

Software Testing, Quality Assurance & Maintenance (ECE453/CS447/CS647/SE465): Final

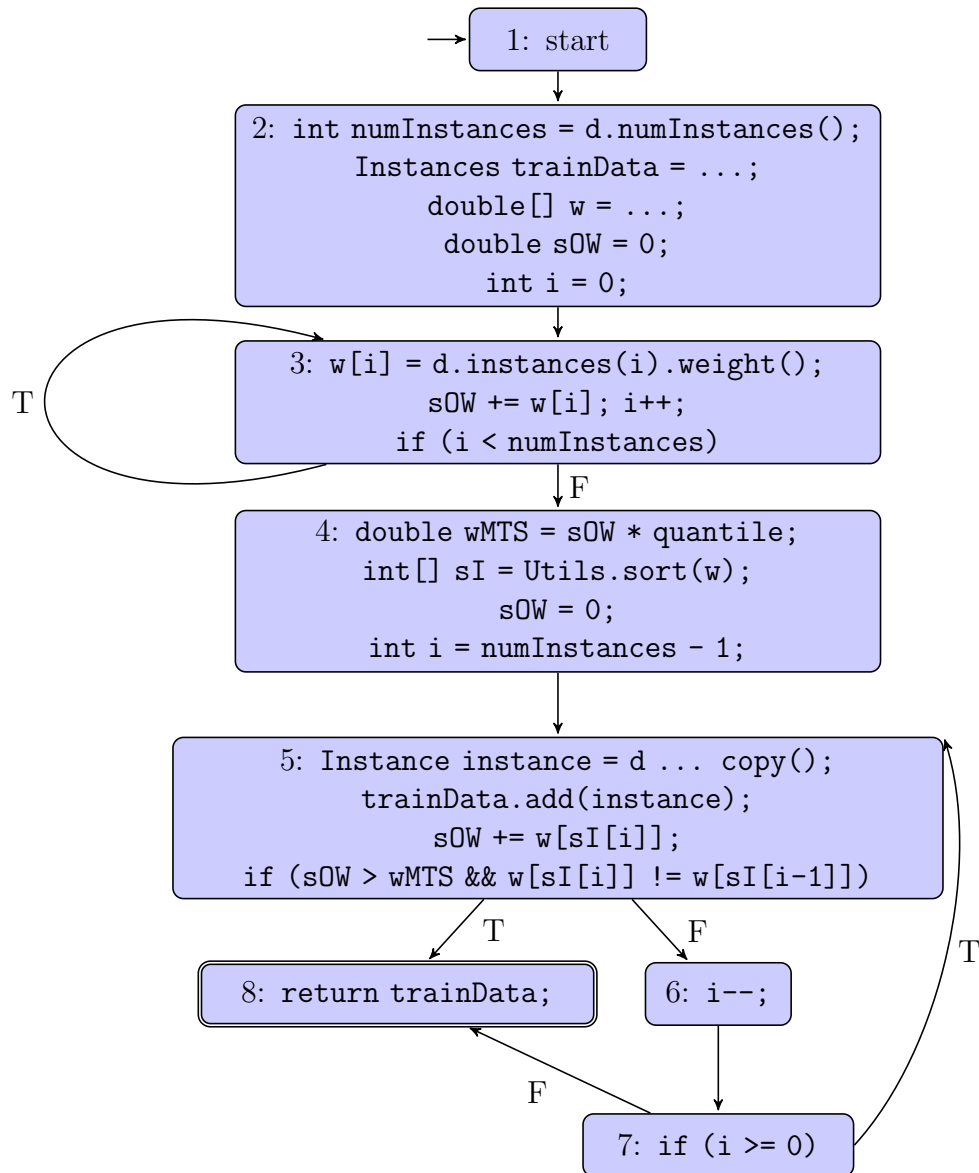
April 20, 2010

Question 1: Short Answer

1. automated; fast-to-execute; comprehensive; up-to-date.
2. false: a test case may handle multiple TRs.
3. preventing dataloss bugs.
4. long-term reliability of the system.
5. linked lists.
6. software is conceptually infinite-state.
7. there is no way to partially close one sub-bug.
8. one-line summary.
9. organizing testing.
10. false: the test suite satisfying A might not actually be any good and not include the actual important tests from the test suite satisfying B .

Question 2: Graph Coverage

CFG: 4 points.



Abbreviations:

d data
w weights
sOW sumOfWeights
wMTS weightMassToSelect
sI sortedIndices

Table of defs and uses (4 points):

Block	Defs	Uses
1	d, quantile	
2	numInstances, trainData, w, sOW, i	d, numInstances
3	w, sOW, i	d, w, i, numInstances, sOW
4	wMTS, sI, sOW, i	sOW, quantile, w, numInstances
5	instance, sOW	d, sI, instance, trainData, w, sOW, wMTS, i
6	i	i
7		i
8		trainData

Test requirements (6 points):

- (implicit) def of **d** at 1: (1)–(2) or (1)–(3) or (1)–(5)
- (implicit) def of **quantile** at 1: (1)–(4)
- def of **numInstances** at 2: (2)–(2) or (2)–(3) or (2)–(4) [so (2)–(2) is going to be the only TR that you really need]
- def of **trainData** at 2: (2)–(5) or (2)–(8) [which is unavoidable on any test path]
- def of **w** at 2: (2)–(3) [note that redef at (3) doesn't kill] or (2)–(4) or (2)–(5)
- def of **sOW** at 2: (2)–(3)
- def of **i** at 2: (2)–(3)
- def of **w** at 3: (3)–(3) or (3)–(4) or (3)–(5)
- def of **sOW** at 3: (3)–(3)
- def of **i** at 3: (3)–(3)
- def of **wMTS** at 4: (4)–(5)
- def of **sI** at 4: (4)–(5)
- def of **sOW** at 4: (4)–(5)
- def of **i** at 4: (4)–(5) or (4)–(6) or (4)–(7)
- def of **instance** at 5: (5)–(5)
- def of **sOW** at 5: (5)–(5)
- def of **i** at 6: (6)–(7), (6)–(5)

There are a lot of test requirements, but you got full marks if you seemed to understand how to generate test requirements for ADC. In particular, you needed to understand that the array assignment at 3 doesn't kill the def of **weights** at 2, and you should have included most of the other test requirements, including at least some (n) – (n) requirements and the implicit requirements.

Satisfying test requirements (6 points): I explicitly did not ask for any specific quality for the test cases, so you could've just provided a test case like `selectWeightQuantile(new Instances(), 0.0)` and that would take care of the implicit definition of `quantile` at (1), used at (2), for instance. You could also state the test requirement for `w`, (2)–(3), and provide a nontrivial input:

```
Instances inst = new Instances();
inst.add(new Instance(4.0)); inst.add(new Instance(5.0));
selectWeightQuantile(inst, 0.5);
```

which would meet the test requirement on `w` (2)–(3) as well as the test requirement on `i` going from (4) to (5). In general, you had to specify which test requirement you satisfied with your test cases.

Question 3: Logic Coverage (18 points)

(4 points) There are only 3 predicates here: (1) `i < numInstances`; (2) `i >= 0` and (3) `(sOW > wMTS) && w[sI[i]] != w[sI[i-1]]`. Let's assign short names to the clauses in (3): say (3a): `sOW > wMTS` and (3b): `w[sI[i]] != w[sI[i-1]]`. (1 point each for (1) and (2)).

(6 points) The test requirements for predicates (1) and (2) are that they are T and F (1 point each); the test requirements for (3), worth 4 points, are more complicated. Choosing (3a) to be major, we need (3b) to be T for (3a) to determine (3); similarly, for (3b) major, we need (3a) T. We get the TRs:

Number	Major	3a	3b
(i)	3a	T	T
(ii)	3a	F	T
(i)	3b	T	T
(iii)	3b	T	F

where the TT case overlaps for both (3a) and (3b) being major.

(8 points) The test suites for (1) and (2) were worth 1 point each, while the test suite for (3) was worth 6 points.

Any test case with a nonempty `data` array would meet the test requirements for (1) and (2), since such a test case would execute the loop body at least once and eventually exit the loop. The above example, for instance, works:

```
Instances inst = new Instances();
inst.add(new Instance(4.0)); inst.add(new Instance(5.0));
selectWeightQuantile(inst, 0.5);
```

The test case for (3) is more difficult, and required you to reason about the method's behaviour. First, case (i): we can try

```
Instances inst = new Instances();
inst.add(new Instance(1.0)); inst.add(new Instance(2.0));
selectWeightQuantile(inst, 0.5);
```

Then $wMTS$ is 1.5. In the first iteration ($i = 1$), we have $sOW = 2.0$, $w[1] = 2.0$ and $w[0] = 1.0$, so that (3a) is T and (3b) is T. This satisfies test requirement (i).

We notice that further iterations won't help with our test requirements, so we try different inputs. In particular, we need $sOW > wMTS$ to be false to satisfy test requirement (ii). That is, on iteration 1, the last (biggest) weight must not exceed sum times quartile.

```
Instances inst = new Instances();
inst.add(new Instance(1.0)); inst.add(new Instance(2.0));
selectWeightQuantile(inst, 1.0);
```

We have $wMTS = 3.0$. When $i = 1$, we have $sOW = 3.0$, which is equal to (hence not greater than) $wMTS$, making (3a) false and hence satisfying (ii).

Finally, we'll need a case with two identical instances to make (3b) false.

```
Instances inst = new Instances();
inst.add(new Instance(1.0)); inst.add(new Instance(1.0));
selectWeightQuantile(inst, 0.25);
```

Now $wMTS = 0.5$. Consider again the first iteration, $i = 1$. Here $sOW = 1.0$, so (3a) is T and (3b) is F, meeting test requirement (iii).

Question 4: Comparing ADUPC and PPC (2 points)

See midterm solutions.

Question 5: Mutation (20 points)

Many mutations are possible, and I wasn't looking for anything in particular. Here are two possibilities: 1) change 0 to 1 in the loop condition on line 6; a killing test case would then be the array $[5, 2, 3]$, which would give $[5, 2, 3]$ on the mutant and $[2, 3, 5]$ on the original; 2) replace $a[j]$ by $a[i]$ on line 7 and use the test case $[4, 2]$ (or the same one from before, actually,) which would give $[2, 4]$ on the original and $[4, 2]$ on the mutant.

Question 6: Input-Space Testing (10 points)

Many partitionings are possible. Using fewer blocks makes the base choice coverage part of the question easier. So, for instance, you could choose (1a) zero-point polygons, (1b) one-point polygons, and (1c) more-than-one-point polygons; (2a) enclosed polygons where

first point == last point and (2b) non-enclosed polygons; (3a) polygons with right angles and (3b) polygons with no right angles.

Base choice coverage here is a bit different from the examples we saw in class, since the three partitionings split one input space three ways. For my example, you could choose the base choice block to be (1c) more-than-one-point polygons which are (2a) enclosed and (3b) contain no right angles. One input which belongs to the base choice block is the scalene triangle $\langle(0, 0), (1, 2), (3, 3), (0, 0)\rangle$.

Base choice coverage then involves picking a different block for each characteristic.

1. (1b) the one-point polygon $\langle(0, 0)\rangle$, which is enclosed and contains no right angles, plus (1a) the zero-point polygon $\langle\rangle$, which we can declare enclosed, with no right angles;
2. (2b) the non-enclosed polygon $\langle(0, 0), (1, 2), (3, 3)\rangle$ which contains more than one point and has no right angles;
3. (3a) the right triangle $\langle(0, 0), (1, 0), (0, 1), (0, 0)\rangle$, which contains a right angle, is enclosed, and contains more than one point.

Question 7: Stub/mock Objects (20 points, 5 per class)

A. This is a struct-like class, so we wouldn't expect that a mock object would be useful in this case. Since **A** has no other dependencies, it would be sensible to use **A** itself when testing **X**. I don't see any challenges in using an actual **A** object.

B. I expected you to notice that class **B** interacts with class **H**; it is supposed to call certain methods on **H** and receive callbacks. I'd therefore use a mock object to test **B**. There could be some complications with the call `h.foo(this)`, depending on what the method `foo()` might do with the **B** object; we would have to replicate this behaviour in the mock object. Stubs and fakes wouldn't do well here.

C. An implementation of **C** would presumably need some key/value pairs programmed in. This might be a case in which a fake or stub object might be most appropriate; such an object might hardcode the key/value pairs. I haven't given you the usage information for **C**, but from the interface, it doesn't look like you'd need to record the calls to **C** nor its return values, so a mock might be inconvenient. Note that **C** appears to have no expectations.

D. If you tested `MegaMek`, this kind of class should be familiar to you. Class **D** also uses a database. One option (which is beyond the scope of this question) is to rewrite class **D** to be more testable.

Short of that, you could create a mock **D** implementation which calls the right methods on the dependencies, but that's not easy. Stub or fake objects could work as well. It would depend on how class **X** actually uses **D**. You could also use \widehat{D} itself with mock dependencies.