University of Washington
CSE 331 Software Design & Implementation
Winter 2011

Midterm exam

Monday, January 31, 2011

Name: ***Solutions***

CSE Net ID (username):

UW Net ID (username):

This exam is closed book, closed notes. You have **50 minutes** to complete it. It contains 27 questions and 8 pages (including this one), totaling 100 points. Before you start, please check your copy to make sure it is complete. Turn in all pages, together, when you are finished. **Write your initials on the top of *ALL* pages.**

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

Good luck!

Page	Max	Score
2	14	
3	18	
4	12	
5	8	
6	24	
7	12	
8	12	
Total	100	

1 True/False

(2 points each) Circle the correct answer. T is true, F is false.

1. **T /** **F** When a client calls a method without meeting the preconditions, the method must throw the exception that is stated in the specification.

2. **T** **/ F** If S1 is a stronger specification than S2, then any implementation that satisfies S1 must also satisfy S2.

3. **T /** **F** If an ADT is immutable, then its representation cannot change.
The abstract value may not change, but the representation may change. An example is benevolent side effects.

4. **T** **/ F** The rep invariant may be satisfied by a concrete state (values of the fields) that cannot occur at run time. (By "cannot occur", we mean that there is no sequence of method calls and field assignments that will produce that concrete state.)

5. **T /** **F** The rep invariant is part of an ADT's specification.

6. **T /** **F** The abstraction function is part of an ADT's specification.

7. **T** **/ F** Derived specification fields are part of an ADT's specification.

2 Multiple choice

Mark each of the following that can be true:

8. (3 points) If you split a revealing subdomain, then:

   (a) each piece is revealing

   (b) each piece is non-revealing

   (c) some pieces may be revealing, and some pieces may be non-revealing

   (d) some pieces may be revealing, some pieces may be non-revealing, and some pieces may be neither revealing nor non-revealing

   a: each piece is revealing

9. (3 points) If you split a non-revealing subdomain, then:

   (a) each piece is revealing

   (b) each piece is non-revealing

   (c) some pieces may be revealing, and some pieces may be non-revealing

   (d) some pieces may be revealing, some pieces may be non-revealing, and some pieces may be neither revealing nor non-revealing

   c: some pieces may be revealing, and some pieces may be non-revealing

Circle one of "may", "must", or "must not". The comparison is to the method that is being overloaded/overridden. Use Java's definitions. For simplicity, assume the method has one formal parameter. (2 points each)

10. An overloading method **may** / **must** / **must not** change the declared formal parameter types.

11. An overloading method **may** / **must** / **must not** change the declared return type.

12. An overloading method **may** / **must** / **must not** return an object of a different class.

13. An overriding method **may** / **must** / **must not** change the declared formal parameter types.

14. An overriding method **may** / **must** / **must not** change the declared return type.
    The change must be covariant — the return type must be a subtype of the overridden method's return type.

15. An overriding method **may** / **must** / **must not** return an object of a different class.

3 Short answer

16. (4 points) Explain in one sentence why "unit testing" was given that name.

    It doesn't test your whole system, but only one unit of it: a single method, class, or interface. Errors in other parts of the system are* not *supposed to make the unit test fail.

17. (4 points) What is the relationship between the `hashCode` and `equals` methods? (Answer in one sentence or equation.)

    If `a.equals(b)`, then `a.hashCode() == b.hashCode()`.

18. (4 points) In one sentence, what is a benevolent side effect?

    A side effect that does not change the abstract value that the object represents.

Suppose that you have the following code:

```
public class MyInteger {

  private int val;

  public MyInteger(int val) {
    this.val = val;
  }

  public boolean equals(MyInteger other) {
    return other.val == val;
  }

}

MyInteger mi1 = new MyInteger(22);
MyInteger mi2 = new MyInteger(22);
Object o1 = new MyInteger(22);
Object o2 = new MyInteger(22);
```

For each of the following four calls to equals, state whether it causes a compile-time error, a run-time error, returns true, or returns false. (2 points each)

19. `mi1.equals(mi2)` ***returns `true`***
20. `o1.equals(o2)` ***returns `false`***
21. `mi1.equals(o2)` ***returns `false`***
22. `o1.equals(mi2)` ***returns `false`***

23. (6 points) A developer should write black box tests before clear box tests. In no more than two sentences, explain why. Give only one reason. If there are multiple reasons, give the most important reason.

If you write the black box after the clear box tests (and the implementation), then you may subconsciously make the same errors while testing that you did during implementation. These correlated errors are less likely if you write the black box tests before you have an implementation in mind.

Saying "write the tests before the implementation" or "without knowledge of the code" is a restatement of the answer, not an explanation.
The fact that black box tests can be written before the implementation and then can give immediate to the developer as soon as the code is written is a reason, but not the most important reason.
Any answer that conflates clear box tests and implementation tests is incorrect.
Saying that black box tests serve to test the specification, whereas clear box tests serve to test the implementation, is incorrect. Both varieties of test serve test the implementation against the specification, and both produce test cases that are valid for any implementation of the specification. They just use different criteria for choosing test inputs.

24. (8 points) Give a buggy program of no more than 5 lines and a set of invocations that covers every statement but does not reveal the bug.

Two answers are:

```
// returns the absolute value of arg
static int abs(int arg) {
  return 1;
}
```
invocations = `abs(-1)`

```
// returns the lesser of a and b
static int min(int a, int b) {
  int r = a;
  if (a <= b) {
    r = a;
  }
  return r;
}
```
invocations = `min(1,2)`

25. (10 points) Suppose that you have a `Map`, `m`, and a value being used as a key, `k`. You know that `m.get(k)` returns a non-null value, and you wish to make an argument to that effect. You may assume that `m` is non-null. What are the two necessary facts that your argument will rely upon?

(a) ***`m`'s values (the range of the mapping) are all non-null***

(b) ***`k` is a key for `m`; that is, `k` is in the domain of the map. This is necessary because `Map.get` returns null otherwise.***

It is fine for `k` to be null, assuming that the map supports null keys.

How do you express each of these facts to the Nullness Checker? For each, give one sentence of explanation, and a code snippet as an example.

(a) ***Supply a non-null type in the declaration of `m`. For example:***
`Map<String,Date> m`, but* not *`Map<String,/*@Nullable*/ Date> m`
It's redundant, but acceptable, to write `Map<String,/*@NonNull*/ Date> m`

(b) ***Write `@KeyFor` in the declaration of `k`.***
For example: `/*@KeyFor("m")*/ String k;`

An answer that changes the definition of the `Map` ADT is incorrect, because it is not under your control to change. A common mistake was trying to place a type annotation on an expression rather than on a type, in a declaration. It's also wrong to say `if (m.containsKey(k)) { m.get(k); }` because we are trying to argue that `m.get(k)` does return a non-null value, not to change the code to remove the call if it might return null.

4 Specifications

26. (12 points) Suppose that you want to change an ADT. In one sentence, under what circumstances is it possible to convert a specification field into a derived specification field (without otherwise changing its meaning)?

    If the derived specification is redundant: that is, it contains no information that is not already available in one or more other specification fields.

    In one sentence each, give a benefit and a drawback of making this change.

    Benefit: ***Each procedure specification becomes shorter, because it does not need to state, in its modifies and effects clause, how the method affects the derived specification field. The abstraction function becomes shorter, for the same reason.***

    Drawback: ***Information about the effect of a method is distributed among different parts of your program — in its own specification and in the class's documentation where the derived specification field is documented — which may make it easier to overlook some of the effects.***

    An incorrect answer is anything about the implementation or the representation invariant, which are unrelated to specification fields. Another incorrect answer is anything having to do with client use of the abstraction. A specification field (whether derived or not) is not necessarily available to the client via a method call nor supplied by the client as an argument to the constructor.

27. (12 points) Suppose that you have the following code.

```
class A {
  /**
   * @requires Req1
   * @modifies M1
   * @throws T1
   * @effects E1
   * @returns Ret1
   */
  int m(int arg1, int arg2) { ... }
}

class B extends A {
  /**
   * @requires Req2
   * @modifies M2
   * @throws T2
   * @effects E2
   * @returns Ret2
   */
  int m(int arg1, int arg2) { ... }
}
```

For each clause, write the property that must hold in order for `B` to be a true subtype of `A`. Your solution should not use the terms "stronger" or "weaker".

`requires`: **Req1** ***logically implies*** **Req2**, ***or*** $Req1 \Rightarrow Req2$

`modifies`: **M2** ***is a subset of*** **M1**
The modifies clause lists the references that **might** ***be modified. There is no guarantee that they will be modified, nor — in the absence of an effects clause — how they are modified.***

`throws`: **T1** ***and*** **T2** ***are the same, for method arguments that satisfy*** **Req1**

`effects`: **E2** ***logically implies*** **E1**, ***for method arguments that satisfy*** **Req1**, ***or*** $Req1 \Rightarrow Req2$

`returns`: **Ret2** ***logically implies*** **Ret1**, ***for method arguments that satisfy*** **Req1**

Your answer needed to use correct types. For example, the **`modifies`** ***clause is a set, so logical implication such as*** $M1 \Rightarrow M2$ ***does not make sense.***

Similarly, the **`effects`** ***clause is a logical formula, so using set inclusion such as*** $E2 \subseteq M1$ ***does not make sense. You could imagine the*** **`effects`** ***clause as a list of formulas, in which case you could apply subset to them, but that would be a wrong answer since*** $E2$ ***can change existing formulas, not just add new ones.***