

Introduction to Software Testing

Paul Ammann and Jeff Offutt

Solutions to Exercises

Student Version

September 19, 2011

Tell me and I may forget. Show me and I may remember. Involve me and I will understand
– Chinese proverb

Introductory Note

As readers will undoubtedly note, as of September 19, 2011, a few problems still do not have complete solutions. Paul (pammann@gmu.edu) and Jeff (offutt@gmu.edu) welcome your help with new solutions and corrections to our errors. In fact, we hope this will be viewed as a community resource and welcome any kind of feedback.

We distinguish between “student solutions” and “instructor only” for the convenience of both. Students can work homeworks then check their own answers. Instructors can assign homeworks with some confidence that students will do their own work instead of looking up the answer in the manual.

Exercises, Section 1.1

1. What are some of the factors that would help a development organization move from Beizer's testing level 2 (testing is to show errors) to testing level 4 (a mental discipline that increases quality)?

Instructor Solution Only

2. The following exercise is intended to encourage you to think of testing in a more rigorous way than you may be used to. The exercise also hints at the strong relationship between specification clarity, faults, and test cases.

- (a) Write a Java method with the signature
`public static Vector union (Vector a, Vector b)`
The method should return a Vector of objects that are in either of the two argument Vectors.

Instructor Solution Only

- (b) Upon reflection, you may discover a variety of defects and ambiguities in the given assignment. In other words, ample opportunities for faults exist. Identify as many possible faults as you can. (*Note: `Vector` is a Java Collection class. If you are using another language, interpret `Vector` as a list.*)

Instructor Solution Only

- (c) Create a set of test cases that you think would have a reasonable chance of revealing the faults you identified above. Document a rationale for each test in your test set. If possible, characterize all of your rationales in some concise summary. Run your tests against your implementation.

Instructor Solution Only

- (d) Rewrite the method signature to be precise enough to clarify the defects and ambiguities identified earlier. You might wish to illustrate your specification with examples drawn from your test cases.

Instructor Solution Only

Exercises, Section 1.2

1. For what do testers use automation? What are the limitations of automation?

Solution:

Automation can help in many areas, most often to relieve the tester from repetitive, mechanical tasks. Checking of testing criteria can be automated through instrumentation, which allows a higher level of testing to be performed. Automation will always run into undecidable problems, such as infeasible paths, test case generation, internal variables, etc. Automation cannot help validate output or make creative decisions.

2. How are faults and failures related to testing and debugging?

Solution:

Faults are problems in the code, failures are incorrect external events. Depending on which of Beizer's levels you are working in, testing is the process of trying to cause failures or to show that they occur at an acceptable rate. In contrast, debugging is a diagnostic process where, given a failure, an attempt is made to find the associated fault.

3. Below are four faulty programs. Each includes a test case that results in failure. Answer the following questions about each program.

```
public int findLast (int[] x, int y)
{
  //Effects: If x==null throw NullPointerException
  // else return the index of the last element
  // in x that equals y.
  // If no such element exists, return -1
  for (int i=x.length-1; i > 0; i--)
  {
    if (x[i] == y)
    {
      return i;
    }
  }
  return -1;
}
// test: x=[2, 3, 5]; y = 2
// Expected = 0
```

```
public static int lastZero (int[] x)
{
  //Effects: if x==null throw NullPointerException
  // else return the index of the LAST 0 in x.
  // Return -1 if 0 does not occur in x

  for (int i = 0; i < x.length; i++)
  {
    if (x[i] == 0)
    {
      return i;
    }
  }
  return -1;
}
// test: x=[0, 1, 0]
// Expected = 2
```

```
public int countPositive (int[] x)
{
  //Effects: If x==null throw NullPointerException
  // else return the number of
  // positive elements in x.
  int count = 0;
  for (int i=0; i < x.length; i++)
  {
    if (x[i] >= 0)
    {
      count++;
    }
  }
  return count;
}
// test: x=[-4, 2, 0, 2]
// Expected = 2
```

```
public static int oddOrPos(int[] x)
{
  //Effects: if x==null throw NullPointerException
  // else return the number of elements in x that
  // are either odd or positive (or both)
  int count = 0;
  for (int i = 0; i < x.length; i++)
  {
    if (x[i]%2 == 1 || x[i] > 0)
    {
      count++;
    }
  }
  return count;
}
// test: x=[-3, -2, 0, 1, 4]
// Expected = 3
```

- Identify the fault.
- If possible, identify a test case that does **not** execute the fault.
- If possible, identify a test case that executes the fault, but does **not** result in an error state.
- If possible identify a test case that results in an error, but **not** a failure. Hint: Don't forget about the program counter.
- For the given test case, identify the first error state. Be sure to describe the complete state.
- Fix the fault and verify that the given test now produces the expected output.

`findLast()` Instructor Solution Only

lastZero() Solution:

(a) *The for-loop should search high to low:*

```
for (int i=x.length-1; i >= 0; i--) {
```

(b) *All inputs execute the fault - even the null input.*

(c) *If the loop is not executed at all, there is no error. If the loop is executed only once, high-to-low and low-to-high evaluation are the same. Hence there is no error for length 0 or length 1 inputs.*

Input: $x = [3]$

Expected Output: -1

Actual Output: -1

Correction 7-Mar-2010: In the case where the array has exactly one element, the last value for variable `i` is -1 in the correct code, but 1 in the original code. Since variable `i` has the wrong value the state clearly meets the definition of an error state. This is a fairly subtle point; the loop predicate evaluates correctly to `false`, and the variable `i` immediately goes out of scope. Thanks to Yasmine Badr who relayed this point from an anonymous student.

(d) *There is an error anytime the loop is executed more than once, since the values of index `i` ascend instead of descend.*

Input: $x = [1, 0, 3]$

Expected Output: 1

Actual Output: 1

(e) *The first error state is when index `i` has the value 0 when it should have a value at the end of the array, namely `x.length-1`. Hence, the first error state is encountered immediately after the assignment to `i` in the for-statement if there is more than one value in `x`.*

Input: $x = [0, 1, 0]$

Expected Output: 2

Actual Output: 0

First Error State:

$x = [0, 1, 0]$

$i = 0$

PC = just after `i = 0`;

(f) *See (a)*

`countPositive()` **Instructor Solution Only**

oddOrPos() Solution:

(a) *The if-test needs to take account of negative values (positive odd numbers are taken care of by the second test):*

```
if (x[i]%2 == -1 || x[i] > 0)
```

(b) *x must be either null or empty. All other inputs result in the fault being executed. We give the empty case here.*

<i>Input:</i>	$x = []$
<i>Expected Output:</i>	0
<i>Actual Output:</i>	0

(c) *Any nonempty x with only non-negative elements works, because the first part of the compound if-test is not necessary unless the value is negative.*

<i>Input:</i>	$x = [1, 2, 3]$
<i>Expected Output:</i>	3
<i>Actual Output:</i>	3

(d) *For this particular program, every input that results in error also results in failure. The reason is that error states are not repairable by subsequent processing. If there is a negative value in x, all subsequent states (after processing the negative value) will be error states no matter what else is in x.*

(e) <i>Input:</i>	$x = [-3, -2, 0, 1, 4]$
<i>Expected Output:</i>	3
<i>Actual Output:</i>	2
<i>First Error State:</i>	
$x = [-3, -2, 0, 1, 4]$	
$i = 0;$	
$count = 0;$	
<i>PC = at end of if statement, instead of just before count++</i>	

(f) See (a)

Thanks to Jim Bowring for correcting this solution. Also thanks to Farida Sabry for pointing out that negative even integers are also possible in the solution to part (c).

Also, note that this solution depends on treating the PC as pointing to the entire predicate $x[i]\%2 == -1 \ || \ x[i] > 0$ rather than to the individual clauses in this predicate. If you choose to consider states where the PC is pointing to the individual clauses in the predicate, then you can indeed get an infection without a failure in part (d). The reason is that for an odd positive input the erroneous first clause, $x[i]\%== 1$, returns true. Hence the if short-circuit evaluation terminates, rather than evaluating the $x[i]>0$ clause, as the correct program would. Bottom line: It's tricky to analyzing errors where the the PC has the wrong value!

Exercises, Section 1.3

1. Suppose that coverage criterion C_1 subsumes coverage criterion C_2 . Further suppose that test set T_1 satisfies C_1 on program P and test set T_2 satisfies C_2 , also on P .

- (a) Does T_1 necessarily satisfy C_2 ? Explain.

Solution:

Yes. This follows directly from the definition of subsumption.

- (b) Does T_2 necessarily satisfy C_1 ? Explain.

Solution:

No. There is no reason to expect test requirements generated by C_1 to be satisfied by T_2 .

- (c) If P contains a fault, and T_2 reveals the fault, T_1 does **not** necessarily also reveal the fault. Explain.

Instructor Solution Only

2. How else could we compare test criteria besides subsumption?

Instructor Solution Only

Exercises, Section 2.1

1. Give the sets N , N_0 , N_f , and E for the graph in Figure 2.2.

Solution:

$$N = \{n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9\}$$

$$N_0 = \{n_0, n_1, n_2\}$$

$$N_f = \{n_7, n_8, n_9\}$$

$$E = \{(n_0, n_3), (n_0, n_4), (n_1, n_4), (n_2, n_5), (n_2, n_6), (n_3, n_7), (n_4, n_7), (n_4, n_8), (n_5, n_1), (n_5, n_9), (n_6, n_9), (n_8, n_5), \}$$

Thanks to Aya Salah for correcting this solution.

2. Give a path that is not a test path in Figure 2.2.

Solution:

Obviously, there are many possibilities. $[n_4, n_8, n_5, n_9]$ is not a test path because it does not start in an initial node. $[n_1, n_4, n_8, n_5]$ is not a test path because it does not end in a final node.

3. List all test paths in 2.2.

Solution:

There are an unbounded number of these. In particular, any path that visits the loop $[n_1, n_4, n_8, n_5]$ can be extended indefinitely. We list some of the shorter test paths below. $[n_0, n_3, n_7]$, $[n_0, n_4, n_7]$, $[n_0, n_4, n_8]$, $[n_0, n_4, n_8, n_5, n_1, n_4, n_8]$, $[n_0, n_4, n_8, n_5, n_9]$, $[n_1, n_4, n_8]$, $[n_1, n_4, n_8, n_5, n_1, n_4, n_8]$, $[n_1, n_4, n_8, n_5, n_9]$, $[n_2, n_5, n_9]$, $[n_2, n_5, n_1, n_4, n_8]$, $[n_2, n_5, n_1, n_4, n_8, n_5, n_1, n_4, n_8]$, $[n_2, n_6, n_9]$.

4. In Figure 2.5, find test case inputs such that the corresponding test path visits edge (n_1, n_3) .

Solution:

As noted in the figure, input $(a = 0, b = 1)$ works.

Exercises, Section 2.2.1

1. Redefine *Edge Coverage* in the standard way (see the discussion for *Node Coverage*).

Instructor Solution Only

2. Redefine *Complete Path Coverage* in the standard way (see the discussion for *Node Coverage*).

Instructor Solution Only

3. Subsumption has a significant weakness. Suppose criterion C_{strong} subsumes criterion C_{weak} and that test set T_{strong} satisfies C_{strong} and test set T_{weak} satisfies C_{weak} . It is not necessarily the case that T_{weak} is a subset of T_{strong} . It is also not necessarily the case that T_{strong} reveals a fault if T_{weak} reveals a fault. Explain these facts.

Instructor Solution Only

4. Answer questions a-d for the graph defined by the following sets:

- $N = \{1, 2, 3, 4\}$
- $N_0 = \{1\}$
- $N_f = \{4\}$
- $E = \{(1, 2), (2, 3), (3, 2), (2, 4)\}$

- (a) Draw the graph.

Solution:

See the graph tool at <http://www.cs.gmu.edu/~offutt/softwaretest/>

- (b) List test paths that achieve Node Coverage, but not Edge Coverage.

Solution:

For this program, this is not possible. All test paths must begin at node 1, visit node 2, and, eventually, end at node 4. Any test path that visits node 3 also visits both edge (2,3) and edge (3,2).

- (c) List test paths that achieve Edge Coverage, but not Edge Pair Coverage.

Solution:

$$T = \{[1, 2, 3, 2, 4]\}$$

Note that the edge pair [3, 2, 3] is not toured by the single test path given.

- (d) List test paths that achieve Edge Pair Coverage.

Solution:

$$T = \{[1, 2, 4], [1, 2, 3, 2, 3, 2, 4]\}$$

Thanks to Justin Donnelly for correcting this solution.

5. Answer questions a-g for the graph defined by the following sets:

- $N = \{1, 2, 3, 4, 5, 6, 7\}$
- $N_0 = \{1\}$
- $N_f = \{7\}$
- $E = \{(1, 2), (1, 7), (2, 3), (2, 4), (3, 2), (4, 5), (4, 6), (5, 6), (6, 1)\}$

Also consider the following (candidate) test paths:

- $t_0 = [1, 2, 4, 5, 6, 1, 7]$
- $t_1 = [1, 2, 3, 2, 4, 6, 1, 7]$

(a) Draw the graph.

Solution:

See the graph tool at <http://www.cs.gmu.edu/~offutt/softwaretest/>

(b) List the test requirements for Edge-Pair Coverage. (Hint: You should get 12 requirements of length 2).

Instructor Solution Only

(c) Does the given set of test paths satisfy Edge-Pair Coverage? If not, identify what is missing.

Instructor Solution Only

(d) Consider the simple path $[3, 2, 4, 5, 6]$ and test path $[1, 2, 3, 2, 4, 6, 1, 2, 4, 5, 6, 1, 7]$. Does the test path tour the simple path directly? With a sidetrip? If so, identify the sidetrip.

Instructor Solution Only

(e) List the test requirements for Node Coverage, Edge Coverage, and Prime Path Coverage on the graph.

Instructor Solution Only

(f) List test paths that achieve Node Coverage but not Edge Coverage on the graph.

Instructor Solution Only

(g) List test paths that achieve Edge Coverage but not Prime Path Coverage on the graph.

Instructor Solution Only

6. Answer questions a-c for the graph in Figure 2.2.

(a) Enumerate the test requirements for Node Coverage, Edge Coverage, and Prime Path Coverage on the graph.

Solution:

$$TR_{NC} = \{n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9\}$$

$$TR_{EC} = \{(n_0, n_3), (n_0, n_4), (n_1, n_4), (n_2, n_5), (n_2, n_6), (n_3, n_7), (n_4, n_7), (n_4, n_8), (n_5, n_1), (n_5, n_9), (n_6, n_9), (n_8, n_5)\}$$

$$TR_{PPC} = \{[n_0, n_3, n_7], [n_0, n_4, n_7], [n_0, n_4, n_8, n_5, n_1], [n_0, n_4, n_8, n_5, n_9], [n_1, n_4, n_8, n_5, n_1], [n_1, n_4, n_8, n_5, n_9], [n_2, n_5, n_1, n_4, n_7], [n_2, n_5, n_1, n_4, n_8], [n_2, n_5, n_9], [n_2, n_6, n_9], [n_4, n_8, n_5, n_1, n_4], [n_5, n_1, n_4, n_8, n_5], [n_8, n_5, n_1, n_4, n_7], [n_8, n_5, n_1, n_4, n_8]\}$$

(b) List test paths that achieve Node Coverage but not Edge Coverage on the graph.

Instructor Solution Only

(c) List test paths that achieve Edge Coverage but not Prime Path Coverage on the graph.

Instructor Solution Only

7. Answer questions a-d for the graph defined by the following sets:

- $N = \{0, 1, 2\}$
- $N_0 = \{0\}$
- $N_f = \{2\}$
- $E = \{(0, 1), (0, 2), (1, 0), (1, 2), (2, 0)\}$

Also consider the following (candidate) paths:

- $p_0 = [0, 1, 2, 0]$
- $p_1 = [0, 2, 0, 1, 2]$
- $p_2 = [0, 1, 2, 0, 1, 0, 2]$
- $p_3 = [1, 2, 0, 2]$
- $p_4 = [0, 1, 2, 1, 2]$

(a) Which of the listed paths are test paths? Explain the problem with any path that is not a test path.

Solution:

Answer: Only p_1 and p_2 are test paths. p_0 fails to terminate at a final node. p_3 fails to start at an initial node. p_4 includes an edge that does not exist in the graph.

(b) List the eight test requirements for Edge-Pair Coverage (only the length two subpaths).

Solution:

Answer: The edge pairs are:

$\{ [n_0, n_1, n_0], [n_0, n_1, n_2], [n_0, n_2, n_0], [n_1, n_0, n_1], [n_1, n_0, n_2], [n_1, n_2, n_0], [n_2, n_0, n_1], [n_2, n_0, n_2] \}$

(c) Does the set of **test** paths (part a) above satisfy Edge-Pair Coverage? If not, identify what is missing.

Solution:

Answer: No. Neither p_1 nor p_2 tours either of the following edge-pairs:

$\{ [n_1, n_0, n_1], [n_2, n_0, n_2] \}$

As discussed in (a), the remaining candidate paths are not test paths.

(d) Consider the prime path $[n_2, n_0, n_2]$ and path p_2 . Does p_2 tour the prime path directly? With a sidetrip?

Solution:

Answer: No, p_2 does not directly tour the prime path. However, p_2 does tour the prime path with the sidetrip $[n_0, n_1, n_0]$.

8. Design and implement a program that will compute all prime paths in a graph, then derive test paths to tour the prime paths. Although the user interface can be arbitrarily complicated, the simplest version will be to accept a graph as input by reading a list of nodes, initial nodes, final nodes, and edges.

Instructor Solution Only

Exercises, Section 2.2.3

1. Below are four graphs, each of which is defined by the sets of nodes, initial nodes, final nodes, edges, and defs and uses. Each graph also contains a collection of test paths. Answer the following questions about each graph.

<p>Graph I. $N = \{0, 1, 2, 3, 4, 5, 6, 7\}$ $N_0 = \{0\}$ $N_f = \{7\}$ $E = \{(0, 1), (1, 2), (1, 7), (2, 3), (2, 4), (3, 2), (4, 5), (4, 6), (5, 6), (6, 1)\}$ $def(0) = def(3) = use(5) = use(7) = \{x\}$ Test Paths: $t1 = [0, 1, 7]$ $t2 = [0, 1, 2, 4, 6, 1, 7]$ $t3 = [0, 1, 2, 4, 5, 6, 1, 7]$ $t4 = [0, 1, 2, 3, 2, 4, 6, 1, 7]$ $t5 = [0, 1, 2, 3, 2, 3, 2, 4, 5, 6, 1, 7]$ $t6 = [0, 1, 2, 3, 2, 4, 6, 1, 2, 4, 5, 6, 1, 7]$</p>	<p>Graph II. $N = \{1, 2, 3, 4, 5, 6\}$ $N_0 = \{1\}$ $N_f = \{6\}$ $E = \{(1, 2), (2, 3), (2, 6), (3, 4), (3, 5), (4, 5), (5, 2)\}$ $def(1) = def(3) = use(3) = use(6) = \{x\}$ // Assume the use of x in 3 precedes the def Test Paths: $t1 = [1, 2, 6]$ $t2 = [1, 2, 3, 4, 5, 2, 3, 5, 2, 6]$ $t3 = [1, 2, 3, 5, 2, 3, 4, 5, 2, 6]$ $t4 = [1, 2, 3, 5, 2, 6]$</p>
<p>Graph III. $N = \{1, 2, 3, 4, 5, 6\}$ $N_0 = \{1\}$ $N_f = \{6\}$ $E = \{(1, 2), (2, 3), (3, 4), (3, 5), (4, 5), (5, 2), (2, 6)\}$ $def(1) = def(4) = use(3) = use(5) = use(6) = \{x\}$ Test Paths: $t_1 = [1, 2, 3, 5, 2, 6]$ $t_2 = [1, 2, 3, 4, 5, 2, 6]$</p>	<p>Graph IV. $N = \{1, 2, 3, 4, 5, 6\}$ $N_0 = \{1\}$ $N_f = \{6\}$ $E = \{(1, 2), (2, 3), (2, 6), (3, 4), (3, 5), (4, 5), (5, 2)\}$ $def(1) = def(5) = use(5) = use(6) = \{x\}$ // Assume the use of x in 5 precedes the def Test Paths: $t1 = [1, 2, 6]$ $t2 = [1, 2, 3, 4, 5, 2, 3, 5, 2, 6]$ $t3 = [1, 2, 3, 5, 2, 3, 4, 5, 2, 6]$</p>

- Draw the graph.
- List all of the du-paths with respect to x . (Note: Include all du-paths, even those that are subpaths of some other du-path).
- For each test path, determine which du-paths that test path tours. For this part of the exercise, you should consider both direct touring and sidetrips. Hint: A table is a convenient format for describing this relationship.
- List a minimal test set that satisfies *all defs* coverage with respect to x . (Direct tours only.) Use the given test paths.
- List a minimal test set that satisfies *all uses* coverage with respect to x . (Direct tours only.) Use the given test paths.
- List a minimal test set that satisfies *all du-paths* coverage with respect to x . (Direct tours only.) Use the given test paths.

Solution:

Solution for Graph I:

- (a) See the graph tool at <http://www.cs.gmu.edu/~offutt/softwaretest/>
 (b) x has 5 du-paths, as enumerated below:

<i>i</i>	[0, 1, 7]
<i>ii</i>	[0, 1, 2, 4, 5]
<i>iii</i>	[3, 2, 4, 5]
<i>iv</i>	[3, 2, 4, 6, 1, 7]
<i>v</i>	[3, 2, 4, 5, 6, 1, 7]

- (c) The numbers in the table below correspond to the du-paths in the previous table. The table indicates whether each test path tours each du-path with or without a sidetrip.

	<i>direct</i>	<i>w/ sidetrip</i>
t_1	<i>i</i>	
t_2		<i>i</i>
t_3	<i>ii</i>	<i>i</i>
t_4	<i>iv</i>	
t_5	<i>iii, v</i>	
t_6		<i>iii, iv, v</i>

- (d) This question has multiple possible answers. Either t_1 or t_3 can be used to directly tour a path that satisfies all-defs for the def at node 0, and either t_4 or t_5 can be used to directly tour a path that satisfies all-defs for the def at node 3.

Possible answers: $\{t_1, t_4\}$ or $\{t_1, t_5\}$ or $\{t_3, t_4\}$ or $\{t_3, t_5\}$

- (e) This question only has one possible answer: $\{t_1, t_3, t_5\}$
 (f) This question only has one possible answer: $\{t_1, t_3, t_4, t_5\}$

Thanks to Matt Rutherford for correcting this solution.

Instructor Solution Only

Solution:

Solution for Graph III: Note that this exercise is the same as Graph II, except that the def/use sets are slightly different.

- (a) See the graph tool at <http://www.cs.gmu.edu/~offutt/softwaretest/>
 (b) x has 6 du-paths, as enumerated below:

i	$[1, 2, 3]$
ii	$[1, 2, 3, 5]$
iii	$[1, 2, 6]$
iv	$[4, 5]$
v	$[4, 5, 2, 3]$
vi	$[4, 5, 2, 6]$

- (c) The numbers in the table below correspond to the du-paths in the previous table. The table indicates whether each test path tours each du-path with or without a sidetrip.

	<i>direct</i>	<i>w/ sidetrip</i>
t_1	i, ii	iii
t_2	i, iv, vi	

Note that neither t_1 nor t_2 tours du-path (v), either directly or with a sidetrip. Also note that neither t_1 nor t_2 tours du-path (iii), except with a sidetrip.

- (d) This question has one possible answer: $\{t_2\}$.
 (e) For all-uses, all six du-paths must be toured. Since the given test set does not have a test path that directly tours either of du-paths (iii) or (v), this question is unsatisfiable. To directly tour the given du – paths, we would two additional test paths. An example all-uses adequate test set (direct touring) is: $\{t_1, t_2, [1, 2, 6], [1, 2, 3, 4, 5, 2, 3, 5, 2, 6]\}$.
 (f) For this exercise, all-du-paths coverage is the same as all-uses coverage. The reason is that there is only one du-path for each du-pair.

Thanks to Rama Kesavan for pointing out the error in this solution. (February 2011).

Instructor Solution Only

Exercises, Section 2.3

1. Use the following program fragment for questions a-e below.

```
w = x;           // node 1
if (m > 0)
{
    w++;         // node 2
}
else
{
    w=2*w;      // node 3
}
// node 4 (no executable statement)
if (y <= 10)
{
    x = 5*y;    // node 5
}
else
{
    x = 3*y+5; // node 6
}
z = w + x;     // node 7
```

- (a) Draw a control flow graph for this program fragment. Use the node numbers given above.

Instructor Solution Only

- (b) Which nodes have defs for variable w ?

Instructor Solution Only

- (c) Which nodes have uses for variable w ?

Instructor Solution Only

- (d) Are there any du-paths with respect to variable w from node 1 to node 7? If not, explain why not. If any exist, show one.

Instructor Solution Only

- (e) Enumerate all of the du-paths for variables w and x .

Instructor Solution Only

2. Select a commercial coverage tool of your choice. Note that some have free trial evaluations. Choose a tool, download it, and run it on some software. You can use one of the examples from this text, software from your work environment, or software available over the Web. Write up a short summary report of your experience with the tool. Be sure to include any problems installing or using the tool. The main grading criterion is that you actually collect some coverage data for a reasonable set of tests on some program.

Solution:

This question doesn't really have a textbook solution. The problem with links to specific tools is that the set of available tools is quite dynamic, and hence links go out of date with alarming regularity. Googling `java coverage` will bring back a large number of mostly current links, typically including links that catalog and summarize available tools. This is an excellent exercise for making the coverage theory in the text "real".

3. Consider the pattern matching example in Figure 2.21. Instrument the code so as to be able to produce the execution paths reported in the text for this example. That is, on a given test execution, your instrumentation program should compute and print the corresponding test path. Run the instrumented program on the test cases listed at the end of Section 2.3.

Solution:

Access `TestPatInstrument.java` at <http://www.cs.gmu.edu/~offutt/softwaretest/>.

4. Consider the pattern matching example in Figure 2.21. In particular, consider the final table of tests in Section 2.3. Consider the variable $iSub$. Number the (unique) test cases, starting at 1, from the top of the $iSub$ part of the table. For example, $(ab, c, -1)$, which appears twice in the $iSub$ portion of the table, should be labeled test t_4 .

Solution:

<i>Test Number</i>	<i>Test</i>
t_1	$(ab, ab, 0)$
t_2	$(ab, a, 0)$
t_3	$(ab, ac, -1)$
t_4	$(ab, c, -1)$
t_5	$(a, bc, -1)$
t_6	$(abc, bc, 1)$
t_7	$(ab, b, 1)$
t_8	$(abc, ba, -1)$
t_4	$(ab, c, -1)$
t_2	$(ab, a, 0)$

(a) Give a minimal test set that satisfies *all defs* coverage. Use the test cases given.

Instructor Solution Only

(b) Give a minimal test set that satisfies *all uses* coverage.

Instructor Solution Only

(c) Give a minimal test set that satisfies *all du-paths* coverage.

Instructor Solution Only

5. Again consider the pattern matching example in Figure 2.21. Instrument the code so as to produce the execution paths reported in the text for this example. That is, on a given test execution, your tool should compute and print the corresponding test path. Run the following three test cases and answer questions a-g below:

- *subject* = "brown owl" *pattern* = "wl" *expected output* = 7
- *subject* = "brown fox" *pattern* = "dog" *expected output* = -1
- *subject* = "fox" *pattern* = "brown" *expected output* = -1

(a) Find the actual path followed by each test case.

Solution:

t_1 : java TestPatInstrument "brown owl" wl

Pattern string begins at the character 7

Path is [1, 2, 3, 4, 10, 3, 4, 10, 3, 4, 10, 3, 4, 5, 6, 7, 8, 10, 3, 4, 10, 3, 4, 10, 3, 4, 10, 3, 4, 5, 6, 7, 9, 6, 10, 3, 11]

t_2 : java TestPatInstrument "brown fox" dog

Pattern string is not a substring of the subject string

Path is [1, 2, 3, 4, 10, 3, 4, 10, 3, 4, 10, 3, 4, 10, 3, 4, 10, 3, 4, 10, 3, 4, 10, 3, 11]

t_3 : java TestPatInstrument fox brown

Pattern string is not a substring of the subject string

Path is [1, 2, 3, 11]

- (b) For each path, give the du-paths that the path tours in the table at the end of Section 2.3. To reduce the scope of this exercise, consider only the following du-paths: $du(10, isSub)$, $du(2, isPat)$, $du(5, isPat)$, and $du(8, isPat)$.

Solution:

In the following table, we give information about both direct tours and tours with sidetrips. Specifically, ‘+’ means ‘tours directly’, ‘-’ means ‘does not tour’, ‘+!’ means ‘tours with def-clear sidetrip’, and ‘-!’ means ‘tours, but sidetrip has def’. Also, we only consider du-paths from Table 2.5. That is, we ignore the du-paths that are prefixes of other du-paths; see Table 2.3 for details. Note that except for the infeasible du-path [5, 6, 10, 3, 4], all du-paths can be toured directly by some test case; hence when we apply Best Effort touring later in this exercise, we demand a direct tour.

Source	du-path	t_1	t_2	t_3
$du(10, isSub)$	[10, 3, 4, 5, 6, 7, 9]	+	-	-
	[10, 3, 4, 5, 6, 10]	+!	-	-
	[10, 3, 4, 5, 6, 7, 8, 10]	+	-	-
	[10, 3, 4, 10]	+	+	-
	[10, 3, 11]	+	+	-
$du(2, isPat)$	[2, 3, 4]	+	+	-
	[2, 3, 11]	-!	+!	+
$du(5, isPat)$	[5, 6, 10, 3, 4]	-	-	-
	[5, 6, 10, 3, 11]	+!	-	-
$du(8, isPat)$	[8, 10, 3, 4]	+	-	-
	[8, 10, 3, 11]	-!	-	-

- (c) Explain why the du-path [5, 6, 10, 3, 4] cannot be toured by any test path.

Solution:

Since the value of `isPat` is set to `true` in node 5 and not reset on the path [6, 10, 3], the next node must be 11, not 4. Hence the du path [5, 6, 10, 3, 4] is infeasible.

- (d) Select tests from the table at the end of Section 2.3 to complete coverage of the (feasible) du-paths that are uncovered in question a.

Solution:

The given tests do not directly tour the following 3 (feasible) du-paths: [10, 3, 4, 5, 6, 10], [5, 6, 10, 3, 11], and [8, 10, 3, 11]. According to Table 2.5, tests (ab,b), (ab,a), and (ab,ac) respectively tour these du-paths directly. Note that Best Effort touring requires a direct tour of each feasible du-path.

- (e) From the tests above, find a minimal set of tests that achieves All-Defs Coverage with respect to the variable `isPat`.

Solution:

To start, it’s helpful to extend the table given in part (b) to include the 3 additional tests. Since direct tours are possible, we leave out the sidetrip information in this version of the table.

Source	du-path	t_1	t_2	t_3	(ab, b)	(ab, a)	(ab, ac)
$du(10, iSub)$	$[10, 3, 4, 5, 6, 7, 9]$	+					
	$[10, 3, 4, 5, 6, 10]$				+		
	$[10, 3, 4, 5, 6, 7, 8, 10]$	+					
	$[10, 3, 4, 10]$	+	+				
	$[10, 3, 11]$	+	+		+	+	+
$du(2, isPat)$	$[2, 3, 4]$	+	+		+	+	+
	$[2, 3, 11]$			+			
$du(5, isPat)$	$[5, 6, 10, 3, 4]$						
	$[5, 6, 10, 3, 11]$				+	+	
$du(8, isPat)$	$[8, 10, 3, 4]$	+					
	$[8, 10, 3, 11]$						+

For All-Defs (Best Effort Touring) with respect to `isPat`, we need to tour 3 du-paths, starting with, respectively, nodes 2, 5, and 8. Possible minimal sets are: $\{t_1, (ab, b)\}$, $\{t_1, (ab, a)\}$, $\{(ab, b), (ab, ac)\}$, or $\{(ab, a), (ab, ac)\}$.

- (f) From the tests above, find a minimal set of tests that achieves All-Uses Coverage with respect to the variable `isPat`.

Solution:

For All-Uses (Best Effort Touring) with respect to `isPat`, we need to tour the 5 feasible du-paths starting with nodes 2, 5, and 8. Tests t_1 , t_3 , and (ab, ac) are always needed since they are the only tests that tour $[8, 10, 3, 4]$, $[2, 3, 11]$, and $[8, 10, 3, 11]$, respectively. In addition, we need either (ab, b) or (ab, a) to tour $[2, 3, 4]$ and $[5, 6, 10, 3, 11]$. Hence there are two possible answers: $\{t_1, t_3, (ab, b), (ab, ac)\}$ or $\{t_1, t_3, (ab, a), (ab, ac)\}$.

- (g) Is there any difference between All-Uses Coverage and all du-paths coverage with respect to the variable `isPat` in the `pat` method?

Solution:

No. The test requirements are the same with respect to `isPat`. Note, however, that they are not the same with respect to `iSub`.

6. Use the following method `fmtRewrap()` for questions a-e below.

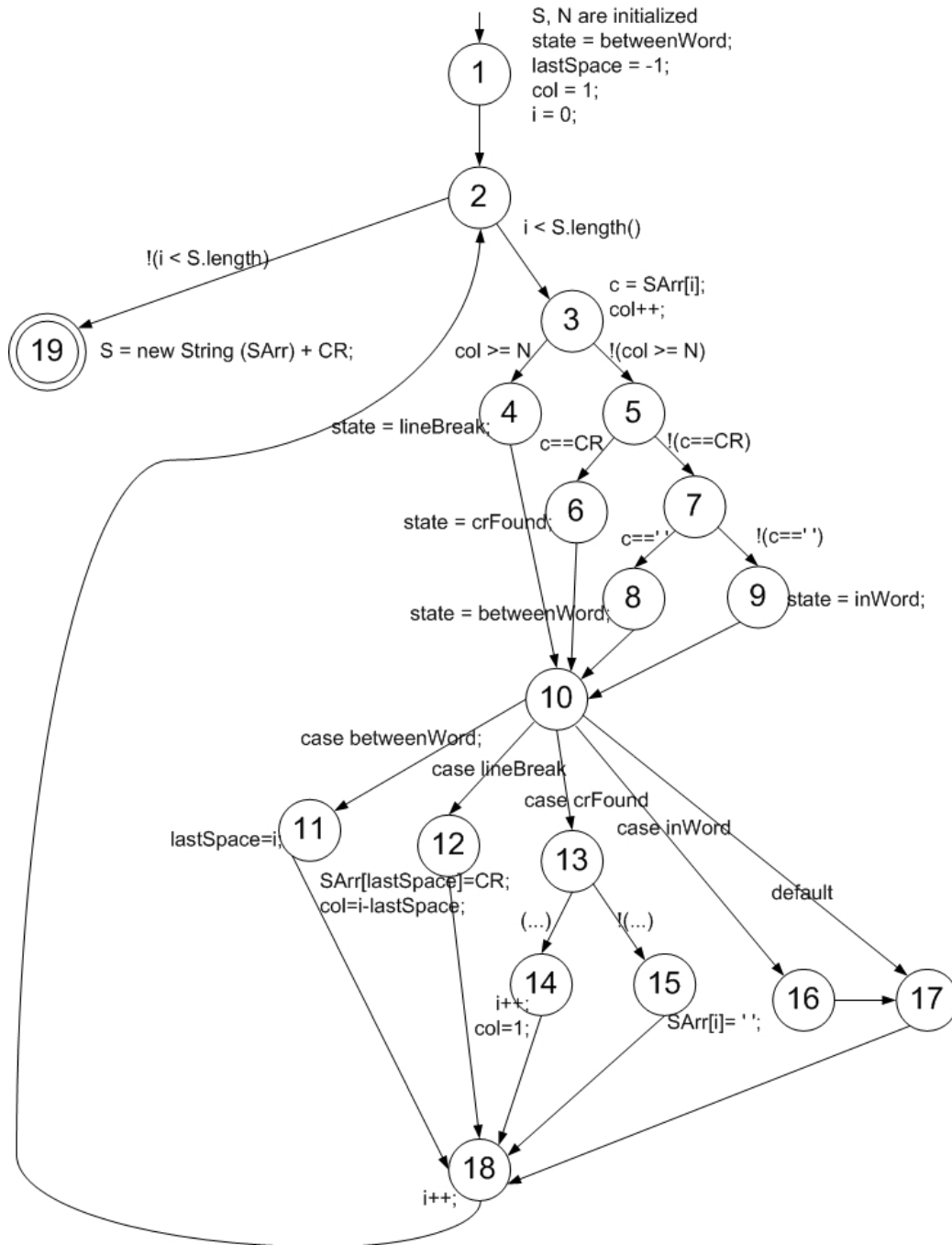
```

1. /** *****
2.  * Rewraps the string (Similar to the Unix fmt).
3.  * Given a string S, eliminate existing CRs and add CRs to the
4.  * closest spaces before column N. Two CRs in a row are considered to
5.  * be "hard CRs" and are left alone.
6.  * ***** */
7.
8. static final char CR = '\n';
9. static final int inWord = 0;
10. static final int betweenWord = 1;
11. static final int lineBreak = 2;
12. static final int crFound = 3;
13. static private String fmtRewrap (String S, int N)
14. {
15.     int state = betweenWord;
16.     int lastSpace = -1;
17.     int col = 1;
18.     int i = 0;
19.     char c;
20.
21.     char SArr [] = S.toCharArray();
22.     while (i < S.length())
23.     {
24.         c = SArr[i];
25.         col++;
26.         if (col >= N)
27.             state = lineBreak;
28.         else if (c == CR)
29.             state = crFound;
30.         else if (c == ' ')
31.             state = betweenWord;
32.         else
33.             state = inWord;
34.         switch (state)
35.         {
36.             case betweenWord:
37.                 lastSpace = i;
38.                 break;
39.
40.             case lineBreak:
41.                 SArr [lastSpace] = CR;
42.                 col = i-lastSpace;
43.                 break;
44.
45.             case crFound:
46.                 if (i+1 < S.length() && SArr[i+1] == CR)
47.                 {
48.                     i++; // Two CRs => hard return
49.                     col = 1;
50.                 }
51.                 else
52.                     SArr[i] = ' ';
53.                 break;
54.
55.             case inWord:
56.             default:
57.                 break;
58.         } // end switch
59.         i++;
60.     } // end while
61.     S = new String (SArr) + CR;
62.     return (S);
63. }

```

(a) Draw the control flow graph for the `fmtRerwrap()` method.

Solution:



Note that in the `switch` statement, there is a separate node for the case `inWord`, which, due to Java semantics, falls through directly to the `default` case.

(b) For `fmtRerwrap()`, find a test case such that the corresponding test path visits the edge that connects the beginning of the `while` statement to the `S = new String(SArr) + CR;` statement **without** going through the body of the while

loop.

Solution:

There is only one test that does this - the empty string S. It doesn't matter what N, the output line length, is. The resulting path is [1, 2, 19].

- (c) Enumerate the test requirements for Node Coverage, Edge Coverage, and Prime Path Coverage for the graph for `fmtRewrap()`.

Solution:

- *Node Coverage: { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 }*
- *Edge Coverage: { (1,2), (2,3), (2,19), (3,4), (3,5), (4,10), (5,6), (5,7), (6,10), (7,8), (7,9), (8,10), (9,10), (10,11), (10,12), (10,13), (10,16), (10,17), (11,18), (12,18), (13,14), (13,15), (14,18), (15,18), (16,17), (17,18), (18,2) }*
- *Prime Path Coverage: There are 403 prime paths for `fmt`; we don't list them here. See the graph tool at <http://www.cs.gmu.edu/~offutt/softwaretest/> for a complete enumeration*
To get a grasp on why there are so many prime paths, consider the "loop" prime paths starting at node 2. It is possible to take any of 4 paths through the `if else` statement, and then combine each of these with 6 paths through the case statement and return to node 2, yielding 24 prime paths. For node 3, the analysis is more complex: 4 choices combined with 6 choices to node 2, combined with 2 more choices (3 or 19), for a total of 48 prime paths.

- (d) List test paths that achieve Node Coverage but not Edge Coverage on the graph.

Solution:

Achieving Node Coverage without also achieving Edge Coverage is not possible for the given graph, since visiting every node implies visiting every edge. Test paths are available online. See the graph tool at <http://www.cs.gmu.edu/~offutt/softwaretest/>.

- (e) List test paths that achieve Edge Coverage but not prime Path Coverage on the graph.

Solution:

This is certainly possible; you can find test paths for Edge Coverage and Prime Path Coverage online. See the graph tool at <http://www.cs.gmu.edu/~offutt/softwaretest/>.

7. Use the following method `printPrimes()` for questions a-f below.

```

1. /** *****
2.  * Finds and prints n prime integers
3.  * Jeff Offutt, Spring 2003
4.  ***** */
5. private static void printPrimes (int n)
6. {
7.     int curPrime;           // Value currently considered for primeness
8.     int numPrimes;         // Number of primes found so far.
9.     boolean isPrime;       // Is curPrime prime?
10.    int [] primes = new int [MAXPRIMES]; // The list of prime numbers.
11.
12.    // Initialize 2 into the list of primes.
13.    primes [0] = 2;
14.    numPrimes = 1;
15.    curPrime = 2;
16.    while (numPrimes < n)
17.    {
18.        curPrime++; // next number to consider ...
19.        isPrime = true;
20.        for (int i = 0; i <= numPrimes-1; i++)
21.        { // for each previous prime.
22.            if (isDivisible (primes[i], curPrime))
23.            { // Found a divisor, curPrime is not prime.
24.                isPrime = false;
25.                break; // out of loop through primes.
26.            }
27.        }
28.        if (isPrime)
29.        { // save it!
30.            primes[numPrimes] = curPrime;
31.            numPrimes++;
32.        }
33.    } // End while
34.
35.    // Print all the primes out.
36.    for (int i = 0; i <= numPrimes-1; i++)
37.    {
38.        System.out.println ("Prime: " + primes[i]);
39.    }
40. } // end printPrimes

```

- (a) Draw the control flow graph for the `printPrimes()` method.

Instructor Solution Only

- (b) Consider test cases $t_1 = (n = 3)$ and $t_2 = (n = 5)$. Although these tour the same prime paths in `printPrimes()`, they do not necessarily find the same faults. Design a simple fault that t_2 would be more likely to discover than t_1 would.

Instructor Solution Only

- (c) For `printPrimes()`, find a test case such that the corresponding test path visits the edge that connects the beginning of the **while** statement to the **for** statement **without** going through the body of the while loop.

Instructor Solution Only

- (d) Enumerate the test requirements for Node Coverage, Edge Coverage, and Prime Path Coverage for the graph for `printPrimes()`.

Instructor Solution Only

- (e) List test paths that achieve Node Coverage but not Edge Coverage on the graph.

Instructor Solution Only

- (f) List test paths that achieve Edge Coverage but not Prime Path Coverage on the graph.

Instructor Solution Only

Exercises, Section 2.4

1. Use the class *Stutter* in Figures 2.34 and 2.35 in Section 2.5 to answer questions a-d below.

- (a) Draw a control flow graph for *Stutter*.

Solution:

The point of this question is getting students to think about how to handle the call sites. Figure 2.36, which gives control flow graphs for methods `stut()`, `checkDupes()`, and `isDelimit()`, illustrates the problem of trying naively to aggregate control flow graphs. The alternate approach of expanding each method call into its corresponding graph unfortunately causes the control flow graph to balloon in size. Hence, neither approach is particularly useful. However, just tracking du-pairs accross call-sites, as parts (b), (c), and (d) do below, does scale.

- (b) List all the call sites.

The version in the book does not have line numbers. The answers below use the line numbers in the following version.

```

1 // Jeff Offutt - June 1989, Java version 2003
2
3 // stutter checks for repeat words in a text file.
4 // It prints a list of repeat words, by line number.
5 // stutter will accept standard input or a list
6 // of file names.
7
8 import java.io.*;
9
10 class stutter
11 {
12     // Class variables used in multiple methods.
13     private static boolean lastdelimit = true;
14     private static String curWord = "", prevWord = "";
15     private static char delimits [] =
16         {' ', ' ', ' ', ' ', ' ', '!', '-', '+', '=', ':', '?', '&', '{', '}', '\\'};
17
18 //*****
19 // main parses the arguments, decides if stdin
20 // or a file name, and calls stut().
21 //*****
22 public static void main (String[] args) throws IOException
23 {
24     String fileName;
25     FileReader myFile;
26     BufferedReader inFile = null;
27
28     if (args.length == 0)
29     { // no file, use stdin
30         inFile = new BufferedReader (new InputStreamReader (System.in));
31     }
32     else
33     {
34         fileName = args [0];
35         if (fileName == null)
36         { // no file name, use stdin
37             inFile = new BufferedReader (new InputStreamReader (System.in));
38         }
39         else
40         { // file name, open the file.
41             myFile = new FileReader (fileName);
42             inFile = new BufferedReader (myFile);
43         }
44     }
45
46     stut (inFile);
47 }
48
49 //*****
50 //*****
51 private static void stut (BufferedReader inFile) throws IOException
52 {
53     String inLine;
54     char c;
55     int linecnt = 1;

```

```

56
57 while ((inLine = inFile.readLine()) != null)
58 { // For each line
59
60     for (int i=0; i<inLine.length(); i++)
61     { // for each character
62         c = inLine.charAt(i);
63
64         if (isDelimit (c))
65         { // Found an end of a word.
66             checkDupes (linecnt);
67         }
68         else
69         {
70             lastdelimit = false;
71             curWord = curWord + c;
72         }
73     }
74     checkDupes (linecnt);
75     linecnt++;
76 }
77 } // end stut
78
79 //*****
80 //*****
81 private static void checkDupes (int line)
82 {
83     if (lastdelimit)
84         return; // already checked, keep skipping
85     lastdelimit = true;
86     if (curWord.equals(prevWord))
87     {
88         System.out.println ("Repeated word on line " + line + ": " + prevWord+ " " + curWord);
89     }
90     else
91     {
92         prevWord = curWord;
93     }
94     curWord = "";
95 } // end checkDupes
96
97 //*****
98 //*****
99 private static boolean isDelimit (char C)
100 {
101     for (int i = 0; i < delimits.length; i++)
102         if (C == delimits [i])
103             return (true);
104     return (false);
105 }
106
107 } // end class stutter

```

Solution:

The callsites are:

- i. Line 46, *main()* → *stut()*
- ii. Line 64, *stut()* → *isDelimit()*
- iii. Line 66, *stut()* → *checkDupes()*
- iv. Line 74, *stut()* → *checkDupes()*

(c) List all du-pairs for each call site.

Solution:

- i.* (*main()*, *curWord*, 14) → (*stut()*, *curWord*, 71) – line 46
 - ii.* (*main()*, *inFile*, 30) → (*stut()*, *inFile*, 57) – line 46
 - iii.* (*main()*, *inFile*, 37) → (*stut()*, *inFile*, 57) – line 46
 - iv.* (*main()*, *inFile*, 42) → (*stut()*, *inFile*, 57) – line 46
 - v.* (*stut()*, *c*, 62) → (*isDelimit()*, *C*, 102) – line 64
 - vi.* (*stut()*, *linecnt*, 55) → (*checkDupes()*, *line*, 88) – line 66
 - vii.* (*stut()*, *linecnt*, 75) → (*checkDupes()*, *line*, 88) – line 66
 - viii.* (*stut()*, *curWord*, 71) → (*checkDupes()*, *curWord*, 86) – line 66
 - ix.* (*stut()*, *lastdelimiter*, 70) → (*checkDupes()*, *lastdelimiter*, 83) – line 66
 - x.* (*checkDupes()*, *curWord*, 94) → (*stut()*, *curWord*, 71) – line 66
 - xi.* (*stut()*, *linecnt*, 55) → (*checkDupes()*, *line*, 88) – line 74
 - xii.* (*stut()*, *linecnt*, 75) → (*checkDupes()*, *line*, 88) – line 74
 - xiii.* (*stut()*, *curWord*, 71) → (*checkDupes()*, *curWord*, 86) – line 74
 - xiv.* (*stut()*, *lastdelimiter*, 70) → (*checkDupes()*, *lastdelimiter*, 83) – line 74
 - xv.* (*checkDupes()*, *curWord*, 94) → (*stut()*, *curWord*, 71) – line 74
- (d) Create test data to satisfy *All-Coupling Use Coverage* for *Stutter*.

Solution:

Consider the following tests:

- t_1 :
word
 - t_2 :
word word
 - t_3 :
first line
word word
- i.* (*main()*, *curWord*, 14) → (*stut()*, *curWord*, 71) – line 46
Test needs to start with a non-delimiter: t_1 .
 - ii.* (*main()*, *inFile*, 30) → (*stut()*, *inFile*, 57) – line 46
Test needs to come from standard input.
 - iii.* (*main()*, *inFile*, 37) → (*stut()*, *inFile*, 57) – line 46
Test not possible in normal usage.
 - iv.* (*main()*, *inFile*, 42) → (*stut()*, *inFile*, 57) – line 46
Test needs to come from file.
 - v.* (*stut()*, *c*, 62) → (*isDelimit()*, *C*, 102) – line 64
Test needs to be nonempty: t_1 .

- vi. (*stut()*, *linecnt*, 55) → (*checkDupes()*, *line*, 88) – line 66
Test needs to stutter on first line: t_2 .
- vii. (*stut()*, *linecnt*, 75) → (*checkDupes()*, *line*, 88) – line 66
Test needs to have on second or later lines: t_3 .
- viii. (*stut()*, *curWord*, 71) → (*checkDupes()*, *curWord*, 86) – line 66
Test needs to find a word, and then a delimiter: t_2 .
- ix. (*stut()*, *lastdelimiter*, 70) → (*checkDupes()*, *lastdelimiter*, 83) – line 66
Test needs to find a word, and then a delimiter: t_2 .
- x. (*checkDupes()*, *curWord*, 94) → (*stut()*, *curWord*, 71) – line 66
Test needs multiple words: t_2 .
- xi. (*stut()*, *linecnt*, 55) → (*checkDupes()*, *line*, 88) – line 74
Test needs to be nonempty: t_1 .
- xii. (*stut()*, *linecnt*, 75) → (*checkDupes()*, *line*, 88) – line 74
Test needs to have on second or later lines: t_3 .
- xiii. (*stut()*, *curWord*, 71) → (*checkDupes()*, *curWord*, 86) – line 74
Test needs to stutter: t_2 .
- xiv. (*stut()*, *lastdelimiter*, 70) → (*checkDupes()*, *lastdelimiter*, 83) – line 74
Test needs a line that ends with a non-delimiter: t_1 .
- xv. (*checkDupes()*, *curWord*, 94) → (*stut()*, *curWord*, 71) – line 74
Test needs a line that ends with a non delimiter: t_1 .

The observant reader might notice that **stutter** actually fails on both t_2 and t_3 , in that it reports the wrong line number. Credit for noticing this goes to Mark Cornwell.

Also thanks to Hooman Safaee for noticing that lines 74 and 75 were switched, both in the text and in the uncorrected solutions. The ripple effect from this error affected a fair amount of the solution.

2. Use the following program fragment for questions a-e below.

```

public static void f1 (int x, int y)
{
    if (x < y) { f2 (y); } else { f3 (y); };
}
public static void f2 (int a)
{
    if (a % 2 == 0) { f3 (2*a); };
}
public static void f3 (int b)
{
    if (b > 0) { f4(); } else { f5(); };
}
public static void f4() {... f6()...}
public static void f5() {... f6()...}
public static void f6() {...}

```

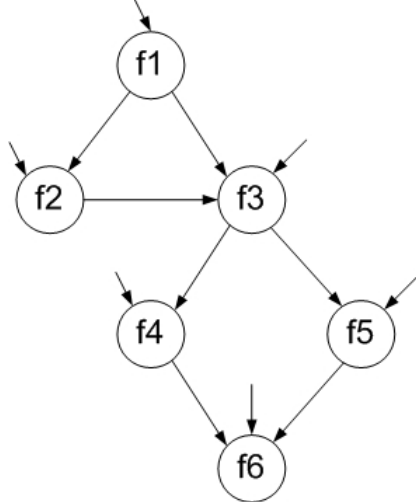
Use the following test inputs:

- $t_1 = f1 (0, 0)$

- $t_2 = f1 (1, 1)$
- $t_3 = f1 (0, 1)$
- $t_4 = f1 (3, 2)$
- $t_5 = f1 (3, 4)$

(a) Draw the call graph for this program fragment.

Solution:



Note that in the call graph, all of the public methods are potentially initial nodes. Final nodes are not marked in the given diagram. However, the code shown suggests that f2 is final (since there is no requirement that further calls are made), and that f6 may be final (no calls are shown). The remaining nodes are not final; there is always another call from f1, f3, f4, and f5.

(b) Give the path in the graph followed by each test.

Solution:

- $t_1: [f1, f3, f5, f6]$
- $t_2: [f1, f3, f4, f6]$
- $t_3: [f1, f2]$
- $t_4: [f1, f3, f4, f6]$
- $t_5: [f1, f2, f3, f4, f6]$

(c) Find a minimal test set that achieves Node Coverage.

Solution:

Three possibilities: $\{t_1, t_2, t_3\}$, $\{t_1, t_3, t_4\}$, or $\{t_1, t_5\}$.

(d) Find a minimal test set that achieves Edge Coverage.

Solution:

One possibility: $\{t_1, t_5\}$

(e) Give the prime paths in the graph. Give the prime path that is not covered by any of the test paths above.

Solution:

There are 4 prime paths: { [f1, f2, f3, f4, f6], [f1, f2, f3, f5, f6], [f1, f3, f4, f6], [f1, f3, f5, f6] }. The second of these paths is not covered by the given test paths.

3. Use the following methods **trash()** and **takeOut()** to answer questions a-c.

```

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

```

- (a) Give all call sites using the line numbers given.

Instructor Solution Only

- (b) Give all pairs of *last-defs* and *first-uses*.

Instructor Solution Only

- (c) Provide test inputs that satisfy *all-coupling-uses* (note that **trash()** only has one input).

Instructor Solution Only

Exercises, Section 2.5

1. Use the class **Queue** in Figure 2.40 for questions a-f below. The queue is managed in the usual circular fashion.

Suppose we build a FSM where states are defined by the representation variables of **Queue**. That is, a state is a 4-tuple defined by the values for [*elements*, *size*, *front*, *back*]. For example, the initial state has the value $[[null, null], 0, 0, 0]$, and the state that results from pushing an object *obj* onto the queue in its initial state is $[[obj, null], 1, 0, 1]$.

- (a) We do not actually care which specific objects are in the queue. Consequently, there are really just four useful values for the variable *elements*. Enumerate them.

Instructor Solution Only

- (b) How many states are there?

Instructor Solution Only

- (c) How many of these states are reachable?

Instructor Solution Only

```

public class Queue
{ // Overview: a Queue is a mutable, bounded FIFO data structure
  // of fixed size (size is 2, for this exercise).
  // A typical Queue is [], [o1], or [o1, o2], where neither o1 nor o2
  // are ever null. Older elements are listed before newer ones.
  private Object[] elements;
  private int size, front, back;
  private static final int capacity = 2;

  public Queue ()
  {
    elements = new Object [capacity];
    size = 0; front = 0; back = 0;
  }

  public void enqueue (Object o)
    throws NullPointerException, IllegalStateException
  { // Modifies: this
    // Effects: If argument is null throw NullPointerException
    // else if this is full, throw IllegalStateException,
    // else make o the newest element of this
    if (o == null)
      throw new NullPointerException ("Queue.enqueue");
    else if (size == capacity)
      throw new IllegalStateException ("Queue.enqueue");
    else
    {
      size++;
      elements [back] = o;
      back = (back+1) % capacity;
    }
  }

  public Object dequeue () throws IllegalStateException
  { // Modifies: this
    // Effects: If queue is empty, throw IllegalStateException,
    // else remove and return oldest element of this

    if (size == 0)
      throw new IllegalStateException ("Queue.dequeue");
    else
    {
      size--;
      Object o = elements [ (front % capacity) ];
      elements [front] = null;
      front = (front+1) % capacity;
      return o;
    }
  }

  public boolean isEmpty() { return (size == 0); }
  public boolean isFull() { return (size == capacity); }

  public String toString()
  {
    String result = "[";
    for (int i = 0; i < size; i++)
    {
      result += elements[ (front + i) % capacity ] . toString();
      if (i < size -1) {
        result += ", ";
      }
    }
    result += "]";
    return result;
  }
}

```

Figure 1: Class Queue for exercises.

- (d) Show the reachable states in a drawing.

Instructor Solution Only

- (e) Add edges for the *enqueue()* and *dequeue()* methods. (For this assignment, ignore the exceptional returns, although you should observe that when exceptional returns are taken, none of the instance variables are modified.)

Instructor Solution Only

- (f) Define a small test set that achieves Edge Coverage. Implement and execute this test set. You might find it helpful to write a method that shows the internal variables at each call.

Instructor Solution Only

2. For the following questions a-c, consider the method FSM for a (simplified) programmable thermostat. Suppose the variables that define the state and the methods that transition between states are:

```
partOfDay : {Wake, Sleep}
temp      : {Low, High}

// Initially "Wake" at "Low" temperature

// Effects: Advance to next part of day
public void advance();

// Effects: Make current temp higher, if possible
public void up();

// Effects: Make current temp lower, if possible
public void down();
```

- (a) How many states are there?

Instructor Solution Only

- (b) Draw and label the states (with variable values) and transitions (with method names). Notice that all of the methods are total.

Instructor Solution Only

- (c) A test case is simply a sequence of method calls. Provide a test set that satisfies Edge Coverage on your graph.

Instructor Solution Only

Exercises, Section 2.6

1. Construct two separate use cases and use case scenarios for interactions with a bank Automated Teller Machine. Do not try to capture all the functionality of the ATM into one graph; think about two different people using the ATM and what each one might do.

Design test cases for your scenarios.

Instructor Solution Only

Exercises, Section 2.7

- Derive and simplify the path expressions for the three graphs in Figure 2.43.

Solution:

For figure 2.43(a), the path expression can be read directly from the figure:

$$a (bd + ce) (fh + gi) j$$

For figure 2.43(b), the path expression also can be read more or less directly from the figure:

$$ab (cd)^* [cf + e] g$$

Figure 2.43(c) is a bit trickier. The general form is $[A + B]^* k$, where A is the small loop and B is the large loop.

A has the form: $a g^* f$

B has the form: $b (ch + di + ej)$

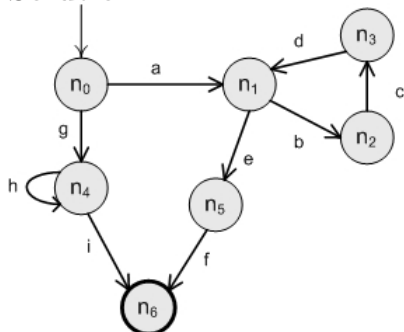
The combined expression is: $[[a g^* f] + [b (ch + di + ej)]]^* k$

- Derive and simplify the path expression for the flow graph in Figure 2.12. Assign reasonable cycle weights and compute the maximum number of paths in the graph and the minimum number of paths to reach all edges.

Instructor Solution Only

- The graph in Figure 2.10 was used as an example for prime test paths. Add appropriate edge labels to the graph, then derive and simplify the path expressions. Next add edge weights of 1 for non-cycle edges and 5 for cycle edges. Then compute the maximum number of paths in the graph and the minimum number of paths to reach all edges. This graph has 25 prime paths. Briefly discuss the number of prime paths with the maximum number of paths and consider the effect of varying the cycle weight on the maximum number of paths.

Solution:



The path expression can be read off directly: $a(bcd)^*ef + gh^*i$

For weights, if we set the weight W_A of b and h to 5 and apply the formula for maximum number of paths, we get:

$$\begin{aligned}
& 1 * ((5)^{0-n} * 1 * 1) * 1 * 1 + 1 * 5^{0-n} * 1 \\
& = 1 * (\sum_{i=0}^n 5^i * 1 * 1) * 1 * 1 + 1 * \sum_{i=0}^n 5^i * 1 \\
& = 2 \sum_{i=0}^n 5^i
\end{aligned}$$

If we choose n to be 2, we get:

$$\begin{aligned}
& 2 \sum_{i=0}^2 5^i \\
& = 2 * (5^0 + 5^1 + 5^2) = 2 * (1 + 5 + 25) = 62
\end{aligned}$$

On the other hand, if we choose n to be 3, we get:

$$\begin{aligned}
& 2 \sum_{i=0}^3 5^i \\
& = 2 * (5^0 + 5^1 + 5^2 + 5^3) = 2 * (1 + 5 + 25 + 125) = 312
\end{aligned}$$

If we apply the formula for the minimum number of paths with W_A set to 5, we get:

$$1 * (W_A * 1 * 1) * 1 * 1 + 1 * W_A * 1,$$

which simplifies to 10.

As for the relation to prime paths, note that setting the number of iterations at anything other than a **very** small number results in a prohibitive number of paths to test under the maximum test path model. While prime paths counts can also grow exponentially, for small, and arguably typical, graphs such as this, the number of paths to test is comparatively quite manageable.

- Section 2.5 presented four different versions of a FSM for **Stutter**. Derive and simplify the path expressions for each of the four variations, then compute the maximum number of paths and the minimum number of paths to reach all edges in each. Discuss how the different numbers affect testing.

Instructor Solution Only

- Perform complementary operations analysis on the graph in Figure 2.32. Assume complementary operators of **open** and **close**.

Instructor Solution Only

- Derive and simplify the path expressions for the activity graph in Figure 2.42. The only loop in that graph has a bound of 3 tries. Use that to compute the maximum number of paths and the minimum number of paths to reach all edges. Discuss the relationship between the scenarios for that graph and the terms in the path expression.

Instructor Solution Only

- Answer questions a-c for the graph defined by the following sets:

- $N = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- $N_0 = \{1\}$
- $N_f = \{10\}$

- $E = \{(1, 2, a), (2, 3, b), (2, 4, c), (3, 5, d), (4, 5, e), (5, 6, f), (5, 7, g), (6, 6, h(1-4)), (6, 10, i), (7, 8, j), (8, 8, k(0-3)), (8, 9, l), (9, 7, m(2-5)), (9, 10, n)\}$

(a) Draw the graph.

Solution:

See the graph tool at <http://www.cs.gmu.edu/~offutt/softwaretest/>

The basic path expression is: $a(bd + cd) [fh^*i + gjk^*l [mjk^*l] n]$

(b) What is the maximum number of paths through the graph?

Solution:

We decorate the path expression with the loop counts and substitute:

$$\begin{aligned}
 & 1(1 * 1 + 1 * 1) * [1 * 1^{1-4} * 1 + 1 * 1 * 1^{0-3} * [1 [1 * 1 * 1^{0-3} * 1]]^{2-5} * 1] \\
 &= (2) * [1^{1-4} + 1^{0-3} * (1^{0-3})^{2-5}] \\
 &= (2) * [\sum_{i=1}^4 1^i + \sum_{i=0}^3 1^i * \sum_{i=2}^5 (\sum_{j=0}^3 1^j)^i] \\
 &= (2) * [(1 + 1 + 1 + 1) + (1 + 1 + 1 + 1) * \sum_{i=2}^5 (1 + 1 + 1 + 1)^i] \\
 &= (2) * [4 + 4 * \sum_{i=2}^5 4^i] \\
 &= (2) * [4 + 4 * 4^2 + 4^3 + 4^4 + 4^5] \\
 &= (2) * [4 + 4 * 16 + 64 + 256 + 1024] \\
 &= (2) * [4 + 4 * 16 + 64 + 256 + 1024] \\
 &= (2) * [4 + 4 * 1360] \\
 &= (2) * [4 + 5440] \\
 &= (2) * [5444] \\
 &= 10888
 \end{aligned}$$

Probably, this is a ridiculous number of test paths for this graph. To get a more reasonable number, the iteration labels on the loops need to be tweaked.

(c) What is the approximate minimum number of paths through the graph?

Solution:

We again start with the same substitution, except we now replace with minimum values. The only tricky part is the loop for edge m . If we assume a single test path does this, we get:

$$\begin{aligned}
 & a(bd + cd) [fh^*i + gjk^*l [mjk^*l] n] \\
 &= \max(2, [1 + \max(1, 1)]) \\
 &= \max(2, 2) \\
 &= 2.
 \end{aligned}$$

Probably, 2 test paths won't do a very good job of testing this graph.

Exercises, Section 3.1

1. List all the clauses for the predicate below:

$$((f \leq g) \wedge (X > 0)) \vee (M \wedge (e < d + c))$$

Solution:

There are four: $f \leq g$, $X > 0$, M , and $e < d + c$.

2. Write the predicate (only the predicate) to represent the requirement: “List all the wireless mice that either retail for more than \$100 or for which the store has more than 20 items. Also list non-wireless mice that retail for more than \$50.”

Solution:

The predicate describing whether to list a given mouse is:

$$((mouseType = wireless) \wedge ((retail > 100) \vee (stock > 20))) \vee (\neg(mouseType = wireless) \wedge (retail > 50))$$

Note: Many students need additional practice with this type of exercise. Typical textbooks used in discrete structures classes are an excellent source for sample problems.

Exercises, Section 3.2

Use predicates (1) through (10) to answer the following questions.

1. $p = a \wedge (\neg b \vee c)$
2. $p = a \vee (b \wedge c)$
3. $p = a \wedge b$
4. $p = a \rightarrow (b \rightarrow c)$
5. $p = a \oplus b$
6. $p = a \leftrightarrow (b \wedge c)$
7. $p = (a \vee b) \wedge (c \vee d)$
8. $p = (\neg a \wedge \neg b) \vee (a \wedge \neg c) \vee (\neg a \wedge c)$
9. $p = a \vee b \vee (c \wedge d)$
10. $p = (a \wedge b) \vee (b \wedge c) \vee (a \wedge c)$

- (a) Identify the clauses that go with predicate p .
- (b) Compute (and simplify) the conditions under which each of the clauses determines predicate p .
- (c) Write the complete truth table for all clauses. Label your rows starting from 1. Use the format in the example underneath the definition of Combinatorial Coverage in Section 3.2. That is, row 1 should be all clauses true. You should include columns for the conditions under which each clause determines the predicate, and also a column for the predicate itself.
- (d) Identify all pairs of rows from your table that satisfy General Active Clause Coverage (GACC) with respect to each clause.
- (e) Identify all pairs of rows from your table that satisfy Correlated Active Clause Coverage (CACC) with respect to each clause.
- (f) Identify all pairs of rows from your table that satisfy Restricted Active Clause Coverage (RACC) with respect to each clause.
- (g) Identify all 4-tuples of rows from your table that satisfy General Inactive Clause Coverage (GICC) with respect to each clause. Identify any infeasible GICC test requirements.
- (h) Identify all 4-tuples of rows from your table that satisfy Restricted Inactive Clause Coverage (RICC) with respect to each clause. Identify any infeasible RICC test requirements.

Instructor Solution Only

Solution:

Solution for $p = a \vee (b \wedge c)$

(a) *Clauses are a, b, c .*

(b) $p_a = \neg b \vee \neg c$

$p_b = \neg a \wedge c$

$p_c = \neg a \wedge b$

(c) *Note: Blank cells represent values of 'F'.*

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T			
2	T	T	F	T	T		
3	T	F	T	T	T		
4	T	F	F	T	T		
5	F	T	T	T		T	T
6	F	T	F		T		T
7	F	F	T		T	T	
8	F	F	F		T		

(d) *GACC pairs for clause a are: $\{2, 3, 4\} \times \{6, 7, 8\}$.*

There is only one GACC pair for clause b : $(5, 7)$.

There is only one GACC pair for clause c : $(5, 6)$.

(e) *CACC pairs for clauses a, b , and c are the same as GACC pairs.*

(f) *RACC pairs for clause a are: $(2, 6), (3, 7), (4, 8)$.*

RACC pairs for clauses b and c are the same as CACC pairs.

(g) *GICC tuples for a are: no feasible pair for $p = F$; $(1, 5)$ for $p = T$.*

GICC tuples for b are: $(6, 8)$ for $p = F$; $\{1, 2\} \times \{3, 4\}$ for $p = T$.

GICC tuples for c are: $(7, 8)$ for $p = F$; $\{1, 3\} \times \{2, 4\}$ for $p = T$.

(h) *RICC tuples for a are same as GICC.*

RICC tuples for b are: $(6, 8)$ for $p = F$; $(1, 3), (2, 4)$ for $p = T$.

RICC tuples for c are: $(7, 8)$ for $p = F$; $(1, 2), (3, 4)$ for $p = T$.

Solution:

Solution for $p = a \wedge b$

(a) *Clauses are a, b .*

(b) $p_a = b$

$p_b = a$

(c) *Note: Blank cells represent values of 'F'.*

	a	b	p	p_a	p_b
1	T	T	T	T	T
2	T	F			T
3	F	T		T	
4	F	F			

(d) *There is only one GACC pair for clause a: (1, 3).*

There is only one GACC pair for clause b: (1, 2).

(e) *CACC pairs for clauses a and b are the same as GACC pairs.*

(f) *RACC pairs for clauses a and b are the same as CACC pairs.*

(g) *GICC tuples for a are: (2, 4) for $p = F$; no feasible pair for $p = T$.*

GICC tuples for b are: (3, 4) for $p = F$; no feasible pair for $p = T$.

(h) *RICC tuples for clauses a and b are the same as GICC tuples.*

Solution:

Solution for $p = a \rightarrow (b \rightarrow c)$

(a) *Clauses are a, b, c .*

(b) $p_a = b \wedge \neg c$

$p_b = a \wedge \neg c$

$p_c = a \wedge b$

(c) *Note: Blank cells represent values of 'F'.*

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T			T
2	T	T	F		T	T	T
3	T	F	T	T			
4	T	F	F	T		T	
5	F	T	T	T			
6	F	T	F	T	T		
7	F	F	T	T			
8	F	F	F	T			

(d) *There is only one GACC pair for clause a: (2, 6).*

There is only one GACC pair for clause b: (2, 4).

There is only one GACC pair for clause c: (1, 2).

(e) *CACC pairs for clauses a, b, and c are the same as GACC pairs.*

(f) *RACC pairs for clauses a, b, and c are the same as CACC pairs.*

(g) *GICC tuples for a are: no feasible pair for $p = F$; $\{1, 3, 4\} \times \{5, 7, 8\}$ for $p = T$.*

GICC tuples for b are: no feasible pair for $p = F$; $\{1, 5, 6\} \times \{3, 7, 8\}$ for $p = T$.

GICC tuples for c are: no feasible pair for $p = F$; $\{3, 5, 7\} \times \{4, 6, 8\}$ for $p = T$.

(h) *RICC tuples for a are: no feasible pair for $p = F$; (1, 5), (3, 7), (4, 8) for $p = T$.*

RICC tuples for b are: no feasible pair for $p = F$; (1, 3), (5, 7), (6, 8) for $p = T$.

RICC tuples for c are: no feasible pair for $p = F$; (3, 4), (5, 6), (7, 8) for $p = T$.

Instructor Solution Only

Solution:

Solution for $p = a \leftrightarrow (b \wedge c)$

(a) *Clauses are a, b, c .*

(b) $p_a = T$

$p_b = c$

$p_c = b$

(c) *Note: Blank cells represent values of 'F'.*

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T	T	T
2	T	T	F		T		T
3	T	F	T		T	T	
4	T	F	F		T		
5	F	T	T		T	T	T
6	F	T	F	T	T		T
7	F	F	T	T	T	T	
8	F	F	F	T	T		

(d) *GACC pairs for clause a are: $\{1, 2, 3, 4\} \times \{5, 6, 7, 8\}$.*

GACC pairs for clause b are: $\{1, 5\} \times \{3, 7\}$.

GACC pairs for clause c are: $\{1, 5\} \times \{2, 6\}$.

(e) *CACC pairs for clause a are: $(1, 5) \cup \{2, 3, 4\} \times \{6, 7, 8\}$.*

CACC pairs for clause b are: $(1, 3), (5, 7)$ for clause b .

CACC pairs for clause c are: $(1, 2), (5, 6)$ for clause c .

(f) *RACC pairs for clause a are: $(1, 5), (2, 6), (3, 7), (4, 8)$.*

RACC pairs for clauses b and c are the same as CACC pairs.

(g) *There are no GICC tuples for clause a .*

GICC tuples for b are: $(2, 4)$ for $p = F$; $(6, 8)$ for $p = T$.

GICC tuples for c are: $(3, 4)$ for $p = F$; $(7, 8)$ for $p = T$.

(h) *RICC tuples for clauses a, b , and c are the same as GICC tuples.*

Solution:

Solution for $p = (a \vee b) \wedge (c \vee d)$

(a) *Clauses are a, b, c, d .*

(b) $p_a = \neg b \wedge (c \vee d)$

$p_b = \neg a \wedge (c \vee d)$

$p_c = \neg d \wedge (a \vee b)$

$p_d = \neg c \wedge (a \vee b)$

(c) *Note: Blank cells represent values of 'F'.*

	a	b	c	d	p	p_a	p_b	p_c	p_d
1	T	T	T	T	T				
2	T	T	T	F	T			T	
3	T	T	F	T	T				T
4	T	T	F	F				T	T
5	T	F	T	T	T	T			
6	T	F	T	F	T	T		T	
7	T	F	F	T	T	T			T
8	T	F	F	F				T	T
9	F	T	T	T	T		T		
10	F	T	T	F	T		T	T	
11	F	T	F	T	T		T		T
12	F	T	F	F				T	T
13	F	F	T	T		T	T		
14	F	F	T	F		T	T		
15	F	F	F	T		T	T		
16	F	F	F	F					

(d) *GACC pairs for clause a are: $\{5, 6, 7\} \times \{13, 14, 15\}$.*

GACC pairs for clause b are: $\{9, 10, 11\} \times \{13, 14, 15\}$.

GACC pairs for clause c are: $\{2, 6, 10\} \times \{4, 8, 12\}$.

GACC pairs for clause d are: $\{3, 7, 11\} \times \{4, 8, 12\}$.

(e) *CACC pairs for clauses $a, b, c,$ and d are the same as GACC pairs.*

(f) *RACC pairs for clause $a,$ $(5, 13), (6, 14), (7, 15)$.*

RACC pairs for clause $b,$ $(9, 13), (10, 14), (11, 15)$.

RACC pairs for clause $c,$ $(2, 4), (6, 8), (10, 12)$.

RACC pairs for clause $d,$ $(3, 4), (7, 8), (11, 12)$.

- (g) *GICC tuples for a are:*
 $\{4, 8\} \times \{12, 16\}$ for $p = F$; $\{1, 2, 3\} \times \{9, 10, 11\}$ for $p = T$.
GICC tuples for b are:
 $\{4, 12\} \times \{8, 16\}$ for $p = F$; $\{1, 2, 3\} \times \{5, 6, 7\}$ for $p = T$.
GICC tuples for c are:
 $\{13, 14\} \times \{15, 16\}$ for $p = F$; $\{1, 5, 9\} \times \{3, 7, 11\}$ for $p = T$.
GICC tuples for d are:
 $\{13, 15\} \times \{14, 16\}$ for $p = F$; $\{1, 5, 9\} \times \{2, 6, 10\}$ for $p = T$.
- (h) *RICC tuples for a are:*
 $(4, 12), (8, 16)$ for $p = F$; $(1, 9), (2, 10), (3, 11)$ for $p = T$.
RICC tuples for b are:
 $(4, 8), (12, 16)$ for $p = F$; $(1, 5), (2, 6), (3, 7)$ for $p = T$.
RICC tuples for c are:
 $(13, 15), (14, 16)$ for $p = F$; $(1, 3), (5, 7), (9, 11)$ for $p = T$.
RICC tuples for d are:
 $(13, 14), (15, 16)$ for $p = F$; $(1, 2), (5, 6), (9, 10)$ for $p = T$.

Solution:

Solution for $p = (\neg a \wedge \neg b) \vee (a \wedge \neg c) \vee (\neg a \wedge c)$

(a) *Clauses are* a, b, c .

(b) $p_a = b \vee c$

$p_b = \neg a \wedge \neg c$

$p_c = a \vee b$

(c) *Note: Blank cells represent values of 'F'.*

	a	b	c	p	p_a	p_b	p_c
1	T	T	T		T		T
2	T	T	F	T	T		T
3	T	F	T		T		T
4	T	F	F	T			T
5	F	T	T	T	T		T
6	F	T	F		T	T	T
7	F	F	T	T	T		
8	F	F	F	T		T	

(d) *GACC pairs for clause* a *are:* $\{1, 2, 3\} \times \{5, 6, 7\}$.

GACC pair for clause b *is:* $(6, 8)$.

GACC pairs for clause c *are:* $\{1, 3, 5\} \times \{2, 4, 6\}$.

(e) *CACC pairs for clause* a *are:* $(1, 5), (1, 7), (2, 6), (3, 5), (3, 7)$.

The CACC pair for clause b *is the same as GACC pair.*

CACC pairs for clause c *are:* $(1, 2), (1, 4), (3, 2), (3, 4), (5, 6)$.

(f) *RACC pair for clause* a *is:* $(1, 5), (2, 6), (3, 7)$.

The RACC pair for clause b *is the same as CACC pair.*

RACC pairs for clause c *are:* $(1, 2), (3, 4), (5, 6)$.

(g) *GICC tuples for clause* a *are:* *no feasible pair for* $p = F$; $(4, 8)$ *for* $p = T$.

GICC tuples for clause b *are:* $(1, 3)$ *for* $p = F$; $(2, 4), (2, 7), (5, 4), (5, 7)$ *for* $p = T$.

GICC tuples for clause c *are:* *no feasible pair for* $p = F$; $(7, 8)$ *for* $p = T$.

(h) *RICC tuples for clauses* a *and* c *are same as GICC tuples.*

RICC tuples for clause b *are:* $(1, 3)$ *for* $p = F$; $(2, 4), (5, 7)$ *for* $p = T$.

Instructor Solution Only

Instructor Solution Only

11. Refine the GACC, CACC, RACC, GICC, and RICC coverage criteria so that the constraints on the minor clauses are made more formal.

Solution:

Solution: We'll start with GACC, and add more constraints we we proceed to RACC. Then, we'll repeat the process for GICC and RICC. The goal here is to help students by giving a more explicit explanation of the test requirements.

We assume p is the predicate, c_i is the major clause, $c_j, j \neq i$ are the minor clauses, and p_{c_i} is the conditions under which c_i determines p .

For each i , GACC has two test requirements: $c_i = T \wedge p_{c_i} = T$ and $c_i = F \wedge p_{c_i} = T$. Note that the values of the minor clauses c_j may differ between the two tests.

For each i , CACC has two test requirements: $c_i = T \wedge p_{c_i} = T$ and $c_i = F \wedge p_{c_i} = T$. Additionally, the value of p resulting from the first test must differ from the value of p resulting from the second. Note that the values of the minor clauses c_j may differ between the two tests.

For each i , RACC has two test requirements: $c_i = T \wedge p_{c_i} = T$ and $c_i = F \wedge p_{c_i} = T$. Additionally, c_i is the only difference between the two tests. That is, the values of the minor clauses c_j must be identical on the two tests.

For each i , GICC has two pairs of test requirements:

Pair 1: $c_i = T \wedge p_{c_i} = F \wedge p = T$. $c_i = F \wedge p_{c_i} = F \wedge p = T$.

Pair 2: $c_i = T \wedge p_{c_i} = F \wedge p = F$. $c_i = F \wedge p_{c_i} = F \wedge p = F$.

The minor clauses c_j may differ between the two tests. Often, one of the pairs is infeasible.

For each i , RICC has two pairs of test requirements:

Pair 1: $c_i = T \wedge p_{c_i} = F \wedge p = T$. $c_i = F \wedge p_{c_i} = F \wedge p = T$.

Pair 2: $c_i = T \wedge p_{c_i} = F \wedge p = F$. $c_i = F \wedge p_{c_i} = F \wedge p = F$.

Additionally, c_i is the only difference between the two tests in Pair 1 and the two tests in Pair 2. That is, the values of the minor clauses c_j must be identical for the two tests in Pair 1 and identical for the two tests in Pair 2. Again, one of the pairs is often infeasible.

12. (**Challenging!**) Find a predicate and a set of additional constraints so that CACC is infeasible with respect to some clause, but GACC is feasible.

Instructor Solution Only

Exercises, Section 3.3

1. Answer the following questions for the method `checkIt()` below:

```
public static void checkIt (boolean a, boolean b, boolean c)
{
    if (a && (b || c))
    {
        System.out.println ("P is true");
    }
    else
    {
        System.out.println ("P isn't true");
    }
}
```

- Transform *checkIt()* to *checkItExpand()*, a method where each *if* statement tests exactly one boolean variable. Instrument *checkItExpand()* to record which edges are traversed. (“print” statements are fine for this.)

Instructor Solution Only

- Derive a GACC test set *T1* for *checkIt()*. Derive an Edge Coverage test set *T2* for *checkItExpand()*. Build *T2* so that it does **not** satisfy GACC on the predicate in *checkIt()*.

Instructor Solution Only

- Run both *T1* and *T2* on both *checkIt()* and *checkItExpand()*.

Instructor Solution Only

Output of the program:

```
true true true
checkIt():      1: P is true
checkItExpand(): 1: P is true
true true false
checkIt():      1: P is true
checkItExpand(): 1: P is true
true false true
checkIt():      1: P is true
checkItExpand(): 2: P is true
true false false
checkIt():      3: P isn't true
checkItExpand(): 3: P isn't true
false true true
checkIt():      3: P isn't true
checkItExpand(): 4: P isn't true
false true false
checkIt():      3: P isn't true
checkItExpand(): 4: P isn't true
false false true
```

```

checkIt():          3: P isn't true
checkItExpand():   4: P isn't true
false false false
checkIt():          3: P isn't true
checkItExpand():   4: P isn't true

```

2. Answer the following questions for the method `twoPred()` below:

```

public String twoPred (int x, int y)
{
    boolean z;

    if (x < y)
        z = true;
    else
        z = false;

    if (z && x+y == 10)
        return "A";
    else
        return "B";
}

```

- Identify test inputs for *twoPred()* that achieve Restricted Active Clause Coverage (RACC).

Instructor Solution Only

- Identify test inputs for *twoPred()* that achieve Restricted Inactive Clause Coverage (RICC).

Instructor Solution Only

3. Answer the following questions for the program fragments below:

fragment P:	fragment Q:
<pre> if (A B C) { m(); } return; </pre>	<pre> if (A) { m(); return; } if (B) { m(); return; } if (C) { m(); } </pre>

- Give a GACC test set for fragment P. (Note that GACC, CACC, and RACC yield identical test sets for this example.)

Solution:

Note that each clause must be true with the other clauses false, and then all of the clauses must be false, thereby yielding 4 tests (numbered 4, 6, 7, 8 in the usual truth table scheme): $T_{GACC} = \{(T, F, F), (F, T, F), (F, F, T), (F, F, F)\}$

- Does the GACC test set for fragment P satisfy Edge Coverage on fragment Q?

Solution:

Yes.

- Write down an Edge Coverage test set for fragment Q. Make your test set include as few tests from the GACC test set as possible.

Solution:

Eight possible answers exist. All answers must include (F, F, F) and (F, F, T), both of which are in the GACC tests. The first misses all calls to m(); and the second reaches the third call. To reach the first call, A must be True but B and C can have any value (four possibilities). To reach the second call, A must be False and B must be true. With “don’tcare” values, we can list the four tests as:

$$T_{EDGE} = \{(T, -, -), (F, T, -), (F, F, T), (F, F, F)\}$$

All the possible answers are:

$$T_{EDGE:1} = \{(T, T, T), (F, T, T), (F, F, T), (F, F, F)\}$$

$$T_{EDGE:2} = \{(T, T, F), (F, T, T), (F, F, T), (F, F, F)\}$$

$$T_{EDGE:3} = \{(T, F, T), (F, T, T), (F, F, T), (F, F, F)\}$$

$$T_{EDGE:4} = \{(T, F, F), (F, T, T), (F, F, T), (F, F, F)\} \text{ (1 in the GACC test)}$$

$$T_{EDGE:5} = \{(T, T, T), (F, T, F), (F, F, T), (F, F, F)\} \text{ (1 in the GACC test)}$$

$$T_{EDGE:6} = \{(T, T, F), (F, T, F), (F, F, T), (F, F, F)\} \text{ (1 in the GACC test)}$$

$$T_{EDGE:7} = \{(T, F, T), (F, T, F), (F, F, T), (F, F, F)\} \text{ (1 in the GACC test)}$$

$$T_{EDGE:8} = \{(T, F, F), (F, T, F), (F, F, T), (F, F, F)\} \text{ (2 in the GACC tests)}$$

4. (**Challenging!**) For the TriTyp program, complete the test sets for the following coverage criteria by filling in the “don’t care” values, ensuring reachability, and deriving the expected output. Download the program, compile it, and run it with your resulting test cases to verify correct outputs.

- Predicate Coverage (PC)
- Clause Coverage (CC)
- Combinatorial Coverage (CoC)
- Correlated Active Clause Coverage (CACC)

Instructor Solution Only

5. Repeat the prior exercise, but for the TestPat program in Chapter 2.

Instructor Solution Only

6. Repeat the prior exercise, but for the Quadratic program in Chapter 2.

Instructor Solution Only

Exercises, Section 3.4

Consider the `remove()` method from the Java `Iterator` interface. The `remove()` method has a complex precondition on the state of the `Iterator`, and the programmer can choose to detect violations of the precondition and report them as `IllegalStateException`.

1. Formalize the precondition.

Instructor Solution Only The problem with this formalization is that the ACC criteria are not defined on quantified predicates.

A different approach is to formalize the precondition without reference to the entire history. Since there are only two mutators in the `Iterator` interface, let A be the predicate that `next()` has been called and B be the predicate that `remove()` has not been called subsequently. Then the precondition is simply $A \wedge B$.

Both of these predicates ignore the fact that calling `next()` isn't sufficient by itself; it is also necessary for `next()` to actually return a value (instead of `NoSuchElementException`).

Finally, the behavior of the entire iterator, and of `remove()` in particular, is determined by whether the client attempts to modify the underlying collection while the iterator is in use. So, for example, a call to `next()`, followed by a modification of the underlying collection, followed by a call to `remove()` is, in many implementations of `Iterator`, going to result in `ConcurrentModificationException` being thrown.

For a more advanced exercise, consider the `ListIterator` interface, which is considerably more complex.

2. Find (or write) an implementation of an `Iterator`. The Java `Collection` classes are a good place to search.

Instructor Solution Only

3. Develop and run CACC tests on the implementation.

Instructor Solution Only

Exercises, Section 3.5

1. For the **Memory Seat** finite state machine, complete the test sets for the following coverage criteria by satisfying the predicates, ensuring reachability, and computing the expected output.

- Predicate Coverage

Instructor Solution Only

- Correlated Active Clause Coverage

Instructor Solution Only

- General Inactive Clause Coverage

Instructor Solution Only

2. Redraw Figure 3.5 to have fewer transitions, but more clauses. Specifically, nodes 1, 2, 4, and 5 each has four transitions to node 3. Rewrite these transitions to have only one transition from each of nodes 1, 2, 4, and 5 to node 3, and the clauses are connected by ORs. Then derive tests to satisfy CACC for the resulting predicates. How do these tests compare with the tests derived from the original graph?

(partial) **Instructor Solution Only**

3. Consider the following deterministic finite state machine:

Current State	Condition	Next State
Idle	$a \vee b$	Active
Active	$a \wedge b$	Idle
Active	$\neg b$	WindDown
WindDown	a	Idle

- (a) Draw the finite state machine.

Instructor Solution Only

- (b) This machine does not specify which conditions cause a state to transition back to itself. However, these conditions can be derived from the existing conditions. Derive the conditions under which each state will transition back to itself.

Instructor Solution Only

- (c) Find CACC tests for each transition from the Active state.

Instructor Solution Only

4. Pick a household appliance such as a watch, calculator, microwave, VCR, clock-radio or programmable thermostat. Draw the FSM that represents your appliance's behavior. Derive tests to satisfy Predicate Coverage, Correlated Active Clause Coverage, and General Inactive Clause Coverage.

Instructor Solution Only

5. Implement the memory seat FSM. Design an appropriate input language to your implementation and turn the tests derived for question 1 into test scripts. Run the tests.

Instructor Solution Only

Exercises, Section 3.6

Use functions (1) through (4) to answer the following questions.

1. $f = ab\bar{c} + \bar{a}b\bar{c}$
 2. $f = \bar{a}\bar{b}\bar{c}\bar{d} + abcd$
 3. $f = ab + \bar{a}\bar{b}c + \bar{a}\bar{b}\bar{c}$
 4. $f = \bar{a}\bar{c}\bar{d} + \bar{c}d + bcd$
- (a) Draw the Karnaugh maps for f and \bar{f} .
 - (b) Find the nonredundant prime implicant representation for f and \bar{f} .
 - (c) Give a test set that satisfies Implicant Coverage (IC) for f .
 - (d) Give a test set that satisfies Unique True Point Coverage (UTPC) for f .
 - (e) Give a test set that satisfies Corresponding Unique True Point and Near False Point Pair Coverage (CUTPNFP) for f .

Solution:

Solution for $f = ab\bar{c} + \bar{a}b\bar{c}$

(a) *Karnaugh map for f :*

		<i>a, b</i>			
		<i>00</i>	<i>01</i>	<i>11</i>	<i>10</i>
<i>c</i>	<i>0</i>		1	1	
<i>c</i>	<i>1</i>				

Karnaugh map for \bar{f} :

		<i>a, b</i>			
		<i>00</i>	<i>01</i>	<i>11</i>	<i>10</i>
<i>c</i>	<i>0</i>	1			1
<i>c</i>	<i>1</i>	1	1	1	1

(b) *Nonredundant prime implicant representation for f :*

$$f = b\bar{c}$$

Nonredundant prime implicant representation for \bar{f} :

$$\bar{f} = \bar{b} + c$$

Note that f is a function of b and c only; a is irrelevant.

- (c) *For IC we choose the nonredundant prime implicant representations. Other choices are possible, of course. This leaves three implicants $\{b\bar{c}, \bar{b}, c\}$ in f and \bar{f} collectively. Test set $\{xTF, xFT\}$ satisfies IC. Note that the second test, which is not a unique true point, satisfies both \bar{b} and c .*
- (d) *For UTPC we need three tests. Many possibilities; one is $\{xTF, xFF, xTT\}$. Again, values for a are irrelevant.*
- (e) *For CUTPNFP we take a unique true point for f , xTF , and pair it with all near false points. The result in this case is the same as for UTPC, namely, $\{xTF, xFF, xTT\}$.*

Solution:

Solution for $f = \bar{a}\bar{b}\bar{c}\bar{d} + abcd$

(a) Karnaugh map for f :

		a, b			
		00	01	11	10
cd	00	1			
	01				
	11			1	
	10				

Karnaugh map for \bar{f} :

		a, b			
		00	01	11	10
cd	00		1	1	1
	01	1	1	1	1
	11	1	1		1
	10	1	1	1	1

(b) Nonredundant prime implicant representation for f (Note: as given):

$$f = \bar{a}\bar{b}\bar{c}\bar{d} + abcd$$

Nonredundant prime implicant representation for \bar{f} :

$$\bar{f} = a\bar{b} + b\bar{c} + c\bar{d} + \bar{a}d$$

(c) For IC we choose the nonredundant prime implicant representations. Other choices are possible, of course. This leaves six implicants $\{\bar{a}\bar{b}\bar{c}\bar{d}, abcd, a\bar{b}, b\bar{c}, c\bar{d}, \bar{a}d\}$ in f and \bar{f} collectively. Test set $\{FFFF, TTTT, FTFF, FTFT\}$ satisfies IC. Note that the third and fourth tests, which are not unique true points, each satisfy two implicants.

(d) UTPC needs six tests. One possibility: $\{FFFF, TTTT, TTFF, TFFF, FFFT, FFTF\}$.

(e) For CUTPNFP we take unique true points for implicants in f , namely $FFFF$ and $TTTT$, and pair each with all near false points. The result is 10 tests. With $FFFF$, we pair $TFFF, FTFF, FFTF, FFFT$, and with $TTTT$, we pair $FTTT, TFTT, TTFT, TTTF$.

Although not part of this exercise, it is instructive to consider CUTPNFP for \bar{f} . In particular, it appears that one could select unique true points $TFFT, TTFF, FTTF$, and $FFFT$, none of which have any near false points. However, this set does not satisfy the CUTPNFP criterion. Instead, CUTPNFP requires that, if it possible to select a unique true point that has a near false point with respect to a given variable, then such a unique true point is chosen. It may be necessary to choose different unique true points to pair with false points with respect to each variable. To continue the example, for $a\bar{b}$, we pair $TFFF$ with $FFFF$ for a , and $TFTT$ with $TTTT$ for b . For $b\bar{c}$ we pair $FTFF$ with $FFFF$ for b , and $TTFT$ with $TTTT$ for c . For $c\bar{d}$ we pair $FFTF$ with $FFFF$ for c , and $TTTF$ with $TTTT$ for d . For $\bar{a}d$ we pair $FFFT$ with $FFFF$ for d , and $FTTT$ with $TTTT$ for a . The union of these results is exactly the same set of 10 test cases as derived for CUTPNFP for f .

Instructor Solution Only

Instructor Solution Only

5 Use the following predicates to answer questions (a) through (e). In the questions, “simplest” means “fewest number of variable references”.

- $W = (B \wedge \neg C \wedge \neg D)$
- $X = (B \wedge D) \vee (\neg B \neg D)$
- $Y = (A \wedge B)$
- $Z = (\neg B \wedge D)$

(a) Draw the Karnaugh map for the predicates. Put AB on the top and CD on the side. Label each cell with W , X , Y , and/or Z as appropriate.

Instructor Solution Only

(b) Write the simplest expression that describes all cells that have more than one definition.

Instructor Solution Only

(c) Write the simplest expression that describes all cells that have no definitions.

Instructor Solution Only

(d) Write the simplest expression that describes $X \vee Z$.

Instructor Solution Only

(e) Give a test set for expression X that uses each prime implicant once.

Instructor Solution Only

6 Develop a counterexample to show that CUTPNFP does not subsume UTPC. Hint: You might wish to start with the expression given in the text, $f = ab + cd$.

Instructor Solution Only

Exercises, Section 4.1

1. Answer the following questions for the method `search()` below:

```
public static int search (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//   else if element is in the list, return an index
//   of element in the list; else return -1
//   for example, search ([3,3,1], 3) = either 0 or 1
//   search ([1,7,5], 2) = -1
```

Base your answer on the following characteristic partitioning:

```
Characteristic: Location of element in list
  Block 1: element is first entry in list
  Block 2: element is last entry in list
  Block 3: element is in some position other than first or last
```

- (a) “Location of element in list” fails the disjointness property. Give an example that illustrates this.
Instructor Solution Only
- (b) “Location of element in list” fails the completeness property. Give an example that illustrates this.
Instructor Solution Only
- (c) Supply one or more new partitions that capture the intent of “Location of e in list” but do not suffer from completeness or disjointness problems.
Instructor Solution Only
2. Derive input space partitioning tests for the **GenericStack** class with the following method signatures:

- `public GenericStack ();`
- `public void Push (Object X);`
- `public Object Pop ();`
- `public boolean IsEmt ();`

Assume the usual semantics for the stack. Try to keep your partitioning simple, choose a small number of partitions and blocks.

- (a) Define characteristics of inputs
Instructor Solution Only
- (b) Partition the characteristics into blocks
Instructor Solution Only
- (c) Define values for the blocks
Instructor Solution Only

Exercises, Section 4.2

1. Enumerate all 64 tests to satisfy the All Combinations (ACoC) criterion for the second categorization of TriTyp's inputs in Table 4.2. Use the values in Table 4.3.

Solution:

$\{(2, 2, 2), (2, 2, 1), (2, 2, 0), (2, 2, -1),$
 $(2, 1, 2), (2, 1, 1), (2, 1, 0), (2, 1, -1),$
 $(2, 0, 2), (2, 0, 1), (2, 0, 0), (2, 0, -1),$
 $(2, -1, 2), (2, -1, 1), (2, -1, 0), (2, -1, -1),$
 $(1, 2, 2), (1, 2, 1), (1, 2, 0), (1, 2, -1),$
 $(1, 1, 2), (1, 1, 1), (1, 1, 0), (1, 1, -1),$
 $(1, 0, 2), (1, 0, 1), (1, 0, 0), (1, 0, -1),$
 $(1, -1, 2), (1, -1, 1), (1, -1, 0), (1, -1, -1),$
 $(0, 2, 2), (0, 2, 1), (0, 2, 0), (0, 2, -1),$
 $(0, 1, 2), (0, 1, 1), (0, 1, 0), (0, 1, -1),$
 $(0, 0, 2), (0, 0, 1), (0, 0, 0), (0, 0, -1),$
 $(0, -1, 2), (0, -1, 1), (0, -1, 0), (0, -1, -1),$
 $(-1, 2, 2), (-1, 2, 1), (-1, 2, 0), (-1, 2, -1),$
 $(-1, 1, 2), (-1, 1, 1), (-1, 1, 0), (-1, 1, -1),$
 $(-1, 0, 2), (-1, 0, 1), (-1, 0, 0), (-1, 0, -1),$
 $(-1, -1, 2), (-1, -1, 1), (-1, -1, 0), (-1, -1, -1)\}$

2. Enumerate all 16 tests to satisfy the Pair-Wise (PWC) criterion for the second categorization of TriTyp's inputs in Table 4.2. Use the values in Table 4.3.

Solution:

Note: Lots of possibilities here, as suggested by the ways in which pairs are chosen below.

$\{(2, 2, 2),$
 $(2, 1, 1),$
 $(2, 0, 0),$
 $(2, -1, -1),$
 $(1, 2, 1),$
 $(1, 1, 2),$
 $(1, 0, -1),$
 $(1, -1, 0),$
 $(0, 2, 0),$
 $(0, 1, -1),$
 $(0, 0, 2),$
 $(0, -1, 1),$
 $(-1, 2, -1),$
 $(-1, 1, 0),$

$(-1, 0, 1),$
 $(-1, -1, 2)\}$

3. Enumerate all 16 tests to satisfy the Multiple Base Choice (MBCC) criterion for the second categorization of TriTyp's inputs in Table 4.2. Use the values in Table 4.3.

Solution:

The text suggests both '2' and '1' base choices for side 1. (Other sides still have 1 base choice of '2'). This give two base tests: $(2, 2, 2)$ and $(1, 2, 2)$. According to the formula given in the text, we should get $2(\text{base}) + 6 + 6 + 6 = 20$ (not 18, as erroneously stated in the text) tests. However, 4 of these are redundant, so the result is 16. To clarify, we list all 20 tests, with the base choices listed first:

$\{(2, 2, 2), (1, 2, 2)$

$(2, 1, 2), (2, 0, 2), (2, -1, 2),$
 $(2, 2, 1), (2, 2, 0), (2, 2, -1),$
 $(1, 2, 2), (0, 2, 2), (-1, 2, 2),$
 $(1, 2, 1), (1, 2, 0), (1, 2, -1),$
 $(1, 1, 2), (1, 0, 2), (1, -1, 2),$
 $(2, 2, 2), (0, 2, 2), (-1, 2, 2)\}$

Here are the 16 nonredundant tests:

$\{(2, 2, 2), (1, 2, 2)$

$(2, 1, 2), (2, 0, 2), (2, -1, 2),$
 $(2, 2, 1), (2, 2, 0), (2, 2, -1),$
 $(0, 2, 2), (-1, 2, 2),$
 $(1, 2, 1), (1, 2, 0), (1, 2, -1),$
 $(1, 1, 2), (1, 0, 2), (1, -1, 2),$
 $\}$

4. Answer the following questions for the method `intersection()` below:

```
public Set intersection (Set s1, Set s2)
// Effects:  If s1 or s2 are null throw NullPointerException
//           else return a (non null) Set equal to the intersection
//           of Sets s1 and s2

Characteristic:  Type of s1
- s1 = null
- s1 = {}
- s1 has at least one element

Characteristic:  Relation between s1 and s2
- s1 and s2 represent the same set
- s1 is a subset of s2
- s2 is a subset of s1
- s1 and s2 do not have any elements in common
```

- (a) Does the partition “Type of s_1 ” satisfy the completeness property? If not, give a value for s_1 that does not fit in any block.

Instructor Solution Only

- (b) Does the partition “Type of s_1 ” satisfy the disjointness property? If not, give a value for s_1 that fits in more than one block.

Instructor Solution Only

- (c) Does the partition “Relation between s_1 and s_2 ” satisfy the completeness property? If not, give a pair of values for s_1 and s_2 that does not fit in any block.

Instructor Solution Only

- (d) Does the partition “Relation between s_1 and s_2 ” satisfy the disjointness property? If not, give a pair of values for s_1 and s_2 that fits in more than one block.

Instructor Solution Only

- (e) If the “Base Choice” criterion were applied to the two partitions (exactly as written), how many test requirements would result?

Instructor Solution Only

5. Derive input space partitioning tests for the **BoundedQueue** class with the following signature:

- `public BoundedQueue (int capacity);`
- `public void Enqueue (Object X);`
- `public Object Dequeue ();`
- `public boolean IsEmpty ();`
- `public boolean IsFull ();`

Assume the usual semantics for a queue with a fixed, maximal capacity. Try to keep your partitioning simple—choose a small number of partitions and blocks.

- (a) Identify all of the variables. Don’t forget the state variables.

Instructor Solution Only

- (b) Identify several characteristics that suggest partitions.

Instructor Solution Only

- (c) Identify the blocks in the partition for each characteristic. Designate one block in each partition as the “Base” block.

Instructor Solution Only

- (d) Define values for the blocks.

Instructor Solution Only

- (e) Define a test set that satisfies Base Choice (BCC) coverage.

Instructor Solution Only

6. Develop a set of characteristics and accompanying partitions for the pattern checking procedure (the method *pat()* in Figure 2.21 in Chapter 2).
- (a) Develop tests to satisfy the Base Choice criterion. Your tests should have both inputs and expected outputs.

Solution:

We consider a fairly straightforward set of characteristics.

- **subject length.**
 - $a_1 : 0$
 - $a_2 : 1$
 - $a_3 : 2$
 - $a_4 : > 2$ // base
- **pattern length.**
 - $b_1 : 0$
 - $b_2 : 1$
 - $b_3 : 2$ // base
 - $b_4 : > 2$
- **pattern occurrences in subject.**
 - $c_1 : 0$
 - $c_2 : 1$ // base
 - $c_3 : > 1$
- **pattern at start of subject.**
 - $d_1 : true$
 - $d_2 : false$ // base
- **pattern at end of subject.**
 - $e_1 : true$
 - $e_2 : false$ // base
- **Relation of pattern length to subject length**
 - $f_1 : pattern.length < subject.length$ // base
 - $f_2 : pattern.length = subject.length$
 - $f_3 : pattern.length > subject.length$

For these 6 characteristics, we get one base test ($a_4, b_2, c_2, d_2, e_2, f_1$) plus $3 + 3 + 2 + 1 + 1 + 2 = 12$ variations for a total of 13 tests. Possible tests (subject, pat, expected output) are:

$(a_4, b_3, c_2, d_2, e_2, f_1) : (abcd, bc, 1)$ // base

Tests for a:

$(a_1, b_3, c_1, d_2, e_2, f_3) : ("", bc, -1) // \text{note change to } c, f$

$(a_2, b_3, c_1, d_2, e_2, f_3) : (a, bc, -1) // \text{note change to } c, f$

$(a_3, b_3, c_2, d_1, e_1, f_2) : (ab, ab, 0) // \text{note change to } d, e, f$

Tests for b:

$(a_4, b_1, c_1, d_2, e_2, f_1) : (abcd, "", -1) // \text{note change to } c$

$(a_4, b_2, c_1, d_2, e_2, f_1) : (abcd, b, 1)$

$(a_4, b_3, c_2, d_1, e_1, f_2) : (abcd, abcd, 0) // \text{note change to } d, e, f$

Tests for c:

$(a_4, b_3, c_1, d_2, e_2, f_1) : (abcd, bd, -1)$

$(a_4, b_3, c_3, d_2, e_2, f_1) : (abcabcd, bc, 1)$

Tests for d:

$(a_4, b_3, c_2, d_1, e_2, f_1) : (abcd, ab, 0)$

Tests for e:

$(a_4, b_3, c_2, d_2, e_1, f_1) : (abcd, cd, 2)$

Tests for f:

$(a_3, b_3, c_2, d_1, e_1, f_2) : (ab, ab, 0) // \text{note change to } a, d, e$

$(a_2, b_3, c_2, d_2, e_2, f_3) : (a, bc, -1) // \text{note change to } a, c$

- (b) Analyze your tests with respect to the data flow test sets developed in Chapter 2. How well does input space partitioning do?

Solution:

Paths for the tests above are:

$(abcd, bc, 1): [1, 2, 3, 4, 10, 3, 4, 5, 6, 7, 9, 6, 10, 3, 11]$

$("", bc, -1): [1, 2, 3, 11]$

$(a, bc, -1): [1, 2, 3, 11]$

$(ab, ab, 0): [1, 2, 3, 4, 5, 6, 7, 9, 6, 10, 3, 11]$

$(abcd, "", -1): \text{Exception: } \text{ArrayIndexOutOfBoundsException}$

$(abcd, b, 1): [1, 2, 3, 4, 10, 3, 4, 5, 6, 10, 3, 11]$

$(abcd, abcd, 0): [1, 2, 3, 4, 5, 6, 7, 9, 6, 7, 9, 6, 7, 9, 6, 10, 3, 11]$

$(abcd, bd, -1): [1, 2, 3, 4, 10, 3, 4, 5, 6, 7, 8, 10, 3, 4, 10, 3, 11]$

$(abcabcd, bc, 1): [1, 2, 3, 4, 10, 3, 4, 5, 6, 7, 9, 6, 10, 3, 11]$

$(abcd, ab, 0): [1, 2, 3, 4, 5, 6, 7, 9, 6, 10, 3, 11]$

$(abcd, cd, 2): [1, 2, 3, 4, 10, 3, 4, 10, 3, 4, 5, 6, 7, 9, 6, 10, 3, 11]$

$(ab, ab, 0): [1, 2, 3, 4, 5, 6, 7, 9, 6, 10, 3, 11]$

$(a, bc, -1): [1, 2, 3, 11]$

There are a number of things to notice in the comparison. First, the exceptional return on the first test for b is an advantage for the input domain model approach. However, in terms of du-paths covered by the input domain model tests, the following du-paths are missing:

[2, 3, 4, 5, 6, 7, 8]

[2, 3, 4, 5, 6, 7, 8]

[8, 10, 3, 11]

[2, 3, 4, 5, 6, 10]

[2, 3, 4, 5, 6, 7, 8, 10]

[9, 6, 7, 8]

[1, 2, 3, 4, 5, 6, 7, 8]

These results suggest that the two approaches are complementary.

Exercises, Section 5.1.1

1. Consider how often the idea of covering nodes and edges pops up in software testing. Write a short essay to explain this.

Instructor Solution Only

2. Just as with graphs, it is possible to generate an infinite number of tests from a grammar. How and what makes this possible?

Instructor Solution Only

Exercises, Section 5.1.2

1. Define mutation score.

Instructor Solution Only

2. How is the mutation score related to coverage from Chapter 1?

Instructor Solution Only

3. Consider the stream BNF in Section 5.1.1 and the ground string “B 10 06.27.94.”
Give three valid and three invalid mutants of the string.

Instructor Solution Only

Exercises, Section 5.2

1. Provide reachability conditions, infection conditions, propagation conditions, and test case values to kill mutants 2, 4, 5, and 6 in Figure 5.1.

Instructor Solution Only

2. Answer questions (a) through (d) for the mutant in the two methods, `findVal()` and `sum()`.

- (a) If possible, find a test input that does **not** reach the mutant.

Instructor Solution Only

- (b) If possible, find a test input that satisfies reachability but **not infection** for the mutant.

Instructor Solution Only

- (c) If possible, find a test input that satisfies infection, but **not propagation** for the mutant.

Instructor Solution Only

- (d) If possible, find a test input that strongly kills mutant m.

Instructor Solution Only

<pre>//Effects: If numbers null throw NullPointerException // else return LAST occurrence of val in numbers[] // If val not in numbers[] return -1 1. public static int findVal (int numbers[], int val) 2. { 3. int findVal = -1; 4. 5. for (int i=0; i<numbers.length; i++) 5'.// for (int i=(0+1); i<numbers.length; i++) 6. if (numbers [i] == val) 7. findVal = i; 8. return (findVal); 9. }</pre>	<pre>//Effects: If x null throw NullPointerException // else return the sum of the values in x 1. public static int sum (int[] x) 2. { 3. int s = 0; 4. for (int i=0; i < x.length; i++) } 5. { 6. s = s + x[i]; 6'. // s = s - x[i]; //AOR 7. } 8. return s; 9. }</pre>
---	---

3. Refer to the `TestPat` program in Chapter 2. Consider Mutant A and Mutant B given below. Answer the following questions for each mutant.

Mutants **A** and **B**:

- (a) `while (isPat == false && isub + patternLen - 1 < subjectLen) // Original`
`while (isPat == false && isub + patternLen - 0 < subjectLen) // Mutant A`

- (b) `isPat = false; // Original`
`isPat = true; // Mutant B`

Instructor Solution Only

- (a) If possible, find test cases that do **not reach** the mutants.

Instructor Solution Only

- (b) If possible, find test inputs that satisfy reachability but **not infection** for the mutants.

Instructor Solution Only

- (c) If possible, find test inputs that satisfy infection, but **not propagation** for the mutants.

Instructor Solution Only

- (d) If possible, find test inputs that strongly kill the mutants.

Instructor Solution Only

4. Why does it make sense to remove ineffective test cases?

Instructor Solution Only

5. Define 12 mutants for the following method *cal()* using the effective mutation operators given previously. Try to use each mutation operator at least once. Approximately how many mutants do you think there would be if all mutants for *cal* were created?

```
public static int cal (int month1, int day1, int month2,
                    int day2, int year)
{
//*****
// Calculate the number of Days between the two given days in
// the same year.
// preconditions : day1 and day2 must be in same year
//                1 <= month1, month2 <= 12
//                1 <= day1, day2 <= 31
//                month1 <= month2
//                The range for year: 1 ... 10000
//*****
    int numDays;

    if (month2 == month1) // in the same month
        numDays = day2 - day1;
    else
    {
        // Skip month 0.
        int daysIn[] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
        // Are we in a leap year?
        int m4 = year % 4;
        int m100 = year % 100;
        int m400 = year % 400;
        if ((m4 != 0) || ((m100 == 0) && (m400 != 0)))
            daysIn[2] = 28;
        else
            daysIn[2] = 29;

        // start with days in the two months
        numDays = day2 + (daysIn[month1] - day1);

        // add the days in the intervening months
        for (int i = month1 + 1; i <= month2-1; i++)
            numDays = daysIn[i] + numDays;
    }
    return (numDays);
}
```

Instructor Solution Only

6. Define 12 mutants for the following method *power()* using the effective mutation operators given previously. Try to use each mutation operator at least once. Approximately how many mutants do you think there would be if all mutants for *power()* were created?

Instructor Solution Only

```
public static int power (int left, int right)
{
  //*****
  // Raises Left to the power of Right
  // precondition : Right >= 0
  // postcondition: Returns Left**Right
  //*****
  int rslt;
  rslt = Left;
  if (Right == 0)
  {
    rslt = 1;
  }
  else
  {
    for (int i = 2; i <= right; i++)
      rslt = rslt * left;
  }
  return (rslt);
}
```

Instructor Solution Only

7. The fundamental premise was stated as: “*In practice, if the software contains a fault, there will usually be a set of mutants that can be killed only by a test case that also detects that fault.*”

- (a) Give a brief argument **in support of** the fundamental mutation premise.

Instructor Solution Only

- (b) Give a brief argument **against** the fundamental mutation premise.

Instructor Solution Only

8. Try to design mutation operators that subsume Combinatorial Coverage. Why wouldn't we want such an operator?

Instructor Solution Only

9. Look online for the tool Jester, which is based on JUnit¹. Based on your reading, evaluate Jester as a mutation-testing tool.

Instructor Solution Only

10. Download and install the Java mutation tool *muJava*: <http://cs.gmu.edu/~offutt/~mujava/>. Enclose the method *cal()* from question 5 inside a class, and use muJava to test *cal()*. Note that a test case is a method call to *cal()*.

¹Jester's web page is <http://jester.sourceforge.net/>

(a) How many mutants are there?

Instructor Solution Only

(b) How many test cases do you need to kill the mutants?

Instructor Solution Only

(c) What mutation score were you able to achieve without analyzing for equivalent mutants?

Instructor Solution Only

(d) How many equivalent mutants are there?

Instructor Solution Only

Instructor Solution Only

Exercises, Section 5.4

1. (Challenging!) Find or write a small SMV specification and a corresponding Java implementation. Restate the program logic in SPEC assertions. Mutate the assertions systematically, and collect the traces from (nonequivalent) mutants. Use these traces to test the implementation. **Instructor Solution Only**

Exercises, Section 5.5

1. Generate tests to satisfy TSC for the bank example grammar based on the BNF in Section 5.5.1. Try **not** to satisfy PDC.

Instructor Solution Only

2. Generate tests to satisfy PDC for the bank example grammar.

Instructor Solution Only

3. Consider the following BNF with start symbol A:

```

A ::= B"@C".B
B ::= BL | L
C ::= B | B".B
L ::= "a" | "b" | "c" | ... | "y" | "z"

```

and the following six possible test cases:

```

t1 = a@a.a
t2 = aa.bb@cc.dd
t3 = mm@pp
t4 = aaa@bb.cc.dd
t5 = bill
t6 = @x.y

```

For each of the six tests, (1) identify the test sequence as either “in” the BNF, and give a derivation, or (2) identify the test sequence as “out” of the BNF, and give a mutant derivation that results in that test. (Use only one mutation per test, and use it only one time per test). **Instructor Solution Only**

4. Provide a BNF description of the inputs to the *cal()* method in the homework set for Section 5.2.2. Succinctly describe any requirements or constraints on the inputs that are hard to model with the BNF.
5. Answer questions (a) through (c) for the following grammar.

```

val    ::= number | val pair
number ::= digit+
pair   ::= number op | number pair op
op     ::= "+" | "-" | "*" | "/"
digit  ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

```

Also consider the following mutation, which adds an additional rule to the grammar:

```

pair ::= number op | number pair op | op number

```

- (a) Which of the following strings can be generated by the (unmutated) grammar?

```

42
4 2
4 + 2
4 2 +
4 2 7 - *
4 2 - 7 *
4 2 - 7 * +

```

Instructor Solution Only

- (b) Find a string that is generated by the mutated grammar, but not by the original grammar.

Instructor Solution Only

- (c) (*Challenging*) Find a string whose generation uses the new rule in the mutant grammar, but is also in the original grammar. Demonstrate your answer by giving the two relevant derivations.

Instructor Solution Only

6. Answer questions (a) and (b) for the following grammar.

```

phoneNumber ::= exchangePart dash numberPart
exchangePart ::= special zeroOrSpecial other
numberPart  ::= ordinary4
ordinary    ::= zero | special | other
zeroOrSpecial ::= zero | special
zero       ::= "0"
special    ::= "1" | "2"
other      ::= "3" | "4" | "5" | "6" | "7" | "8" | "9"
dash      ::= "-"

```

- (a) Classify the following as either phoneNumbers (or not). For non-phone numbers, indicate the problem.

- 123-4567
- 012-3456
- 109-1212
- 346-9900
- 113-1111

Instructor Solution Only

- (b) Consider the following mutation of the grammar:

```

exchangePart ::= special ordinary other

```

If possible, identify a string that appears in the mutated grammar but not in the original grammar, another string that is in the original but not the mutated, and a third string that is in both.

Instructor Solution Only

7. Java provides a package, *java.util.regex*, to manipulate regular expressions. Write a regular expression for URLs and then evaluate a set of URLs against your regular expression. This assignment involves programming, since input structure testing without automation is pointless.

- (a) Write (or find) a regular expression for a URL. Your regular expression does not need to be so general that it accounts for every possible URL, but give your best effort (for example "*" will not be considered a good effort). You are strongly encouraged to do some web surfing to find some candidate regular expressions. One suggestion is to visit the *Regular Expression Library*.

Instructor Solution Only

- (b) Collect a set of URLs from a small web site (such as a set of course web pages). Your set needs to contain at least 20 (different) URLs. Use the *java.util.regex* package to validate each URL against your regular expression.

Instructor Solution Only

- (c) Construct a valid URL that is not valid with respect to your regular expression (and show this with the appropriate *java.util.regex* call). If you have done an outstanding job in part 1, explain why your regular expression does not have any such URLs.

Instructor Solution Only

8. Why is the equivalent mutant problem solvable for BNF grammars but not for program-based mutation? (Hint: The answer to this question is based on some fairly subtle theory.)

Instructor Solution Only

Changes to the Solution Manual

This section documents changes so that users can download a fresh copy and quickly locate additions and corrections.

- 19-Sept-2011: Corrected solution to Exercise 1.2 Number 3e, `countPositive()` (Instructor only)
- 16-Jun-2011: Corrected solution to Exercise 2.4 Number 1 (page 73).
- 23-Feb-2011: Corrected solution to Exercise 2.2.3 Number 1 (page 51). For Graph III, the solutions to parts (e) and (f) were incorrect.
- 12-Jan-2011: Corrected solution Section 2.1, number 1.
- 16-Feb-2011: Applied page 51 errata correction (Exercise 2.2.3, number 1) to solution manuals.
- 12-Jan-2011: Corrected solution Section 2.1, number 1.
- 27-Dec-2010: Added clarification to the answer to 1.2, #3(c)
- 7-Dec-2010: Corrected the answer to 5.5, #6
- 5-Nov-2010: Added solutions to several additional questions (2.7:1, 5.2:3, 5.2:6, 5.2:7, 5.2:10, and 5.5:7).
- 3-Nov-2010: Clarification of RICC tests for question 2, Section 3.3 (Instructor only)
- 2-Nov-2010: Clarification of ambiguity to question 3, Section 3.5 (Instructor only)
- 10-Aug-2009: Correction to question 4, Section 3.6, Karnaugh map (Instructor only)
- 17-Oct-2008: Added an answer to Section 3.5, question 4 (Instructor only)
- 17-Oct-2008: Added an answer to Section 5.1.1, question 1 (Instructor only)
- 13-Oct-2008: Correction to question 3, Section 3.3, Edge Coverage