Soft. Verification & Validation
Exam One
June 9, 2016

Material for the exam: *You need only a pen and your student ID card. Bags and backpacks, including mobile phones, should be left by the blackboard. Violations to this norm will invalidate your test.*
On exam writing: *Start each new group on a new page.*
Duration: *Two hours and thirty minutes.*

Group 1. [Input Space Partitioning. 3 points]

Consider the following class describing a map in which the keys are small integers and the values are arbitrary objects, due to Koffman and Wolgang.

```
public class IntKeyMap {
  /**
    * Construct a map, initially empty, whose keys will be
    * integers in the range 0 through m-1.
    */
  public IntKeyMap(int m) {...}
  /**
    * Return the value in the entry with key in this map, or
    * null if there is no such entry.
    */
  public Object get(int key) {...}
  /**
    * Remove the entry with key (if any) from this map. Return
    * the value in that entry, or null if there was no such entry.
    */
  public Object remove(int key) {...}
  /**
    * Add the entry <key, val> to this map, replacing any
    * existing entry with key. Return the value in that entry,
    * or null if there was no such entry.
    */
  public Object put(int key, Object val) {...}
}
```

a) Identify at least three characteristics and respective blocks for method remove. Avoid binary characteristics, that is, strive to find characteristics with at least three blocks.

b) Identify test requirements for criterion Base Choice Coverage (BCC).

c) Characterise tests for the BCC requirements in the form of sequences of calls to methods in class IntMapKey. Are there infeasible tests? Justify.

d) Describe the combination strategy criteria called Each Choice Coverage (ECC). Does it subsume BCC? What about the other way round? Now that you have developed tests for BCC, is it worth considering ECC?

Group 2. [Logic-based Test Coverage. 3.5 points]

Consider the following method from class String in `openjdk 6-b14`, shared by String and StringBuffer to do searches.

```
 1  /**
 2   * @param source the characters being searched.
 3   * @param sourceOffset offset of the source string.
 4   * @param sourceCount count of the source string.
 5   * @param target the characters being searched for.
 6   * @param targetOffset offset of the target string.
 7   * @param targetCount count of the target string.
 8   * @param fromIndex the index to begin searching from.
 9   */
10  static int indexOf(char[] source, int sourceOffset, int sourceCount,
11      char[] target, int targetOffset, int targetCount, int fromIndex) {
12    if (fromIndex >= sourceCount)
13      return targetCount == 0 ? sourceCount : -1;
14    if (fromIndex < 0)
15      fromIndex = 0;
16    if (targetCount == 0)
17      return fromIndex;
18    char first = target[targetOffset];
19    int max = sourceOffset + (sourceCount - targetCount);
20    for (int i = sourceOffset + fromIndex; i <= max; i++) {
21      /* Look for first character. */
22      if (source[i] != first) {
23        while (++i <= max && source[i] != first)
24          ;
25        /* Found first character, now look at the rest of v2 */
26        if (i <= max) {
27          int j = i + 1;
28          int end = j + targetCount - 1;
29          for (int k = targetOffset + 1; j < end
30                && source[j] == target[k]; j++, k++)
31            ;
32          if (j == end)
33            /* Found whole string. */
34            return i - sourceOffset;
35        }
36      }
37    }
38    return -1;
39  }
```

a) Identify the predicates and clauses in method indexOf.

b) Identify the test requirements for Clause Coverage, TR(CC).

c) Identify the reachability predicates. Build a table that relates predicate p to the reachability predicate $r(p)$ of p.

d) Characterise tests that cover TR(CC), for the clauses in predicates up to (and including) line 22. You do not have to code the tests.

e) Does Predicate Coverage, PC, subsume CC? Does CC subsume PC? Justify. Give examples from the source code of method indexOf in case of negative answer(s).

Group 3. [Graph Coverage. 3.5 points]

Consider the following method that gives the difference between the largest and the smallest integer in an array.

```
 1  public static int range (int[] v) {
 2    if (v == null)
 3      throw new NullPointerException ();
 4    if (v.length == 0)
 5      return 0;
 6    int min = v[0];
 7    int max = v[0];
 8    for (int i = 1; i < v.length; i++)
 9      if (v[i] > max)
10        max = v[i];
11      else if (v[i] < min)
12        min = v[i];
13    return max - min;
14  }
```

a) Draw the control flow graph for the method.

b) For each node and edge, identify all definitions and all uses.

c) Exhibit a path from a definition of variable max to a use of the same variable that is *not* def-clear with respect to max.

d) Describe all du-paths.

e) For variables max and i only, identify a set of du-paths that satisfy All-Defs Coverage (ADC) but not All-Uses Coverage (AUC). Justify your choice.

f) Characterise tests that cover all du-paths identified in **e)**. You do not have to code the tests.

Group 4. [Program Mutation Testing. 3 points]

Referring to method range above, consider the following source code mutations.

#	Line	Was	Becomes
1	4	v.length == 0	v.length != 0
2	8	i < v.length	i <= v.length
3	9	v[i] > max	v[i] >= max
4	13	max − min	max + min

a) For each mutant induced by the mutations in the table, identify a test that kills the mutant, if possible. Justify your choices.

b) Are there functionally equivalent mutants? Which? Justify your answer.

c) For each mutant identified in **a)**, show an additional test that reaches the mutation but does not lead to an error state, if possible. Justify your choice.

d) For each mutant identified in **a)**, show an additional test that leads to an error state but not to a failure (a weak kill), if possible.

Group 5. [Test Patterns. 1 points]

Discuss *three* of the following Principles of Test Automation. For each principle of your choice describe the principle, its objectives, and its reason.

- Write the Tests First;
- Design for Testability;
- Communicate Intent;
- Don't Modify the SUT;
- Keep Tests Independent;
- Verify One Condition per Test.