CS 3250: Software Testing (Fall 2025)

Activity: Graph coverage for design element

(no submission)

Purpose: Understand and apply graph coverage to test various parts of the software design; get ready for assignment 4; prepare for the final exam

You may make a copy of a worksheet and complete this activity, or type your answers in any text editor.
You may work alone or with at most two other students in this course.

Consider the `trashgraph` class

```
// Modified from
// https://www.albany.edu/faculty/offutt/softwaretest/java/TrashAndTakeOut.java
public class trashgraph
{
   public static void trash(int x)
   {
      int m, n;
      m = 0;
      if (x > 0)
         m = 4;
      if (x > 5)
         n = 3 * m;
      else
         n = 4 * m;
      int o = takeOut(m, n);
      System.out.println("o is: " + o);
   }

   public static int takeOut(int a, int b)
   {
      int d, e;
      d = 42 * a;
      if (a > 0)
         e = 2 * b + d;
      else
         e = b + d;
      return(e);
   }
}
```

```
// Modified from https://cs.gmu.edu/~offutt/softwaretest/java/TrashAndTakeOut.java

1.   public class trashgraph
2.   {
3.      public static void trash(int x)
4.      {
5.         int m, n;
6.         m = 0;
7.         if (x > 0)
8.            m = 4;
9.         if (x > 5)
10.           n = 3 * m;
11.        else
12.           n = 4 * m;
13.        int o = takeOut(m, n);
14.        System.out.println("o is: " + o);
15.     }
16.
17.     public static int takeOut(int a, int b)
18.     {
19.        int d, e;
20.        d = 42 * a;
21.        if (a > 0)
22.           e = 2 * b + d;
23.        else
24.           e = b + d;
25.        return(e);
26.     }
27.  }
```

call site
last def
first use

note: since we are considering coupling DU-pairs, def and use of d are excluded

1. Locate a call site between `trash()` and `takeOut()` methods. You may circle directly in the source code.

   Line 13, trash() → takeOut()

2. Write down **all variables** that are shared or passed between `trash()` and `takeOut()` methods.

   m and n in trash() → a and b in takeOut()
   e in takeOut() → o is trash()

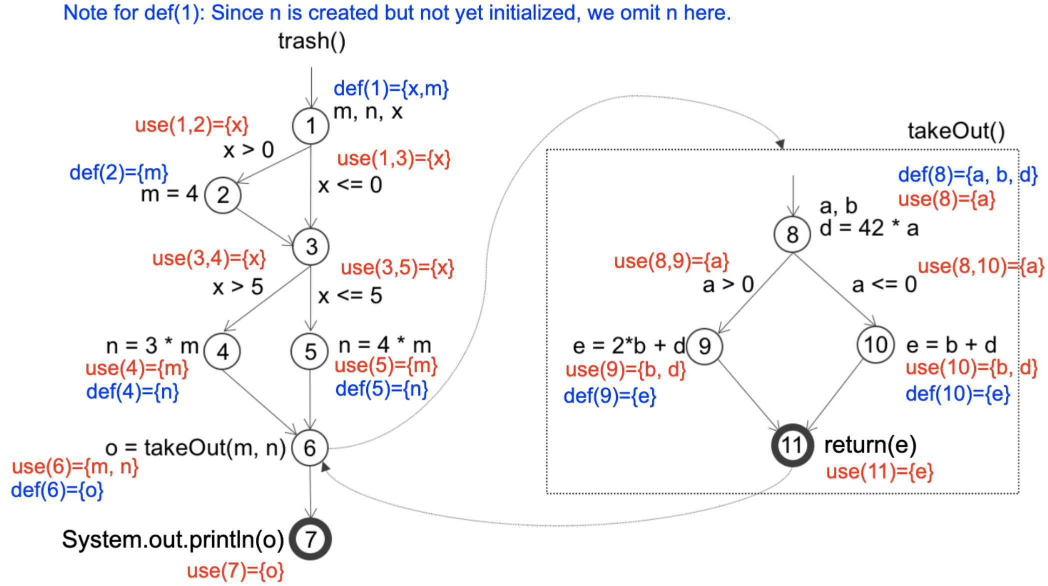
3. Find all coupling **last-defs** and **first-uses**. Circle directly in the source code. Write down whether that circle is for last-def or first-use.

   Last defs:
   - m, lines 6 and 8
   - n, lines 10 and 12
   - e, lines 22 and 24

   First uses:
   - a, line 20
   - b, lines 22 and 24
   - o, line 14

4. Draw a graph representation of the `trash()` and `takeOut()` method. Be sure to include coupling between the caller and the callee. (You may draw the graph by hand, take a screenshot of your graph, and embed it in your write-up)

Note for def(1): Since n is created but not yet initialized, we omit n here.



5. Find all the **coupling DU-pairs**.

Write them down as pairs of triplets. Each triplet is a method name, variable name, and node. The pair has the **last-def** on the left and a **first-use** on the right. For example:

    (method1(), var1, node1) — (method2(), var2, node2)

where last-def of a variable *var1* is in method *method1* at node *node1* and first-use of a variable *var1* appears as a variable *var2* in method *method2* at node *node2*.

Note: in general, if a graph is unavailable or has not been created, each triplet can be written as

    (method1(), var1, line1) — (method2(), var2, line2)

where last-def of a variable *var1* is in method *method1* at line number *line1* and first-use of a variable *var1* appears as a variable *var2* in method *method2* at line number *line2*.

If we write them as *(method1(), var1, line1) — (method2(), var2, line2)*

1. (trash(), m, 6) → (takeOut(), a, 20)
2. (trash(), m, 8) → (takeOut(), a, 20)
3. (trash(), n, 10) → (takeOut(), b, 22)
4. (trash(), n, 10) → (takeOut(), b, 24)
5. (trash(), n, 12) → (takeOut(), b, 22)
6. (trash(), n, 12) → (takeOut(), b, 24)
7. (takeOut(), e, 22) → (trash(), o, 14)
8. (takeOut(), e, 24) → (trash(), o, 14)

If we write them as *(method1(), var1, node1) — (method2(), var2, node2)*

1. (trash(), m, 1) → (takeOut(), a, 8)
2. (trash(), m, 2) → (takeOut(), a, 8)
3. (trash(), n, 4) → (takeOut(), b, 9)
4. (trash(), n, 4) → (takeOut(), b, 10)
5. (trash(), n, 5) → (takeOut(), b, 9)
6. (trash(), n, 5) → (takeOut(), b, 10)
7. (takeOut(), e, 9) → (trash(), o, 7)
8. (takeOut(), e, 10) → (trash(), o, 7)

6. List test requirements (i.e., Coupling DU-paths for each variable) that satisfy **All-Coupling-Defs** coverage

Assume we consider `(method1(), var1, node1) — (method2(), var2, node2)`

For m, need coupling du-pair 1, 2
For n, need (either 3 or 4) and (either 5 or 6); suppose we choose 3 and 5
For e, need 7, 8
Thus, du-pairs = 1,2,3,5,7,8
The coupling du-paths are used as test requirements; list du-paths for each du-pair.

du-pairs	**du-paths**
1 (trash(), m, 1) → (takeOut(), a, 8)	[1,3,4,6,8], (infeasible) [1,3,5,6,8] *suppose we pick [1,3,5,6,8]*
2 (trash(), m, 2) → (takeOut(), a, 8)	[2,3,4,6,8], [2,3,5,6,8] *suppose we pick [2,3,4,6,8]*
3 (trash(), n, 4) → (takeOut(), b, 9)	[4,6,8,9]
5 (trash(), n, 5) → (takeOut(), b, 9)	[5,6,8,9]
7 (takeOut(), e, 9) → (trash(), o, 7)	[9,11,6,7]
8 (takeOut(), e, 10) → (trash(), o, 7)	[10,11,6,7]

TR = { [1,3,5,6,8], [2,3,4,6,8], [4,6,8,9], [5,6,8,9], [9,11,6,7], [10,11,6,7] }

Assume we don't fix the infeasible DU-paths.

Note: def-clear paths

7. List test requirements (i.e., Coupling DU-paths for each variable) that satisfy **All-Coupling-Uses** coverage

Assume we consider `(method1(), var1, node1) — (method2(), var2, node2)`

For m, need coupling du-pair 1, 2
For n, need 3,4, 5,6
For e, need 7, 8
Thus, du-pairs = 1,2,3,4,5,6,7,8
The coupling du-paths are used as test requirements; list du-paths for each du-pair.

du-pairs	du-paths
1 (trash(), m, 1) → (takeOut(), a, 8)	[1,3,4,6,8], (infeasible) [1,3,5,6,8] *suppose we pick [1,3,5,6,8]*
2 (trash(), m, 2) → (takeOut(), a, 8)	[2,3,4,6,8], [2,3,5,6,8] *suppose we pick [2,3,4,6,8]*
3 (trash(), n, 4) → (takeOut(), b, 9)	[4,6,8,9]
4 (trash(), n, 4) → (takeOut(), b, 10)	[4,6,8,10] (infeasible)
5 (trash(), n, 5) → (takeOut(), b, 9)	[5,6,8,9]
6 (trash(), n, 5) → (takeOut(), b, 10)	[5,6,8,10]
7 (takeOut(), e, 9) → (trash(), o, 7)	[9,11,6,7]
8 (takeOut(), e, 10) → (trash(), o, 7)	[10,11,6,7]

TR = { [1,3,5,6,8], [2,3,4,6,8], [4,6,8,9], ~~[4,6,8,10]~~, [5,6,8,9], [5,6,8,10], [9,11,6,7], [10,11,6,7] }

Assume we don't fix the infeasible DU-paths.

[4,6,8,10] is infeasible. Thus the coverage level will never reach 100%.

Note: def-clear paths

8. List test requirements (i.e., Coupling DU-paths for each variable) that satisfy **All-Coupling-DU-Paths** coverage

Assume we consider `(method1(), var1, node1) — (method2(), var2, node2)`

For m, need coupling du-pair 1, 2
For n, need 3,4, 5,6
For e, need 7, 8
Thus, du-pairs = 1,2,3,4,5,6,7,8
The coupling du-paths are used as test requirements; list du-paths for each du-pair.

du-pairs	**du-paths**
1 (trash(), m, 1) → (takeOut(), a, 8)	[1,3,4,6,8], *(infeasible)* [1,3,5,6,8], [1,2,3,5,6,8], *(**not def-clear**, m def at node 1 is redefined at node 2)* [1,2,3,4,6,8] *(**not def-clear,** m def at node 1 is redefined at node 2)*
2 (trash(), m, 2) → (takeOut(), a, 8)	[2,3,4,6,8], [2,3,5,6,8]
3 (trash(), n, 4) → (takeOut(), b, 9)	[4,6,8,9]
4 (trash(), n, 4) → (takeOut(), b, 10)	[4,6,8,10] (infeasible)
5 (trash(), n, 5) → (takeOut(), b, 9)	[5,6,8,9]
6 (trash(), n, 5) → (takeOut(), b, 10)	[5,6,8,10]
7 (takeOut(), e, 9) → (trash(), o, 7)	[9,11,6,7]
8 (takeOut(), e, 10) → (trash(), o, 7)	[10,11,6,7]

TR = { ~~[1,3,4,6,8]~~, [1,3,5,6,8], ~~[1,2,3,5,6,8]~~, ~~[1,2,3,4,6,8]~~, [2,3,4,6,8], [2,3,5,6,8], [4,6,8,9], ~~[4,6,8,10]~~, [5,6,8,9], [5,6,8,10], [9,11,6,7], [10,11,6,7] }

Assume we don't fix the infeasible DU-paths.

[1,3,4,6,8] and [4,6,8,10] are infeasible. Thus the coverage level will never reach 100%.

Note: def-clear paths

9. Provide test inputs that satisfy **All-Coupling-Uses** (note that trash() only has one input)

[refer to question 7]

du-pairs	du-paths
1 (trash(), m, 1) → (takeOut(), a, 8)	[1,3,4,6,8], (infeasible) [1,3,5,6,8] *suppose we pick [1,3,5,6,8]*
2 (trash(), m, 2) → (takeOut(), a, 8)	[2,3,4,6,8], [2,3,5,6,8] *suppose we pick [2,3,4,6,8]*
3 (trash(), n, 4) → (takeOut(), b, 9)	[4,6,8,9]
4 (trash(), n, 4) → (takeOut(), b, 10)	[4,6,8,10] (infeasible)
5 (trash(), n, 5) → (takeOut(), b, 9)	[5,6,8,9]
6 (trash(), n, 5) → (takeOut(), b, 10)	[5,6,8,10]
7 (takeOut(), e, 9) → (trash(), o, 7)	[9,11,6,7]
8 (takeOut(), e, 10) → (trash(), o, 7)	[10,11,6,7]

TR = { [1,3,5,6,8], [2,3,4,6,8], [4,6,8,9], ~~[4,6,8,10]~~, [5,6,8,9], [5,6,8,10], [9,11,6,7], [10,11,6,7] }

The trash() method takes only one input, which is X.

X = 0 satisfies 1, 6, 8
X = 1 satisfies 2, 5, 7
X = 6 satisfies 2, 3, 7

Cannot satisfy du-pair #4 (i.e., du-path [4,6,8,10]). This is because

- To reach node 4, x must be > 5.
- If `trash` has x > 5, (reach node 6) m = 4, and n = 12.
- Then (reach node 8) m becomes a in `takeOut`, which will have a > 0.
- Since a > 0, node 10 will never be reached.

Thus, def of n at node 4 (line 10) does not reach use at node 19 (line 24).

[optional]
For more hands-on experience, take the test inputs you provided to satisfy **All-Coupling-Uses**, construct a set of test cases, and then automate your test set (in JUnit) and run it against the `trashgraph` class.