edge in the control flow graph.

35. Give an example of an immutable abstraction with a mutable representation. (The example may be in words or in code.) Briefly (one sentence) describe why an immutable representation would be inferior. [6 points]

One example is an immutable set abstraction implemented via a Vector representation. This permits the move-to-front optimization: lookups can put the requested item at the front of the vector, which is likely to substantially reduce total runtime if there is temporal locality (repetition) among the items looked up.

36. The following program finds the index of an array element by recursively dividing the array in half.

```
public class Buggy {
    // returns: the first index, between left and right inclusive, of n in array
    // throws: RuntimeException if n does not appear in a[left..right]
    private static int indexOf(int[] a, int n, int left, int right) {
        // ... recursive implementation elided ...
    }

    public static int indexOf(int[] a, int n) {
        return indexOf(a, n, 0, a.length-1);
    }

    public static void main(String argv[]) {
        int[] a = { 0, 5, 3, 2, 7, 6, 9, 1, 4, 8 };
        System.out.println("indexOf(0): " + indexOf(a,0));
        System.out.println("indexOf(1): " + indexOf(a,1));
    }
}
```

The program throws an exception after printing the index of 0. Give two hypotheses for the source of the error and briefly discuss how you would test those hypotheses, including what you would learn from your investigations. [8 points]

One hypothesis is that the second operation always fails, perhaps because of caching in the implementation. This can be tested by trying to look up a single value twice. If the second lookup succeeds, then the hypothesis is refuted.

Another hypothesis is that the array needs to be sorted. This can be tested by trying the same lookups, but changing a as follows:

```
int[] a = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

If the lookup of 1 still fails, then the hypothesis is refuted.

Another hypothesis is that lookups anywhere except the first element fail. This can be tested by trying the same lookups (either in the same order or a different order), but changing a as follows:

```
int[] a = { 1, 0 };
```

If the lookup of 0 still works, then the hypothesis is refuted.

37. Give an example of classes A and B such that B is a behavioral (true) subtype of A but not a Java subtype (subclass) of A, or briefly (one sentence) explain why no such pair exists. [5 points]

The following two classes satisfy all the requirements for A to be a true subtype of B.

```
class A { }  
class B { }
```

They would be Java subclasses if the code were changed as follows:

```
class A { }  
class B extends A { }
```

38. Briefly (one sentence each) give two reasons why specifications are advantageous. [5 points]

Specifications permit reasoning about behavior in the absence of code.

Specifications permit replacement of code by other code with the same (or a stronger) specification.

39. Briefly (one sentence each) give two situations in which a require clauses is preferable to specifying that an exception is thrown. [7 points]

If the condition is very expensive to check, then it is better (except during debugging) to require a particular property to be true than to check it in order to ensure that the right exception is thrown.

If the method is private and/or meant to be called from strictly controlled situations in which the property has already been checked and/or established, then it is not advantageous to clutter the code with exceptions that will never be thrown by well-behaved, correct executions.