Introduction to Software Testing
Edition 2

Paul Ammann and Jeff Offutt

Solutions to Exercises

Student Version

April 18, 2017

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

— Chinese proverb

Introductory Note

This document contains the work-in-progress solutions for the second edition of the text. The goal is to keep the solution manuals synchronized with the textbook so that there are no "TBD" solutions, as persisted in the first edition for many years.

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.

Chapter 1

Exercises, Chapter 1

1. What are some 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. What is the difference between software **fault** and software **failure**?

   Instructor Solution Only

3. What do we mean by "*level 3 thinking is that the purpose of testing is to reduce risk?*" What risk? Can we reduce the risk to zero?

   Instructor Solution Only

4. 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. Describe 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

5. Below are four faulty programs. Each includes test inputs that result in failure. Answer the following questions about each program.

```
/**
 * Find last index of element
 *
 * @param x array to search
 * @param y value to look for
 * @return last index of y in x; -1 if absent
 * @throws NullPointerException if x is null
 */
public int findLast (int[] x, int y)
{
   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
// Book website: FindLast.java
// Book website: FindLastTest.java
```

```
/**
 * Find last index of zero
 *
 * @param x array to search
 *
 * @return last index of 0 in x; -1 if absent
 * @throws NullPointerException if x is null
 */
public static int lastZero (int[] x)
{
   for (int i = 0; i < x.length; i++)
   {
      if (x[i] == 0)
      {
         return i;
      }
   }
   return -1;
}
// test:  x = [0, 1, 0]; Expected = 2
// Book website: LastZero.java
// Book website: LastZeroTest.java
```

```
/**
 * Count positive elements
 *
 * @param x array to search
 * @return count of positive elements in x
 * @throws NullPointerException if x is null
 */
public int countPositive (int[] 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]; Expcted = 2
// Book website: CountPositive.java
// Book website: CountPositiveTest.java
```

```
/**
 * Count odd or postive elements
 *
 * @param x array to search
 * @return count of odd/positive values in x
 * @throws NullPointerException if x is null
 */
public static int oddOrPos(int[] x)
{
   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
// Book website: OddOrPos.java
// Book website: OddOrPosTest.java
```

(a) Explain what is wrong with the given code. Describe the fault precisely by proposing a modification to the code.

(b) If possible, give a test case that does **not** execute the fault. If not, briefly explain why not.

(c) If possible, give a test case that executes the fault, but does **not** result in an error state. If not, briefly explain why not.

(d) If possible give a test case that results in an error state, but **not** a failure. Hint: Don't forget about the program counter. If not, briefly explain why not.

(e) For the given test case, describe the first error state. Be sure to describe the complete state.

(f) Implement your repair and verify that the given test now produces the expected output. Submit a screen printout or other evidence that your new program works.

`findLast()`

Instructor Solution Only

`lastZero()`

Solution:
*This problem brings up a subtle issue with respect to faults and failures: there are always multiple ways to fix a fault. Which states are error states depends on which alternate program is chosen as "correct". While this may sound mysterious, it isn't. In fact, any time a programmer uses a debugger and decides that a variable has the "wrong" value, she is implicitly also choosing alternate code. We illustrate this point by giving two answers for part (a) below (**v1** and **v2**), and then carrying both versions through the subsequent parts of the answer. Note how the answers to the sub-questions differ for solution **v1** and solution **v2**.*

***Instructor note:** This exercise can lead to a very interesting in-class discussion, with deep insights into the location-based fault model. Ultimately, the model's foundations are traditional program verification as introduced by Hoare and Dijkstra. Specifically, code is just code. It can't be "right" or "wrong" until one assigns (or derives) expected behaviors to it with preconditions and postconditions. No one ever does this formally in practice, but* **every** *programmer does this informally* **every** *time she identifies code as faulty. This discussion is clearly beyond the scope of this text.*

(a) *The incorrect behavior is that the method returns the index of the **first** occurrence of 0, not the last. The faulty version of the method starts at the beginning and searches forward until it finds a 0, then returns its index. There are many possible ways to fix `lastZero()`; we describe two here. One is to invert the loop so that we search from high to low (version **v1** below). Another (version **v2**) is to keep searching the entire array, even after finding the 0, and storing the most recent index found until the search is finished. Solution **v1** only requires changing the for-loop statement, and is more efficient since only part of the array may need to be searched. Solution **v2** requires three separate changes and always requires searching the entire array. Both solutions are reasonable.*

__v1__: The for-loop is backwards. It starts from the beginning and returns the index of the first 0 found. It should start at the end and count down. The proposed repair is:

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

*__v2__: The statement inside the loop returns the index the first time it is reached. The loop should continue until the **last** occurrence of 0 is found. To do this requires an additional variable. The proposed repair is:*

```
int index = -1; // Added before the loop
index = i; // Replaces return statement inside the loop body
return index; // Replaces return statement after the loop
```

(b) *__v1__: All inputs start the loop, so all inputs execute the fault–even the null input.*

__v2__: All inputs "execute" the missing initialization added before the loop. Hence, all inputs execute the fault.

(c) *__v1__: All inputs result in an error state.*

*If **x** is null, execution continues beyond the initialization of **i** in the for-loop in the faulty program. The exception isn't raised until the loop predicate is evaluated. In the repaired*

version, the exception is raised during the initialization of `i`*. Hence, the faulty program has an extra state in its execution, and that extra state is an error state.*

If the loop is executed zero or one times, high-to-low and low-to-high evaluation are the same. However, 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.)*

v2*: All inputs result in an error state, even the null value for* `x`*. The reason is the difference between the second state in the original program and the second state in the proposed repair. To make this specific, consider an execution with an arbitrary value of* `x`*.*

Second State Original:	*x = ...*
	i = 0
	PC = just after `i= 0;`
Second State Repair:	*x = ...*
	index = -1
	PC = just before the for loop

Two points merit notice: First, the missing code means that there is at least one state variable missing in the execution of the original program. Hence, all states after the missing code are, by definition, error states. Second, once the variable `index` *is introduced, the states of the two programs are no longer defined by the same variables. Again, this means that the states after the missing code are, by definition, error states.*

(d) ***v1****: Even though all executions contain error states, the program does return the correct result in many cases. For example:*

Input:	*x = [1, 0, 3]*
Expected Output:	*1*
Actual Output:	*1*

v2*: As noted in part (c) above, all executions contain error states. But the faulty code still computes the correct outputs in cases such as the one listed above in the solution for* ***v1****.*

(e) ***v1****: 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`*. 0 and* `x.length-1` *are different unless* `x` *contains exactly one value. Hence, the first error state is encountered immediately after the initialization of* `i` *in the for-statement.*

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;`

v2: *As noted in part (c) above, the first error state occurs immediately after the missing code is "executed."*

12

`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.*

Input:	*x = []*
Expected Output:	*0*
Actual Output:	*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.*

Input:	*x = [1, 2, 3]*
Expected Output:	*3*
Actual Output:	*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)

Input:	*x = [-3, -2, 0, 1, 4]*
Expected Output:	*3*
Actual Output:	*2*

First Error State:
x = [-3, -2, 0, 1, 4]
i = 0;
count = 0;
PC = at end of `if` *statement, instead of just before* `count++`

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 difficult to analyze errors when the the PC has the wrong value!*

6. Answer question (a) or (b), **but not both**, depending on your background.

   (a) If you do, or have, worked for a software development company, what level of test maturity do you think the company worked at? (0: testing=debugging, 1: testing shows correctness, 2: testing shows the program doesn't work, 3: testing reduces risk, 4: testing is a mental discipline about quality).

   Solution:
   There is no "correct" solution for this exercise. The goal is to get students to reflect on the technical culture with respect to testing at their place of work.

   (b) If you have **never** worked for a software development company, what level of test maturity do you think that **you** have? (0: testing=debugging, 1: testing shows correctness, 2: testing shows the program doesn't work, 3: testing reduces risk, 4: testing is a mental discipline about quality).

   Solution:
   Again, there is no "correct" solution for this exercise. The goal is to get students to reflect on the personal level technical competence with respect to testing.

7. Consider the following three example classes. These are OO faults taken from Joshua Bloch's *Effective Java,* Second Edition. Answer the following questions about each.

```
class Vehicle implements Cloneable
{
   private int x;
   public Vehicle (int y) { x = y;}
   public Object clone()
   {
      Object result = new Vehicle (this.x);
      // Location "A"
      return result;
   }
   // other methods omitted
}
class Truck extends Vehicle
{
   private int y;
   public Truck (int z) { super (z); y = z;}
   public Object clone()
   {
      Object result = super.clone();
      // Location "B"
      ((Truck) result).y = this.y;  // throws ClassCastException
      return result;
   }
   // other methods omitted
}
// Test: Truck suv = new Truck (4); Truck co = suv.clone()
//       Expected: suv.x = co.x; suv.getClass() = co.getClass()

Note: Revelant to Bloch, Item 11 page 54.
Book website: Vehicle.java, Truck.java, CloneTest.java
```

```
public class BigDecimalTest
{
  BigDecimal x = new BigDecimal ("1.0");
  BigDecimal y = new BigDecimal ("1.00");
  // Fact:  !x.equals (y), but x.compareTo (y) == 0

  Set <BigDecimal> BigDecimalTree = new TreeSet <BigDecimal> ();
  BigDecimalTree.add (x);
  BigDecimalTree.add (y);
  // TreeSet uses compareTo(), so BigDecimalTree now has 1 element

  Set <BigDecimal> BigDecimalHash = new HashSet <BigDecimal> ();
  BigDecimalHash.add (x);
  BigDecimalHash.add (y);
  // HashSet uses equals(), so BigDecimalHash now has 2 elements
}
// Test: System.out.println ("BigDecimalTree = " + BigDecimalTree);
//       System.out.println ("BigDecimalHash = " + BigDecimalHash);
//       Expected: BigDecimalTree = 1; BigDecimalHash = 1

// See Java Doc for add() in Set Interface
// The problem is that in BigDecimal, equals() and compareTo()
// are inconsistent. Let's suppose we decide that compareTo() is correct,
// and that equals()is faulty.

Note: Revelant to Bloch, Item 12 page 62.
Book website: BigDecimalTest.java
```

```
class Point
{
   private int x; private int y;
   public Point (int x, int y) { this.x=x; this.y=y; }

   @Override public boolean equals (Object o)
   {
      // Location A
      if (!(o instanceof Point)) return false;
      Point p = (Point) o;
      return (p.x == this.x) && (p.y == this.y);
   }
}
class ColorPoint extends Point
{
   private Color color;
   // Fault: Superclass instantiable; subclass state extended

   public ColorPoint (int x, int y, Color color)
   {
      super (x,y);
      this.color=color;
   }
   @Override public boolean equals (Object o)
   {
      // Location B
      if (!(o instanceof ColorPoint)) return false;
      ColorPoint cp = (ColorPoint) o;
      return (super.equals(cp) &&  (cp.color == this.color));
   }
   // Tests:
      Point p  = new Point (1,2);
      ColorPoint cp1   = new ColorPoint (1,2,RED);
      ColorPoint cp2   = new ColorPoint (1,2,BLUE);
      p.equals (cp1);    // Test 1: Result = true;
      cp1.equals (p);    // Test 2: Result = false;
      cp1.equals (cp2);  // Test 3: Result = false;
   // Expected: p.equals (cp1) = true; cp1.equals (p) = true,
   //         cp1.equals (cp2) = false

Note: Relevant to Bloch, Item 17 page 87.
Book website: Point.java, ColorPoint.java, PointTest.java
```

(a) Explain what is wrong with the given code. Describe the fault precisely by proposing a modification to the code.

(b) If possible, give a test case that does **not** execute the fault. If not, briefly explain why not.

(c) If possible, give a test case that executes the fault, but does **not** result in an error state. If not, briefly explain why not.

(d) If possible give a test case that results in an error, but **not** a failure. If not, briefly explain why not. Hint: Don't forget about the program counter.

(e) In the given code, describe the first error state. Be sure to describe the complete state.

(f) Implement your repair and verify that the given test now produces the expected output. Submit a screen printout or other evidence that your new program works.

`clone()`

Instructor Solution Only

`compareTo() and equals() inconsistancy`

Solution:

(a) As noted in the exercise, `BigDecimal's, equals()` *and* `compareTo()` *methods are inconsistent.* `BigDecimal equals()` *requires the values to be equal in value and scale (2.0 is not equal to 2.00), while* `BigDecimal compareTo()` *only checks that the numbers are equal in value, not scale (2.0 is equal to 2.00). In the given code, the* `Hashset add()` *method uses* `BigDecimal equals()` *and* `BigDecimal hashCode()` *to evaluate elements.* `BigDecimal` *Treeset* `add()` *uses* `BigDecimal compareTo()` *to evaluate elements.*

There is a subtle point here, in that the `BigDecimal hashCode()` *method is required to be consistent with the* `BigDecimal equals()` *method. Hence, if we decide to change the semantics of* `equals()`*, we are also required to redefine* `hashCode()`*.*

Bottom line: If we decide the `BigDecimal equals()` *is faulty, we also must agree that* `BigDecimal hashCode()` *is faulty.*

The fault is executed during `BigDecimalHash.add()` *calls, which always result in a call to* `hashCode()`*, and sometimes result in a call to* `equals()`*.*

To formulate sensible answers to this problem, we have to make a decision about the level of abstraction at which we define state. For the purpose of this exercise, we'll assume that we have access to the state represented by variables in the given code, but do not have access to state internal to BigDecimal or the sets.

(b) All tests that involve calls to `equals()` *and/or* `hashCode()` *execute the fault. In terms of the given code, this means calls to certain* `HashSet` *methods.*

(c) Here are two example test cases that execute the fault but don't result an observable error.

i. Adding elements that have the same value and scale:

Input:	*BigDecimal a = new BigDecimal("5.0");* *BigDecimal b = new BigDecimal("5.0");* *Set<BigDecimal>BDTree = new TreeSet <BigDecimal> ();* *BDTree.add(a);* *BDTree.add(b);* *Set<BigDecimal>BDHash = new HashSet <BigDecimal> ();* *BDHash.add(a);* *BDHash.add(b);*
Expected Output:	*System.out.println(BDTree); //[5.0]* *System.out.println(BDHash); //[5.0]*
Actual Output:	*System.out.println(BDTree); //[5.0]* *System.out.println(BDHash); //[5.0]*

ii. Adding different values:

Input:	*BigDecimal a = new BigDecimal("6.0");* *BigDecimal b = new BigDecimal("7.00");* *Set<BigDecimal>BDTree = new TreeSet <BigDecimal> ();* *BDTree.add(a);*

BDTree.add(b);
Set<BigDecimal>BDHash = new HashSet <BigDecimal> ();
BDHash.add(a);
BDHash.add(b);

Expected Output: *System.out.println(BDTree); //[6.0, 7.00]*
System.out.println(BDHash); //[6.0, 7.00]

Actual Output: *System.out.println(BDTree); //[6.0, 7.00]*
System.out.println(BDHash); //[6.0, 7.00]

(d) An error state in the given context means that a `HashSet` *object contains an object it should not, or does not contain an object it should. Hence, every error state results in failure if we call an appropriate observer on the* `HashSet`*.*

(e) When the PC is just past s.add(y), s now contains two values, 1.0 and 1.00.

(f) The only way to fix the fault is to change how `equals()` *evaluates numbers for* `BigDecimal`*. Since* `BigDecimal` *is a part of Java API, there is no way to do this without breaking client code.*

`equals() inconsistency with inheritance`

Solution:

(a) *The equals() method in the subclass ColorPoint is not consistent with its superclass Point* `equals()` *method. One of the aspects of the equals contract is symmetry of evaluation between two objects.* `x.equals(y)` *should produce the same result as* `y.equals(x)`.
It is not possible to have a correctly implemented equals method in an instantiable subclass that contains an overriding equals method and more client visible state than the instantiable superclass.

(b) *All uses of* `equals()` *in Colorpoint would execute the fault.*

(c) *Create two ColorPoint objects and evalute the objects using their equals methods. The fault would be executed, but since the classes are the same, there is no error state. Note that the only time there is a problem is when both* `ColorPoint` *and* `Point` *objects exist in the same computation.*

Input:	*cp1.equals(cp2)*
	cp2.equals(cp1)
Expected Output:	*false, false*
Actual Output:	*false, false*

(d) *Not possible in the given situation.*

(e) *When PC is after cp1.equals(p), the result is false and breaks the expected symmetry of p.equals(cp1) result equaling cp1.equals(p) result.*

(f) *There is no way to fix the current class setup and maintain all of the following three properities: symmetry, transitivity, and substitution principle. One way out of this conundrum is to replace inheritance with composition (see Bloch for a defense of this approach, Item 16 page 81). An alternative is to sacrifice the substitution principle. (See Wagner's Effective C# for a defense of this approach, and also Bloch for a critique.)*

Chapter 2

Exercises, Chapter 2

1. 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.

2. Answer question (a) or (b), **but not both**, depending on your background.

   (a) If you do, or have, worked for a software development company, how much effort did your testing / QA team put into each of the four test activities? (test design, automation, execution, evaluation)

   (b) If you have **never** worked for a software development company, which of the four test activities do you think you are best qualified for? (test design, automation, execution, evaluation)

   Instructor Solution Only

Chapter 3

Exercises, Chapter 3

1. Why do testers automate tests? 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. Give a one or two paragraph explanation for how the **inheritance** hierarchy can affect controllability and observability.

   Instructor Solution Only

3. Develop JUnit tests for the `BoundedQueue` class. A compilable version is available on the book website in the file `BoundedQueue.java`. Make sure your tests check every method, but we will not evaluate the quality of your test designs and do not expect you to satisfy any test criteria. Turn in a printout of your JUnit tests and either a printout or a screen shot showing the results of each test.

   Instructor Solution Only

4. Delete the explict *throw* of `NullPointerException` in the `Min` program (figure 3.2). Verify that the JUnit test for a list with a single *null* element now fails.

   Solution:
   There is no "answer" for this exercise; instead, the point is to delete the following line of code:

   ```
   if (result == null) throw new NullPointerException ("Min.min");
   ```

   and then observe the `testForSoloNullElement()` *test fail.*

5. The following JUnit test method for the `sort()` method has a non-syntactic flaw. Find the flaw and describe it in terms of the RIPR model. Be as precise, specific, and concise as you can. For full credit, you must use the terminology introduced in the book.

   In the test method, `names` is an instance of an object that stores strings and has methods `add()`, `sort()`, and `getFirst()`, which do exactly what you would expect from their names. You can assume that the object `names` has been properly instantiated and the `add()` and `sort()` methods have already been tested and work correctly.

   ```
   @Test
   public void testSort()
   {
      names.add ("Laura");
   ```

```
   names.add ("Han");
   names.add ("Alex");
   names.add ("Ashley");
   names.sort();
   assertTrue ("Sort method", names.getFirst().equals ("Alex"));
}
```

Solution:
The assertion only checks a small part of the final state (the first element in the list). So if a test causes a fault to infect, and then propagate to another part of the final state, the failure will not be ***revealed****. The test oracle needs to look at the entire list.*

6. Consider the following example class. `PrimeNumbers` has three methods. The first, `computePrimes()`, takes one integer input and computes that many prime numbers. `iterator()` returns an `Iterator` that will iterate through the primes, and `toString()` returns a string representation.

```
public class PrimeNumbers implements Iterable<Integer>
{
   private List<Integer> primes = new ArrayList<Integer>();

   public void computePrimes (int n)
   {
      int count = 1; // count of primes
      int number = 2; // number tested for primeness
      boolean isPrime; // is this number a prime
      while (count <= n)
      {
         isPrime = true;
         for (int divisor = 2; divisor <= number / 2; divisor++)
         {
            if (number % divisor == 0)
            {
               isPrime = false;
               break; // for loop
            }
         }
         if (isPrime && (number % 10 != 9)) // FAULT
         {
            primes.add (number);
            count++;
         }
         number++;
      }
   }

   @Override public Iterator<Integer> iterator()
   {
      return primes.iterator();
   }

   @Override public String toString()
   {
      return primes.toString();
   }
}
```

`computePrimes()` has a fault that causes it **not** to include prime numbers whose last digit is 9 (for example, it omits 19, 29, 59, 79, 89, 109, ...). If possible, describe five tests. You

can describe the tests as sequences of calls to the above methods, or briefly describe them in words. Note that the last two tests require the test oracle to be described.

(a) A test that does not reach the fault

(b) A test that reaches the fault, but does not infect

(c) A test that infects the state, but does not propagate

(d) A test that propagates, but does not reveal

(e) A test that reveals the fault

If a test cannot be created, explain why.

Instructor Solution Only

7. Reconsider the `PrimeNumbers` class from the previous exercise. Normally, this problem is solved with the Sieve of Eratosthenes. The change in algorithm changes the consequences of the fault. Specifically, false positives are now possible in addition to false negatives. Recode the algorithm to use the Sieve approach, but leave the fault. What is the first false positive, and how many "primes" must a test case generate before encountering it? What does this exercise show about the RIPR model?

   Instructor Solution Only

8. Develop a set of data-driven JUnit tests for the `Min` program. Make your `@Parameters` method produce both *String* and *Integer* values.

   Instructor Solution Only

9. When overriding the `equals()` method, programmers are also required to override the `hashCode()` method; otherwise clients cannot store instances of these objects in common `Collection` structures such as `HashSet`. For example, the `Point` class from Chapter 1 is defective in this regard.

   (a) Demonstrate the problem with `Point` using a `HashSet`.

   Solution:

```
Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2); // note that p1.equals(p2)
Set<Point> s = new HashSet<Point>();
s.add(p1);
boolean b = s.contains(p2); // we really want b to be true!
```

While it possible that `b` *is true, it is far more likely that the two* `Point` *objects hash to different buckets, in which case* `b` *is false. Ouch!*

(b) Write down the mathematical relationship required between `equals()` and `hashCode()`.

Solution:

For a detailed explanation, see Bloch's Effective Java, Second edition, Item 9. The relationship is: if two objects are considered equal (as determined by the `equals()` method), then they must have the same hash codes (as determined by the `hashCode()` method). Note that the inverse is not true: it is perfectly fine for unequal objects to share a hash code. See Bloch Item 9 for extensive guidance on implementing good hash codes.

(c) Write a simple JUnit test to show that `Point` objects do not enjoy this property.

Solution:

```
@Test public void hashConsistentWithEquals() {
   Point p1 = new Point(1,2);
   Point p2 = new Point(1,2);
   assertTrue("Hash codes must match", p1.hashCode() == p2.hashCode());
}
```

(d) Repair the `Point` class to fix the fault.

Solution:

Following Bloch's recipe (again, Item 9):

```
@Override public int hashCode() {
   int result = 17;
   result = 31 * result + x;
   result = 31 * result + y;
   return result;
}
```

(e) Rewrite your JUnit *test* as an appropriate JUnit *theory*. Evaluate it with suitable `DataPoints`.

Solution:

```
@Theory public void hashConsistentWithEqualsTheory(Object o1, Object o2) {
   assumeTrue(o1 != null);
   assumeTrue(o2 != null);
   assumeTrue(o1.equals(o2));
   assertTrue("Hash codes must match", o1.hashCode() == o2.hashCode());
}
@DataPoints public static Object[] objects = {
   new Point(1,2), new Point(1,2), new Point(1,3), "ant", null
};
```

There are several things of note about this theory. First, to make the theory as general as possible, and hence as widely useful as possible, the test engineer should always choose parameter types that are close to the root of the type hierarchy as possible. Hence, the parameters to the theory method are of type `Object`*, and not of type* `Point`*. This is appropriate because the* `equals()` *and* `hashCode()` *methods are defined in the* `Object` *class. In other words, even though the motivation for this theory is* `Point` *objects, the result applies to all Java objects! Second, the assumptions about non-null values mean that this theory can be applied without worrying about whether the associated* `DataPoints` *happen to contain null values. Third, this theory is evaluated over the cross product of the set of five values in the* `DataPoints` *structure with itself. Of these 5*5=25 values, 5 do not pass the first precondition. Of the remaining 20, 4 do not pass the second precondition. Of the remaing 16, 6 pass the third precondition. All 6 of these satisfy the postcondition. (If you are not sure about these numbers, try it and see!)*

10. Replace each occurrence of a set with a list in the JUnit theory `removeThenAddDoesNot-ChangeSet`. Is the resulting theory valid or invalid? How many of the tests that pass the precondition also pass the postcondition? Explain.

    Solution:
    See the Java class `ListTheoryTest` *online. The resulting theory is definitely not valid because order matters in lists. Hence the JUnit theory fails.*

Chapter 4

Exercises, Chapter 4

1. Chapter 3 contained the program `Calc.java`. It is available on the program listings page on the book website.

   `Calc` currently implements one function: it adds two integers. Use test-driven design to add additional functionality to subtract two integers, multiply two integers, and divide two integers. First create a failing test for one of the new functionalities, modify the class until the test passes, then perform any refactoring needed. Repeat until all of the required functionality has been added to your new version of `Calc`, and all tests pass.

   Remember that in TDD, the tests determine the requirements. This means you must encode decisions such as whether the division method returns an integer or a floating point number in automated tests **before** modifying the software.

   Submit printouts of all tests, your final version of `Calc`, and a screen shot showing that all tests pass. Most importantly, include a narrative describing each TDD test created, the changes needed to make it pass, rnd any refactoring that was necessary.

   Solution:
   This is a "completion" exercise: the exact artifacts aren't important. What is important is going through the TDD process.

   In our experience, the most common mistake students make is to create all, or several, tests at one time. This violates the TDD process, and although it will work quite well for such a tiny example, that kind of process doesn't scale well to large programs. It is also, in our experience, a common mistake made in industry, often by companies who then say "TDD doesn't work." Another very common mistake in industry, by the way, is to skip refactoring. This quickly puts the software into mainetenance debt, again causing the managers and engineers to say "TDD doesn't work."

   An interesting variation on this exercise would be to require the Calc to be modified to include memory. That is, a value can be saved and then reused in calculations instead of simply constants. This would require lots of refactoring. Most importantly, the methods should no longer be static to avoid memory being shared amoing different uses of the class. That is, the memory variables should be individual to each object, not to the entire class.

2. Set up a contiuous integration server. Include version control for both source code and tests, and populate both with a simple example. Experiment with "breaking the build", by either introducing a fault into the source code or adding a failing test case. Restore the build.

   Solution:
   This is "completion" type exercise. There is no "right" answer; rather the point is to become familiar with at least one tool. Note: This is a great exercise for introducing students to how source code and tests are managed in practice at typical companies. At the time of this writing, Jenkins is a popular CI server, and GitHub is a popular version management system. But other tools are certainly available.

3. Most continuous integration systems offer far more than automated test execution. Extend the prior exercise so that the continuous integration server uses additional verification tools such as code coverage or a static analysis.

   Solution:
   Again, this is a "completion" type exercise, and hence has no "right" answer.

4. Find a refactoring in some large, existing system. Build tests that capture the behavior relevant to that part of the system. Refactor, and then check that the tests still pass.

   Solution:
   Another "completion" type exercise. Often, the hard part for students is picking the source code. One idea is to choose an open-source project. Another is to pick well-built and familar code such as one of the classes in the `Collection` *framework in* `java.util`*.*

5. Repair a fault in an existing system. That is, find the code that needs to change and capture the current behavior with tests. At least one of these tests must fail, thus demonstrating that you found the fault. Repair the fault and check that all of your tests now pass.

   Solution:
   Yet another "completion" type exercise. Again, the hard part for many students is choosing the source code. One trick that works well here is to intentionally place a fault in some existing code and then proceed with the exercise.

Chapter 5

Exercises, Chapter 5

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

Chapter 6

Exercises, Section 6.1

1. Return to the example at the beginning of the chapter of the two characteristics "File F sorted ascending" and "File F sorted descending." Each characteristic has two blocks. Give test case values for all four combinations of these two characteristics.

   Solution:
   We have four possibilities. Of course, many values could be used. We give simple strings using commas to indicate line returns.

   TT : cat
   TF : cat, dog
   FT : dog, cat
   FF : dog, cat, possum

2. A tester defined three characteristics based on the input parameter *car*: **Where Made**, **Energy Source**, and **Size**. The following partitionings for these characteristics have at least two mistakes. Correct them.

Where Made		
North America	Europe	Asia
Energy Source		
gas	electric	hybrid
Size		
2-door	4-door	hatch-back

   Solution:
   Where Made *is not complete. Add "other"*

   Size *overlaps, a hatch-back could be 2-door or 4-door. Either add "2-door + hatch-back," and "4-door + hatch-back," or create two new characteristics:*
   Side Doors*: 2, 4*
   Hatch-back*: yes, no*

3. 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 element in list" but do not suffer from completeness or disjointness problems.

Instructor Solution Only

4. Derive input space partitioning test inputs for the `GenericStack` class with the following method signatures:

   - `public GenericStack ();`
   - `public void push (Object X);`
   - `public Object pop ();`
   - `public boolean isEmpty ();`

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

   (a) List all of the input variables, including the state variables.

   (b) Define characteristics of the input variables. Make sure you cover all input variables.

   Instructor Solution Only

   (c) Partition the characteristics into blocks.

   Instructor Solution Only

   (d) Define values for each block.

   Instructor Solution Only

5. Consider the problem of searching for a pattern string in a subject string. One possible implementation with a specification is on the book website; `PatternIndex.java`. This particular version has an incomplete specification–and a decent interface-based input domain model singles out the problematic input! Assignment: find the the problematic input, complete the specification, and revise the implementation to match the revised specification.

   Solution:
   The problem is what to do with empty patterns - an easy case for interface-based input domain models, since the empty string is a standard special case for string types. As written, the specification doesn't address empty patterns at all – are they everywhere or nowhere?

   Note that empty subjects are a different matter - the specification has a natural interpretation for empty subjects in that patterns are never found in empty subjects. The JUnit tests in `PatternIndexTest.java` address the case of the empty subject.

   The implementation is not satisfactory as written, since the result is an exception complaining about an out-of-bounds index, yet the caller isn't supplying any indices.

One good solution is to amend the specification to explicitly reject empty patterns with the exception mechanism:

```
@throws IllegalArgumentException if pattern is empty
```

For the implementation to match, it needs an explicit check for an empty pattern, along with an explicit throws clause:

```
if (patternLen == 0) throw new IllegalArgumentException("PatternIndex.patternIndex");
```

Finally a test case should be added to `PatternIndexTest.java` *that calls* `patternIndex()` *with an empty pattern and looks for this exception.*

As a forward pointer, the `patternIndex()` *method is subject to significant scrutiny in the graph testing chapter. Yet the resulting tests don't uncover this anomaly.*

Exercises, Section 6.2

1. Write down all 64 tests to satisfy the All Combinations (ACoC) criterion for the second categorization of `triang()`'s inputs in Table 6.2. Use the values in Table 6.3.

   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. Write down all 16 tests to satisfy the Pair-Wise (PWC) criterion for the second categorization of `triang()`'s inputs in Table 6.2. Use the values in Table 6.3.

   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. Write down all 16 tests to satisfy the Multiple Base Choice Coverage (MBCC) for the second categorization of `triang()`'s inputs in Table 6.2. Use the values in Table 6.3.

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 get 2(base) + 4 + 6 + 6 = 18 tests. However, 2 of these are redundant, so the result is 16. To clarify, we list all 18 tests, generated according to the formula:*

$\{(2, 2, 2),$ $//First\ base\ test$

$(0, 2, 2), (-1, 2, 2),$ $//Vary\ first\ characteristic$
$(2, 1, 2), (2, 0, 2), (2, -1, 2),$ $//Vary\ second\ characteristic$
$(2, 2, 1), (2, 2, 0), (2, 2, -1),$ $//Vary\ third\ characteristic$

$\{(1, 2, 2),$ $//Second\ base\ test$
$(0, 2, 2), (-1, 2, 2),$ $//Vary\ first\ characteristic$
$(1, 1, 2), (1, 0, 2), (1, -1, 2),$ $//Vary\ second\ characteristic$
$(1, 2, 1), (1, 2, 0), (1, 2, -1),$ $//Vary\ third\ characteristic$
$\}$

Here are the 16 nonredundant tests:

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

Thanks to Richard Carver for correcting this solution.

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

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

   Characteristic:  Validity 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 "Validity of s1" satisfy the completeness property? If not, give a value for $s1$ that does not fit in any block.

   Instructor Solution Only

   (b) Does the partition "Validity of s1" satisfy the disjointness property? If not, give a value for $s1$ that fits in more than one block.

Instructor Solution Only

(c) Does the partition "Relation between s1 and s2" satisfy the completeness property? If not, give a pair of values for $s1$ and $s2$ that does not fit in any block.

Instructor Solution Only

(d) Does the partition "Relation between s1 and s2" satisfy the disjointness property? If not, give a pair of values for $s1$ and $s2$ 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. Use the following characteristics and blocks for the questions below.

Characteristics	Block 1	Block 2	Block 3	Block 4
Value 1	< 0	0	> 0	
Value 2	< 0	0	> 0	
Operation	$+$	$-$	$\times$	$\div$

(a) Give tests to satisfy the *Each Choice* criterion.

Solution:

Four tests are needed.

V1	*V2*	*Op*
-2	*-2*	$+$
0	*0*	$-$
2	*2*	$\times$
2	*2*	$\div$

(b) Give tests to satisfy the *Base Choice* criterion. Assume base choices are *Value 1* $=> 0$, *Value 2* $=> 0$, and *Operation* $= +$.

Solution:

Eight tests are needed.

V1	*V2*	*Op*
2	*2*	$+$
-2	*2*	$+$
0	*2*	$+$
2	*-2*	$+$
2	*0*	$+$
2	*2*	-
2	*2*	$\times$
2	*2*	$\div$

(c) How many tests are needed to satisfy the *All Combinations* criterion? (Do not list all the tests!)

Solution:

*3 * 3 * 4 = 36*

(d) Give tests to satisfy the *Pair-Wise Coverage* criterion.

Solution:

Pairs: 7 + 7 + 7 + 4 + 4 + 4 = 33

Since each test can accommodate 3 pairs, at least 11 tests are needed. The best solution involves one extra test, for a total of 12 tests:

V1	*V2*	*Op*
-2	*-2*	*+*
-2	*0*	-
-2	*2*	×
2	*-2*	÷
2	*0*	*+*
0	*2*	-
0	*-2*	×
0	*0*	÷
-2	*2*	÷
0	*0*	×
2	*-2*	-

6. Derive input space partitioning test inputs for the **BoundedQueue** class with the following signature:

   - `public BoundedQueue (int capacity); // The maximum number of elements`
   - `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) List all of the input variables, including the state variables.

   Instructor Solution Only

   (b) Define characteristics of the input variables. Make sure you cover all input variables.

   Instructor Solution Only

   (c) Partition the characteristics into blocks. Designate one block in each partition as the "Base" block.

   Instructor Solution Only

   (d) Define values for each block.

   Instructor Solution Only

   (e) Define a test set that satisfies Base Choice Coverage (BCC). Write your tests with the values from the previous step. Be sure to include the test oracles.

   Instructor Solution Only

7. Design an input domain model for the logic coverage web application on the book's website. That is, model the logic coverage web application using the input domain modeling technique.

   (a) List all of the input variables, including the state variables.

   Instructor Solution Only

   (b) Define characteristics of the input variables. Make sure you cover all input variables.
   (c) Partition the characteristics into blocks.
   (d) Designate one block in each partition as the "Base" block.

   Instructor Solution Only

   (e) Define values for each block.

   Instructor Solution Only

   (f) Define a test set that satisfies Base Choice Coverage (BCC). Write your tests with the values from the previous step. Be sure to include the test oracles.

   Instructor Solution Only

   (g) Automate your tests using the web test automation framework *HttpUnit*. Demonstrate success by submitting the HttpUnit tests and a screen dump or output showing the result of execution.
   (Note to instructors: HttpUnit is based on JUnit and is quite similar. The tests must include a URL and the framework issues the appropriate HTTP request. We usually use this question as a non-required bonus, allowing students to choose whether to learn HttpUnit on their own.)

Exercises, Section 6.4

1. The restriction on interleaving `next()` and `remove()` calls is quite complex. The JUnit tests in `IteratorTest.java` only devote one test for this situation, which may not be enough. Refine the input domain model with one or more additional characteristics to probe this behavior, and implement these tests in JUnit.

   Instructor Solution Only

2. (**Challenging!**) It is possible to modify an `ArrayList` without using the `remove()` method and yet have a subsequent call to `remove()` **fail** to throw `ConcurrentModificationException`. Develop a (failing!) JUnit test that exhibits this behavior.

   Instructor Solution Only

Chapter 7

Exercises, Section 7.1

1. Give the sets N, N_0, N_f, and E for the graph in Figure 7.2.

   Solution:
   $N = \{1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10\}$
   $N_0 = \{1,\ 2,\ 3\}$
   $N_f = \{8,\ 9,\ 10\}$
   $E = \{(1,\ 4), (1,\ 5), (2,\ 5), (3,\ 6), (3,\ 7), (4,\ 8), (5,\ 8), (5,\ 9),$
   $(6,\ 2), (6,\ 10), (7,\ 10), (9,\ 6), \}$

   Thanks to Aya Salah, Steven Dastvan, and Rich Dillon for correcting this solution.

2. Give a path that is not a test path in Figure 7.2.

   Solution:
   Obviously, there are many possibilities. $[5,\ 9,\ 5,\ 10]$ *is not a test path because it does not start in an initial node.* $[2,\ 5,\ 9,\ 5]$ *is not a test path because it does not end in a final node.*

3. List all test paths in 7.2.

   Solution:
   There are an unbounded number of these. In particular, any path that visits the loop $[2,\ 5,\ 9,\ 6]$ *can be extended indefinitely. We list some of the shorter test paths below.* $[1,\ 4,\ 8]$, $[1,\ 5,\ 8]$, $[1,\ 5,\ 9]$, $[1,\ 5,\ 9,\ 6,\ 2, 5, 9]$, $[1,\ 5,\ 9,\ 6,\ 10]$, $[2,\ 5,\ 9]$, $[2,\ 5,\ 9,\ 6,\ 2, 5, 9]$, $[2,\ 5,\ 9,\ 6,\ 10]$, $[3,\ 6,\ 10]$, $[3,\ 6, 2,\ 5,\ 9]$, $[3,\ 6, 2,\ 5,\ 9,\ 6,\ 2, 5, 9]$, $[3,\ 7,\ 10]$.

4. In Figure 7.5, find test case inputs such that the corresponding test path visits edge $(2,\ 4)$.

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

Exercises, Section 7.2.2

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) If possible, list test paths that achieve Node Coverage, but not Edge Coverage. If not possible, explain why not.

   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) If possible, list test paths that achieve Edge Coverage, but not Edge Pair Coverage. If not possible, explain why not.

   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:
   - $p_1 = [1,\ 2,\ 4,\ 5,\ 6,\ 1,\ 7]$

- $p_2 = [1,\ 2,\ 3,\ 2,\ 4,\ 6,\ 1,\ 7]$
- $p_3 = [1,\ 2,\ 3,\ 2,\ 4,\ 5,\ 6,\ 1,\ 7]$

(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, state 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, write down 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 from the given set that achieve Node Coverage but not Edge Coverage on the graph.

Instructor Solution Only

(g) List test paths from the given set 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 7.2.

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

Solution:
$TR_{NC} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
$TR_{EC} = \{(1,4),\ (1,5),\ (2,5),\ (3,6),\ (3,7),\ (4,8),\ (5,8),\ (5,9),\ (6,2),\ (6,10),\ (7,10),\ (9,6)\ \}$
$TR_{PPC} = \{\ [1,4,8],\ [1,5,8],\ [1,5,9,6,2],\ [1,5,9,6,10],\ [2,5,9,6,2],\ [2,5,9,6,10],\ [3,6,2,5,8],\ [3,6,2,5,9],\ [3,6,10],\ [3,7,10],\ [5,9,6,2,5],\ [6,2,5,9,6],\ [9,6,2,5,8],\ [9,6,2,5,9]\ \}$

(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:

- $N = \{1,\ 2,\ 3\}$

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

Also consider the following (candidate) paths:

- $p_1 = [1,\ 2,\ 3,\ 1]$
- $p_2 = [1,\ 3,\ 1,\ 2,\ 3]$
- $p_3 = [1,\ 2,\ 3,\ 1,\ 2,\ 1,\ 3]$
- $p_4 = [2,\ 3,\ 1,\ 3]$
- $p_5 = [1,\ 2,\ 3,\ 2,\ 3]$

(a) Which of the listed paths are test paths? For any path that is not a test path, explain why not.

Solution:

Answer: p_2 and p_3 are test paths. p_1 does not terminate at a final node. p_4 does not start at an initial node. p_5 includes an edge that does not exist in the graph $(3,2)$.

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

Solution:

Answer: The edge pairs are:

$\{\ [1,2,1],\ [1,2,3],\ [1,3,1],\ [2,1,2],\ [2,1,3],\ [2,3,1],\ [3,1,2],\ [3,1,3]\ \}$

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

Solution:

Answer: No. Neither p_2 nor p_3 tours either of the following edge-pairs:

$\{\ [2,1,2],\ [3,1,3]\ \}$

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

(d) Consider the prime path [3, 1, 3] and path p_3. Does p_3 tour the prime path directly? With a sidetrip?

Solution:

Answer: p_3 does not directly tour the prime path. However, p_3 does tour the prime path with the sidetrip $[1,2,1]$.

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 7.2.3

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 some test paths. Answer the following questions about each graph.

Graph I.
$N = \{1, 2, 3, 4, 5, 6, 7, 8\}$
$N_0 = \{1\}$
$N_f = \{8\}$
$E = \{(1,2), (2,3), (2,8), (3,4), (3,5), (4,3), (5,6), (5,7), (6,7), (7,2)\}$
$def(1) = def(4) = use(6) = use(8) = \{x\}$
Test Paths:
$t1 = [1, 2, 8]$
$t2 = [1, 2, 3, 5, 7, 2, 8]$
$t3 = [1, 2, 3, 5, 6, 7, 2, 8]$
$t4 = [1, 2, 3, 4, 3, 5, 7, 2, 8]$
$t5 = [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 2, 8]$
$t6 = [1, 2, 3, 4, 3, 5, 7, 2, 3, 5, 6, 7, 2, 8]$

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:
$t1 = [1, 2, 6]$
$t2 = [1, 2, 3, 4, 5, 2, 3, 5, 2, 6]$
$t3 = [1, 2, 3, 5, 2, 3, 4, 5, 2, 6]$
$t4 = [1, 2, 3, 5, 2, 6]$

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:
$t_1 = [1, 2, 3, 5, 2, 6]$
$t_2 = [1, 2, 3, 4, 5, 2, 6]$

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:
$t1 = [1, 2, 6]$
$t2 = [1, 2, 3, 4, 5, 2, 3, 5, 2, 6]$
$t3 = [1, 2, 3, 5, 2, 3, 4, 5, 2, 6]$

   (a) Draw the graph.
   (b) 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).
   (c) Determine which du-paths each test path tours. Write them in a table with test paths in the first column and the du-paths they cover in the second column. For this part of the exercise, you should consider both direct touring and sidetrips.
   (d) List a minimal test set that satisfies *all defs* coverage with respect to x. (Direct tours only.) Use the given test paths.
   (e) List a minimal test set that satisfies *all uses* coverage with respect to x. (Direct tours only.) Use the given test paths.
   (f) 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	*[1, 2, 8]*
ii	*[1, 2, 3, 5, 6]*
iii	*[4, 3, 5, 6]*
iv	*[4, 3, 5, 7, 2, 8]*
v	*[4, 3, 5, 6, 7, 2, 8]*

(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.*

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

(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, Ignacio Martín, Stephanie Blake, and Rob Jones for correcting various parts of 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.*

	direct	*w/ sidetrip*
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) directly. t_1 does tour du-path (iii) with a sidetrip. But t_2 does not tour du-path (iii) with a sidetrip; the problem is the def of x in node 4.

(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 7.3

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) List 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 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 7.25. Instrument the code to produce the execution paths in the text for this example. That is, on a given test execution, your instrumentated program should compute and print the corresponding test path. Run the instrumented program on the test cases listed at the end of Section 7.3.

Solution:
Access PatternIndexInstrument.java on the book website.

4. Consider the pattern matching example in Figure 7.25. In particular, consider the final table of tests in Section 7.3. 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:

Test Number	*Test*
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 7.25. Instrument the code 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:

Access PatternIndexInstrument.java on the book website.

t_1: `java PatternIndexInstrument "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 PatternIndexInstrument "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 PatternIndexInstrument 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 7.3. To reduce the scope of this exercise, consider only the following du-paths: *du (10, iSub)*, *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 7.5. That is, we ignore the du-paths that are prefixes of other du-paths; see Table 7.3 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,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]	-!	-	-

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

Solution:
Since the value if `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 7.3 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 7.5, 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* $\{t_1,\ t_3\ (ab, a),\ (ab, ac)\}$*.*

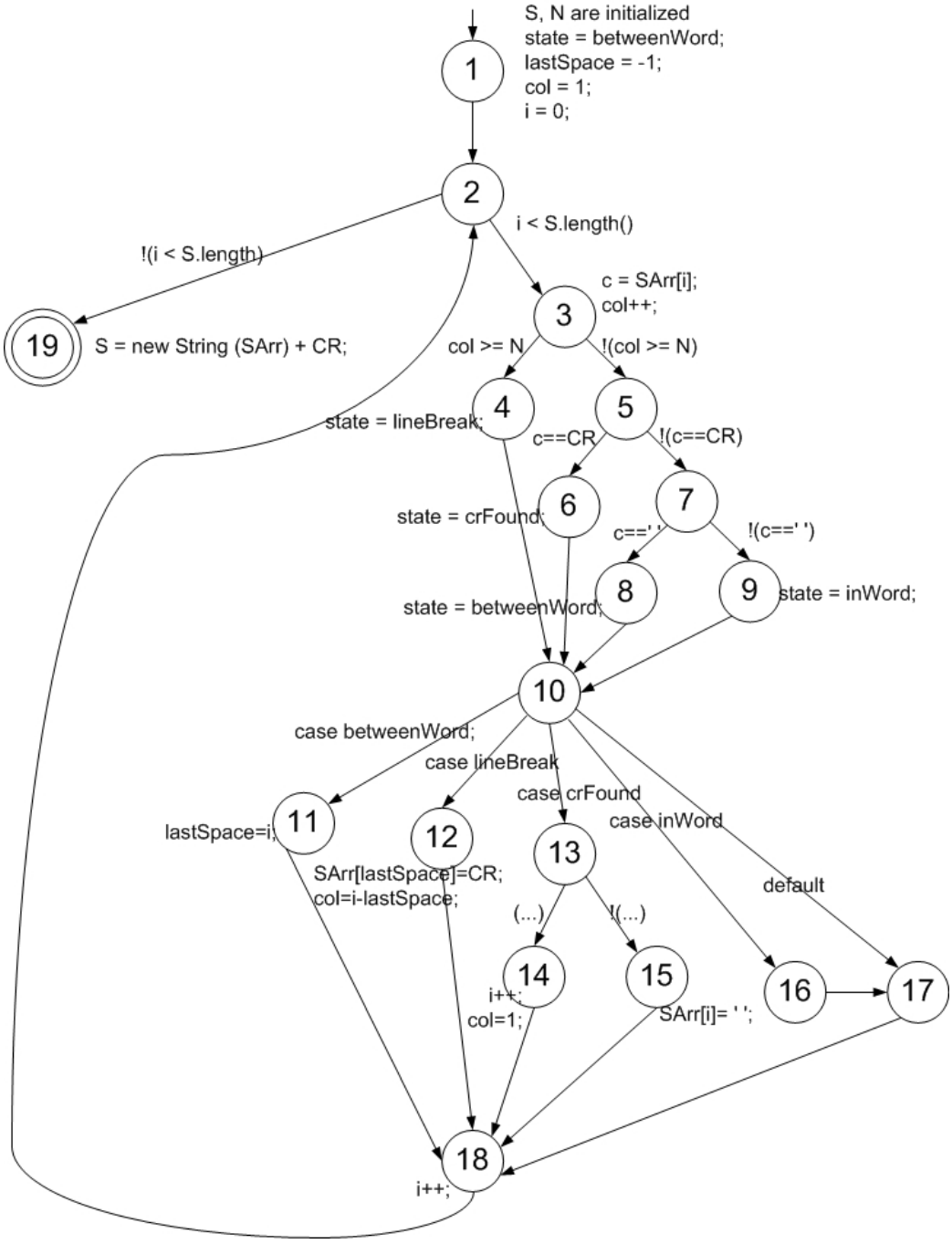
(g) Is there any difference between All-Uses Coverage and all-DU-Paths Coverage with respect to the variable *isPat* in the *pat()* method?

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 method `fmtRewrap()` for questions a–e below. A compilable version is available on the book website in the file `FmtRewrap.java`. A line-numbered version suitable for this exercise is available on the book website in the file `FmtRewrap.num`.

(a) Draw the control flow graph for the `fmtRewrap()` 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 `fmtRewrap()`, 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) List the test requirements for Node Coverage, Edge Coverage, and Prime Path Coverage.

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:

This is certainly possible. The key is noticing that it is possible to visit nodes 10, 16, and 17 without traversing edge (10,17). Test paths are available online. See the graph tool at `http://www.cs.gmu.edu/~offutt/softwaretest/`.

Thanks to Martin Gebert for correcting this solution.

(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 method `printPrimes()` for questions a–f below. A compilable version is available on the book website in the file `PrintPrimes.java`. A line-numbered version suitable for this exercise is available on the book website in the file `PrintPrimes.num`.

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

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

8. Consider the `equals()` method from the *java.util.AbstractList<E>* class:

```
public boolean equals (Object o)
{
  if (o == this) // A
     return true;
  if (!(o instanceof List)) // B
     return false;

  ListIterator<E> e1 = listIterator();
  ListIterator<?> e2 = ((List) o).listIterator();
  while (e1.hasNext() && e2.hasNext()) // C
  {
     E o1 = e1.next();
     Object o2 = e2.next();
     if (!(o1 == null ? o2 == null : o1.equals (o2))) // D
        return false;
  }
  return !(e1.hasNext() || e2.hasNext()); // E
}
```

(a) Draw a control flow graph for this method. Several possible values can be used for the node numbers in the graph. Choose something reasonable.

Instructor Solution Only

(b) Label edges and nodes in the graph with the corresponding code fragments. You may abbreviate predicates as follows when labeling your graph:

A: `o == this`
B: `!(o instanceof List)`
C: `e1.hasNext() && e2.hasNext()`
C: `e1.hasNext() && e2.hasNext()`
D: `!(o1 == null ? o2 == null : o1.equals(o2))`
E: `!(e1.hasNext() || e2.hasNext())`

Instructor Solution Only

(c) Node coverage requires (at least) four tests on this graph. Explain why.

Instructor Solution Only

(d) Provide four tests (as calls to `equals()`) that satisfy node coverage on this graph. Make your tests short. You need to include output assertions. Assume that each test is independent and starts with the following state:

List<String> list1 = new ArrayList<String>();
List<String> list2 = new ArrayList<String>();
Use the constants null, “ant”, “bat”, etc. as needed.

Instructor Solution Only
Instructor Solution Only
Instructor Solution Only
Instructor Solution Only
Instructor Solution Only

Exercises, Section 7.4

1. Use the class `Watch` in Figures 7.38 and 7.39 in Section 7.5 to answer questions a–d below.

   (a) Draw a control flow graph for `Watch`.

   Solution:
   The point of this question is getting students to think about how to handle the call sites. Figure 7.40, which gives control flow graphs for methods `doTransition()` and `changeTime()`, 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  public class Watch
 2  {
 3     // Constant values for the button (inputs)
 4     private static final int NEXT = 0;
 5     private static final int UP   = 1;
 6     private static final int DOWN = 2;
 7
 8     // Constant values for the state
 9     private static final int TIME      = 5;
10     private static final int STOPWATCH = 6;
11     private static final int ALARM     = 7;
12
13     // Primary state variable
14     private int mode = TIME;
15
16     // Three separate times, one for each state
17     private Time watch, stopwatch, alarm;
18
19     // Inner class keeps track of hours and minutes
20     public class Time
21     {
22        private int hour   = 0;
23        private int minute = 0;
24
25        // Increases or decreases the time.
26        // Rolls around when necessary.
27        public void changeTime (int button)
28        {
29           if (button == UP)
30           {
31              minute += 1;
32              if (minute >= 60)
33              {
34                 minute = 0;
35                 hour += 1;
36                 if (hour >= 12)
37                    hour = 0;
38              }
39           }
40           else if (button == DOWN)
41           {
42              minute -= 1;
43              if (minute < 0)
44              {
45                 minute = 59;
46                 hour -= 1;
47                 if (hour <= 0)
48                    hour = 12;
49              }
50           }
51        }  // end changeTime()
52
53        public String toString ()
54        {
55           return (hour + ":" + minute);
56        }  // end toString()
57     }  // end class Time
58
```

```
 59
 60     public Watch () // Constructor
 61     {
 62        watch = new Time();
 63        stopwatch = new Time();
 64        alarm = new Time();
 65     }  // end Watch constructor
 66
 67     public String toString ()  // Converts values
 68     {
 69        return ("watch is: " + watch + "\n"
 70                + "stopwatch is: " + stopwatch + "\n"
 71                + "alarm is: " + alarm);
 72     }  // end toString()
 73
 74     public void doTransition (int button) // Handles inputs
 75     {
 76        switch (mode)
 77        {
 78           case TIME:
 79              if (button == NEXT)
 80                 mode = STOPWATCH;
 81              else
 82                 watch.changeTime (button);
 83              break;
 84           case STOPWATCH:
 85              if (button == NEXT)
 86                 mode = ALARM;
 87              else
 88                 stopwatch.changeTime (button);
 89              break;
 90           case ALARM:
 91              if (button == NEXT)
 92                 mode = TIME;
 93              else
 94                 alarm.changeTime (button);
 95              break;
 96           default:
 97              break;
 98        }
 99     }  // end doTransition()
100  }  // end Watch
```

Solution:

The call sites are:

i. *Line 62, Watch::constructor() → Time:constructor()*
ii. *Line 63, Watch::constructor() → Time:constructor()*
iii. *Line 64, Watch::constructor() → Time:constructor()*
iv. *Line 69, Watch::toString() → Time:toString()*
v. *Line 70, Watch::toString() → Time:toString()*
vi. *Line 70, Watch::toString() → Time:toString()*
vii. *Line 82, Watch::doTransition() → Time:changeTime()*
viii. *Line 88, Watch::doTransition() → Time:changeTime()*
ix. *Line 94, Watch::doTransition() → Time:changeTime()*

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

Solution:

i. *Call sites i, ii, and iii are implicit within the constructor for* `Watch`*. Since Time does not have a constructor defined, the default constructor is called (by Java's rules, that means the default values are assigned to the instance variales, that is, hour and minute both get the value 0).*

ii. *Call sites iv, v, and vi are from* `Watch`*'s toString() method to Time's toString method. The Time objects are passed in, and a string is returned. Time's toString() method does not refer to the object explicitly, so we use "instance" for the variable name. The six du-pairs are:*

   A. *(Watch::toString(), watch, 69) → (Time::toString(), instance, 55)*
   B. *(Watch::toString(), stopwatch, 70) → (Time::toString(), instance, 55)*
   C. *(Watch::toString(), alarm, 71) → (Time::toString(), instance, 55)*
   D. *(Time::toString(), String, 55) → (Watch::toString(), String, 69)*
   E. *(Time::toString(), String, 55) → (Watch::toString(), String, 70)*
   F. *(Time::toString(), String, 55) → (Watch::toString(), String, 71)*

iii. *Call sites vii, viii, and ix are from Watch's doTransition() to Time's changeTime(). The parameter button is implicitly defined at line 74 (entrance to the method), and the Time objects are defined within changeTime(). The first three coupling du-pairs have the same line numbers for the last-defs and first-uses, so are annoted with the call site number. Because of the complex logic in changeTime(), it contains six last-defs of its instance variable (assignments to minute and hour), resulting in 18 coupling du-pairs for values returned to doTransition().*

   A. *Call site vii: (Watch::doTransition(), button, 74) → (Time::changeTime(), 29)*
   B. *Call site viii: (Watch::doTransition(), button, 74) → (Time::changeTime(), 29)*
   C. *Call site vix: (Watch::doTransition(), button, 74) → (Time::changeTime(), 29)*
   D. *(Time::changeTime(), instance, 31) → (Watch::doTransition(), watch, 82)*
   E. *(Time::changeTime(), instance, 35) → (Watch::doTransition(), watch, 82)*
   F. *(Time::changeTime(), instance, 37) → (Watch::doTransition(), watch, 82)*
   G. *(Time::changeTime(), instance, 42) → (Watch::doTransition(), watch, 82)*
   H. *(Time::changeTime(), instance, 46) → (Watch::doTransition(), watch, 82)*
   I. *(Time::changeTime(), instance, 48) → (Watch::doTransition(), watch, 82)*
   J. *(Time::changeTime(), instance, 31) → (Watch::doTransition(), stopwatch, 88)*
   K. *(Time::changeTime(), instance, 35) → (Watch::doTransition(), stopwatch, 88)*
   L. *(Time::changeTime(), instance, 37) → (Watch::doTransition(), stopwatch, 88)*
   M. *(Time::changeTime(), instance, 42) → (Watch::doTransition(), stopwatch, 88)*
   N. *(Time::changeTime(), instance, 46) → (Watch::doTransition(), stopwatch, 88)*
   O. *(Time::changeTime(), instance, 48) → (Watch::doTransition(), stopwatch, 88)*
   P. *(Time::changeTime(), instance, 31) → (Watch::doTransition(), alarm, 94)*
   Q. *(Time::changeTime(), instance, 35) → (Watch::doTransition(), alarm, 94)*

R. *(Time::changeTime(), instance, 37) → (Watch::doTransition(), alarm, 94)*
S. *(Time::changeTime(), instance, 42) → (Watch::doTransition(), alarm, 94)*
T. *(Time::changeTime(), instance, 46) → (Watch::doTransition(), alarm, 94)*
U. *(Time::changeTime(), instance, 48) → (Watch::doTransition(), alarm, 94)*

(d) Create test data to satisfy *All-Coupling-Use Coverage* for `Watch`.

Solution:

*This example demonstrates the power of All-Coupline-Use, because it requires some very long tests. A common concern about test criteria and automated test data generation is that they might not require long tests. To tour the coupling du-pairs on the "rollover" assignments of minute at lines 35 and 46, we must increment the watch 60 times. To tour the coupling du-pairs on the "rollover" assignments of hour at lines 37 and 48, we must increment the watch 60*24 = 1440 times! The other coupling du-pairs are trivial, so we only give tests for the 18 coupling du-pairs from changeTime() to doTransition().*

i. *t1: mode = TIME, button = UP*
ii. *t2: mode = TIME, button = UP, UP, ..., UP 60 times (we abbreviate this as* UP^{60}*)*
iii. *t3: mode = TIME, button =* UP^{1440}
iv. *t4: mode = TIME, button = DOWN*
v. *t5: mode = TIME, button =* $DOWN^{60}$
vi. *t6: mode = TIME, button =* $DOWN^{1440}$
vii. *t7: mode = STOPWATCH, button = UP*
viii. *t8: mode = STOPWATCH, button =* UP^{60}
ix. *t9: mode = STOPWATCH, button =* UP^{1440}
x. *t10: mode = STOPWATCH, button = DOWN*
xi. *t11: mode = STOPWATCH, button =* $DOWN^{60}$
xii. *t12: mode = STOPWATCH, button =* $DOWN^{1440}$
xiii. *t13: mode = ALARM, button = UP*
xiv. *t14: mode = ALARM, button =* UP^{60}
xv. *t15: mode = ALARM, button =* UP^{1440}
xvi. *t16: mode = ALARM, button = DOWN*
xvii. *t17: mode = ALARM, button =* $DOWN^{60}$
xviii. *t18: mode = ALARM, button =* $DOWN^{1440}$

2. Use the class `Stutter` for questions a–d below. A compilable version is available on the book website in the file `Stutter.java`. A line-numbered version suitable for this exercise is available on the book website in the file `Stutter.num`.

(a) Draw control flow graphs for the methods in `Stutter`.

Instructor Solution Only

(b) List all the call sites.

Instructor Solution Only

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

Instructor Solution Only

(d) Create test data to satisfy *All-Coupling Use Coverage* for `Stutter`.

Instructor Solution Only

3. Use the following program fragment for questions a–e below.

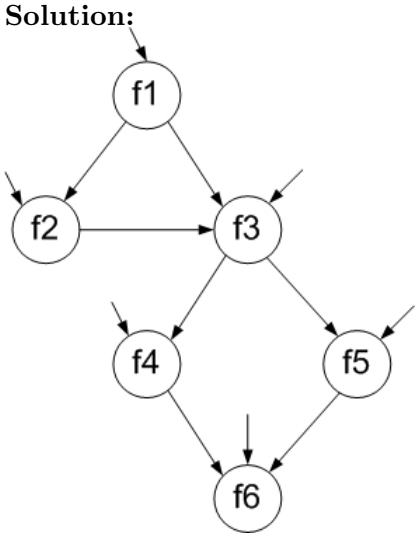
```
public static void f1 (int x, int y)
{
   if (x < y) { f2 (y); } else { f3 (y); };
}
public static void f2 (int a)
{
   if (a % 2 == 0) { f3 (2*a); };
}
public static void f3 (int b)
{
   if (b > 0) { f4(); } else { f5(); };
}
public static void f4() {... f6()....}
public static void f5() {... f6()....}
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. Which prime path is not covered by any of the tests 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.

4. Use the following methods `trash()` and `takeOut()` to answer questions a–c.

```
 1 public void trash (int x)
 2 {
 3    int m, n;
 4
 5    m = 0;
 6    if (x > 0)
 7       m = 4;
 8    if (x > 5)
 9       n = 3*m;
10    else
11       n = 4*m;
12    int o = takeOut (m, n);
13    System.out.println ("o is: " + o);
14 }
```

```
15 public int takeOut (int a, int b)
16 {
17    int d, e;
18
19    d = 42*a;
20    if (a > 0)
21       e = 2*b+d;
22    else
23       e = b+d;
24    return (e);
25 }
```

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

Instructor Solution Only

(b) Give all pairs of *last-def*s and *first-use*s.

Instructor Solution Only

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

Instructor Solution Only

Exercises, Section 7.5

1. Use the class `BoundedQueue2` for questions a–f below. A compilable version is available on the book website in the file `BoundedQueue2.java`. The queue is managed in the usual circular fashion.

   Suppose we build a FSM where states are defined by the representation variables of **BoundedQueue2**. 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*. What are they?

   Instructor Solution Only

   (b) How many states are there?

   Instructor Solution Only

   (c) How many of these states are reachable?

   Instructor Solution Only

(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 FSM that models 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 7.6

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

Chapter 8

Exercises, Section 8.1

1. List all the clauses for the predicate below:
   $((f <= g) \wedge (X > 0)) \vee (M \wedge (e < d + c))$

   Solution:
   There are four: $f <= g$, $X > 0$, M, *and* $e < d + c$.

2. List all the clauses for the predicate below:
   $(G \vee ((m > a) \vee (s <= o + n)) \wedge U)$

   Instructor Solution Only

3. 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.

4. Use predicates (i) through (x) to answer the following questions.

   i. $p = a \wedge (\neg b \vee c)$
   ii. $p = a \vee (b \wedge c)$
   iii. $p = a \wedge b$
   iv. $p = a \rightarrow (b \rightarrow c)$
   v. $p = a \oplus b$
   vi. $p = a \leftrightarrow (b \wedge c)$
   vii. $p = (a \vee b) \wedge (c \vee d)$
   viii. $p = (\neg a \wedge \neg b) \vee (a \wedge \neg c) \vee (\neg a \wedge c)$
   ix. $p = a \vee b \vee (c \wedge d)$
   x. $p = (a \wedge b) \vee (b \wedge c) \vee (a \wedge c)$

   (a) List the clauses that go with predicate p.

   (b) Compute (and simplify) the conditions under which each clause determines predicate p.

   (c) Write the complete truth table for the predicate. Label your rows starting from 1. Use the format in the example underneath the definition of Combinatorial Coverage in Section 8.1.1. 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 value of the predicate itself.

(d) List **all** pairs of rows from your table that satisfy General Active Clause Coverage (GACC) with respect to each clause.

(e) List **all** pairs of rows from your table that satisfy Correlated Active Clause Coverage (CACC) with respect to each clause.

(f) List **all** pairs of rows from your table that satisfy Restricted Active Clause Coverage (RACC) with respect to each clause.

(g) List **all** 4-tuples of rows from your table that satisfy General Inactive Clause Coverage (GICC) with respect to each clause. List any infeasible GICC test requirements.

(h) List **all** 4-tuples of rows from your table that satisfy Restricted Inactive Clause Coverage (RICC) with respect to each clause. List any infeasible RICC test requirements.

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

Instructor Solution Only

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

(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 (iii), $p = a \wedge b$

(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 (iv), $p = a \rightarrow (b \rightarrow c)$

(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* $p = T$.

Solution for (v), $p = a \oplus b$

Instructor Solution Only

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

(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 (vii), $p = (a \vee b) \wedge (c \vee d)$

(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 (viii), $p = (\neg a \wedge \neg b) \vee (a \wedge \neg c) \vee (\neg a \wedge c)$

(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* $p = T$.

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

Instructor Solution Only

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

Instructor Solution Only

5. Show that GACC does **not** subsume PC when the exclusive *or* operator is used. Assume $p = a \oplus b$.

   Instructor Solution Only

6. In Section 8.1.6, we introduced the example $p = (a \vee b) \wedge c$, and provided expanded versions of the clauses using program variables. We then gave specific values to satisfy PC. We also gave truth values to satisfy CC. Find values for the program variables given to satisfy CC; that is, refine the abstract tests into concrete test values.

   Instructor Solution Only

7. 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.

8. (**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 8.2

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

   i. $f = ab\bar{c} + \bar{a}b\bar{c}$
   ii. $f = \bar{a}\bar{b}\bar{c}\bar{d} + abcd$
   iii. $f = ab + a\bar{b}c + \bar{a}\bar{b}c$
   iv. $f = \bar{a}\bar{c}\bar{d} + \bar{c}d + bcd$

   (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 Multiple Unique True Points (MUTP) for f.
   (e) Give a test set that satisfies Corresponding Unique True Point and Near False Point Pair Coverage (CUTPNFP) for f.
   (f) Give a test set that satisfies Multiple Near False Points (MNFP) for f.
   (g) Give a test set that is guaranteed to detect all faults in figure 8.2.

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

(a) Karnaugh map for f*:*

		$a,\ b$			
		00	*01*	*11*	*10*
c	*0*		*1*	*1*	
	1				

Karnaugh map for $\bar{f}$*:*

		$a,\ b$			
		00	*01*	*11*	*10*
c	*0*	*1*			*1*
	1	*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 MUTP, see the online tool.

(e) For CUTPNFP, see the online tool.

(f) For MNFP, see the online tool.

(g) Any version of MUMCUT works; see the online tool.

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

(a) Karnaugh map for f*:*

cd \ $a,\ b$	*00*	*01*	*11*	*10*
00	*1*			
01				
11			*1*	
10				

Karnaugh map for $\bar{f}$*:*

cd \ $a,\ b$	*00*	*01*	*11*	*10*
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* $\{FFFF, TTTT, TFTF, FTFT\}$ *satisfies IC. Note that the third and fourth tests, which are not unique true points, each satisfy two implicants.*
(d) For MUTP, see the online tool.
(e) For CUTPNFP, see the online tool.
(f) For MNFP, see the online tool.
(g) Any version of MUMCUT works; see the online tool.

Instructor Solution Only

Instructor Solution Only

2 Use the following predicates to answer questions (a) through (f).

- $W = (b \wedge \neg c \wedge \neg d)$
- $X = (b \wedge d) \vee (\neg b \neg d)$
- $Y = (a \wedge b)$
- $Z = (\neg b \wedge d)$

(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) Find the minimal DNF expression that describes all cells that have more than one definition.

Instructor Solution Only

(c) Find the minimal DNF expression that describes all cells that have no definitions.

Instructor Solution Only

(d) Find the minimal DNF expression that describes $X \vee Z$.

Instructor Solution Only

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

Instructor Solution Only

3 Consider "stuck-at" faults, where a literal is replaced by the constant *true* or the constant $false$. These faults do not appear in the fault list given in table 8.1 or the corresponding fault detection relationships given in figure 8.2.

(a) Which fault dominates the stuck-at fault for the constant *true*? That is, find the fault in figure 8.2 such that if a test set is guaranteed to detect every occurrence of that fault, then the test set also detects all stuck-at *true* faults. Explain your answer.

Solution:
Detecting all LOF (Literal Omission) faults guarantees detecting all stuck-at true faults. The reason is that omitting a literal is logically the same as replacing a literal by the constant true.

(b) Which fault dominates the stuck-at fault for the constant $false$? That is, find the fault in figure 8.2 such that if a test set is guaranteed to detect every occurrence of that fault, then the test set also detects all stuck-at $false$ faults. Explain your answer.

Instructor Solution Only

Exercises, Section 8.3

1. Complete and run the tests to satisfy PC for the `Thermostat` class.

   Instructor Solution Only

2. Complete and run the tests to satisfy CC for the `Thermostat` class.

   Instructor Solution Only

3. Complete and run the tests to satisfy CACC for the `Thermostat` class.

   Instructor Solution Only

4. For the `Thermostat` class, check the computations for how to make each major clause determine the value of the predicate by using the online tool, then the tabular method.

   Instructor Solution Only

5. 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");
  }
}
```

   (a) 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

   (b) 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

   (c) 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
```

6. 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";
}
```

   (a) List test inputs for `twoPred()` that achieve Restricted Active Clause Coverage (RACC). **Instructor Solution Only**

   (b) List test inputs for `twoPred()` that achieve Restricted Inactive Clause Coverage (RICC). **Instructor Solution Only**

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

```
fragment P:
   if (A || B || C)
   {
      m();
   }
   return;
```

```
fragment Q:
   if (A)
   {
      m();
      return;
   }
   if (B)
   {
      m();
      return;
   }
   if (C)
   {
      m();
   }
```

(a) 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): $T_{GACC} = \{(T,F,F),(F,T,F),(F,F,T),(F,F,F)\}$

(b) Does the GACC test set for fragment `P` satisfy Edge Coverage on fragment `Q`?

Solution:
Yes.

(c) 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:
$T_{EDGE:1} = \{(T,T,T),(F,T,T),(F,F,T),(F,F,F)\}$
$T_{EDGE:2} = \{(T,T,F),(F,T,T),(F,F,T),(F,F,F)\}$
$T_{EDGE:3} = \{(T,F,T),(F,T,T),(F,F,T),(F,F,F)\}$
$T_{EDGE:4} = \{(T,F,F),(F,T,T),(F,F,T),(F,F,F)\}$ *(1 in the GACC test)*
$T_{EDGE:5} = \{(T,T,T),(F,T,F),(F,F,T),(F,F,F)\}$ *(1 in the GACC test)*
$T_{EDGE:6} = \{(T,T,F),(F,T,F),(F,F,T),(F,F,F)\}$ *(1 in the GACC test)*
$T_{EDGE:7} = \{(T,F,T),(F,T,F),(F,F,T),(F,F,F)\}$ *(1 in the GACC test)*
$T_{EDGE:8} = \{(T,F,F),(F,T,F),(F,F,T),(F,F,F)\}$ *(2 in the GACC tests)*

8. For the `Pattern` program in Chapter 7, complete the test sets for the following coverage criteria by filling in the "don't care" values. Make sure to ensure reachability. Then derive the expected output. Download the program, compile it, and run it with your resulting test cases to verify correct outputs.

   (a) Predicate Coverage (PC)
   (b) Clause Coverage (CC)
   (c) Combinatorial Coverage (CoC)
   (d) Correlated Active Clause Coverage (CACC)

Instructor Solution Only

9. For the `Quadratic` program in Chapter 6, 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.

(a) Predicate Coverage (PC)
(b) Clause Coverage (CC)
(c) Combinatorial Coverage (CoC)
(d) Correlated Active Clause Coverage (CACC)

Instructor Solution Only

10. The program `TriTyp` is an old and well used example from the unit testing research literature. `Trityp` is used as a teaching tool for the same reasons it has staying power in the literature: the problem is familiar; the control structure is interesting enough to illustrate most issues; and it does not use language features that make this analysis really hard, such as loops and indirect references. This version of `Trityp` is more complicated than some, but that helps illustrate the concepts. `Trityp` is a simple triangle classification program. Line numbers were added to allow us to refer to specific decision statements in the the answers.

    Use `TriTyp`, a numbered version of which is available on the book web site, to answer the questions below. Only the `triang()` method is considered.

    (a) List all predicates in the `triang()` method. Index them by the line numbers in the program listing.

    Instructor Solution Only

    (b) Compute reachability for each of `triang()`'s predicates. You may abbreviate the input variables as `S1`, `S2`, and `S3`.

    Instructor Solution Only

    (c) Many of the reachability predicates contain an internal variable (`triOut`). Resolve the internal variable in terms of input variables. That is, determine what values the input variables need to have to give `triOut` each possible value.

    Instructor Solution Only

    (d) Rewrite the reachability predicates by solving for `triOut`. That is, the reachability predicates should be completely in terms of the input variables.

    Instructor Solution Only

    (e) Find values for each predicate to satisfy predicate coverage (PC).

    Instructor Solution Only

    (f) Find values for each predicate to satisfy clause coverage (CC).

    Instructor Solution Only

    (g) Find values for each predicate to satisfy correlated active clause coverage (CACC).

    Instructor Solution Only

11. (**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.

    (a) Predicate Coverage (PC)
    (b) Clause Coverage (CC)
    (c) Combinatorial Coverage (CoC)

(d) Correlated Active Clause Coverage (CACC)

Instructor Solution Only

12. Consider the `GoodFastCheap` class, available on the book web site. This class implements the old engineering joke: "Good, Fast, Cheap: Pick any two!"

    (a) Develop tests that achieve RACC for the predicate in the `isSatisfactory()` method. Implement these tests in JUnit.

    Solution:
    Access GoodFastCheapRACC.java on the book website

    (b) Suppose we refactor the `isSatisfactory()` method as shown below:

```
public boolean isSatisfactory()
{
   if (good && fast) return true;
   if (good && cheap) return true;
   if (fast && cheap) return true;

   return false;
}
```

    The RACC tests from the original method do not satisfy RACC on the refactored method. List what is missing, and add the missing tests to the JUnit from the prior exercise.

    Instructor Solution Only

    (c) Develop tests that achieve MUMCUT for the predicate in the `isSatisfactory()` method of the `GoodFastCheap` class. Implement these tests in JUnit.

    Solution:
    The online tool shows that tests 2, 3, 4, 5, 6, and 7 are needed for this predicate. Access GoodFastCheapMUMCUT.java on the book website

Exercises, Section 8.4

1. 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`.

   (a) Formalize the precondition.
   Instructor Solution Only

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

   (c) Develop and run CACC tests on the implementation.
   Instructor Solution Only

Exercises, Section 8.5

1. For the `Memory Seat` finite state machine, complete the test sets for the predicate coverage criterion (PC) by satisfying the predicates, ensuring reachability, and computing the expected output.

   Solution:
   The Memory Seat FSM has five states and 24 transitions. For each transition, we give the transition, the predicate, the prefix values, and the truth assignments needed to satisfy the criterion.

   (a) **Transition**: $1 \longrightarrow 2$
   Predicate: $Button2 \wedge (Gear = Park \vee ignition = off)$
   Prefix: *values:* $Gear = Park$, $Button1$

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	$Button2 \wedge Gear = Park$	T	*2*
	$Button1 \wedge Gear = Park$	F	*1*

   (b) **Transition**: $1 \longrightarrow 3$
   Predicate: $sideMirrors \wedge ignition = on$
   Prefix values: $Gear = Park$, $Button1$

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	$sideMirrors \wedge ignition = on$	T	*3*
	$sideMirrors \wedge ignition = off$	F	*1*

   (c) **Pre-state**: $1 \longrightarrow 3$
   Predicate: $seatBottom \wedge ignition = on$
   Prefix values: $Gear = Park$, $Button1$

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	$seatBottom \wedge ignition = on$	T	*3*
	$seatBottom \wedge ignition = off$	F	*1*

   (d) **Pre-state**: $1 \longrightarrow 3$
   Predicate: $lumbar \wedge ignition = on$
   Prefix values: $Gear = Park$, $Button1$

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	$lumbar \wedge ignition = on$	T	*3*
	$lumbar \wedge ignition = off$	F	*1*

(e) **Pre-state**: *1 ⟶ 3*
Predicate: *seatBack ∧ ignition = on*
Prefix values: *Gear = Park, Button1*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*seatBack ∧ ignition = on*	*T*	*3*
	seatBack ∧ ignition = off	*F*	*1*

(f) **Pre-state**: *2 ⟶ 1*
Predicate: *Button1 ∧ (Gear = Park ∨ ignition = off)*
Prefix values: *Gear = Park, Button2*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*Button1 ∧ Gear = Park*	*T*	*1*
	Button2 ∧ Gear = Park	*F*	*2*

(g) **Pre-state**: *2 ⟶ 3*
Predicate: *sideMirrors ∧ ignition = on*
Prefix values: *Gear = Park, Button2*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*sideMirrors ∧ ignition = on*	*T*	*3*
	sideMirrors ∧ ignition = off	*F*	*2*

(h) **Pre-state**: *2 ⟶ 3*
Predicate: *seatBottom ∧ ignition = on*
Prefix values: *Gear = Park, Button2*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*seatBottom ∧ ignition = on*	*T*	*3*
	seatBottom ∧ ignition = off	*F*	*2*

(i) **Pre-state**: *2 ⟶ 3*
Predicate: *lumbar ∧ ignition = on*
Prefix values: *Gear = Park, Button2*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*lumbar ∧ ignition = on*	*T*	*3*
	lumbar ∧ ignition = off	*F*	*2*

(j) **Pre-state**: *2 ⟶ 3*
Predicate: *seatBack ∧ ignition = on*
Prefix values: *Gear = Park, Button2*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*seatBack ∧ ignition = on*	*T*	*3*
	seatBack ∧ ignition = off	*F*	*2*

(k) **Pre-state**: *3 ⟶ 1*
Predicate: *Button1 ∧ (Gear = Park ∨ ignition = off)*
Prefix values: *Gear = Park, Button1, ignition=on, sideMirrors*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*Button1 ∧ Gear = Park*	*T*	*1*
	Button2 ∧ Gear = Park	*F*	*2*

(l) **Pre-state**: *3 ⟶ 2*
Predicate: *Button2 ∧ (Gear = Park ∨ ignition = off)*
Prefix values: *Gear = Park, Button1, ignition=on, sideMirrors*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*Button2 ∧ Gear = Park*	*T*	*2*
	Button1 ∧ Gear = Park	*F*	*1*

(m) **Pre-state**: *3 ⟶ 4*
Predicate: *Reset ∧ Button1 ∧ ignition = on*
Prefix values: *Gear = Park, Button1, ignition=on, sideMirrors*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*Reset ∧ Button1 ∧ ignition = on*	*T*	*4*
	Reset ∧ Button1 ∧ ignition = off	*F*	*3*

(n) **Pre-state**: *3 ⟶ 5*
Predicate: *Reset ∧ Button2 ∧ ignition = on*
Prefix values: *Gear = Park, Button1, ignition=on, sideMirrors*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*Reset ∧ Button2 ∧ ignition = on*	*T*	*5*
	Reset ∧ Button2 ∧ ignition = off	*F*	*3*

(o) **Pre-state**: *4 ⟶ 1*
Predicate: *ignition = off*
Prefix values: *Gear = Park, Button1, ignition = on, sideMirrors, Reset and Button1*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*ignition = off*	*T*	*1*
	ignition = on	*F*	*4*

(p) **Pre-state**: *4 ⟶ 3*
Predicate: *sideMirrors ∧ ignition = on*
Prefix values: *Gear = Park, Button1, ignition = on, sideMirrors, Reset and Button1*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*sideMirrors ∧ ignition = on*	*T*	*3*
	sideMirrors ∧ ignition = off	*F*	*1*

(q) **Pre-state**: *4 ⟶ 3*
Predicate: *seatBottom ∧ ignition = on*
Prefix values: *Gear = Park, Button1, ignition = on, sideMirrors, Reset and Button1*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*seatBottom ∧ ignition = on*	*T*	*3*
	seatBottom ∧ ignition = off	*F*	*1*

(r) **Pre-state**: *4 ⟶ 3*
Predicate: *lumbar ∧ ignition = on*
Prefix values: *Gear = Park, Button1, ignition = on, sideMirrors, Reset and Button1*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*lumbar ∧ ignition = on*	*T*	*3*
	lumbar ∧ ignition = off	*F*	*1*

(s) **Pre-state**: *4 ⟶ 3*
Predicate: *seatBack ∧ ignition = on*
Prefix values: *Gear = Park, Button1, ignition = on, sideMirrors, Reset and Button1*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*seatBack ∧ ignition = on*	*T*	*3*
	seatBack ∧ ignition = off	*F*	*1*

(t) **Pre-state**: *5 ⟶ 2*
Predicate: *ignition = off*
Prefix values: *Gear = Park, Button2, ignition = on, sideMirrors, Reset and Button2*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*ignition = off*	*T*	*2*
	ignition = on	*F*	*5*

(u) **Pre-state**: *5 ⟶ 3*
Predicate: *sideMirrors ∧ ignition = on*
Prefix values: *Gear = Park, Button2, ignition = on, sideMirrors, Reset and Button2*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*sideMirrors ∧ ignition = on*	*T*	*3*
	sideMirrors ∧ ignition = off	*F*	*2*

(v) **Pre-state**: *5 ⟶ 3*
Predicate: *seatBottom ∧ ignition = on*
Prefix values: *Gear = Park, Button2, ignition = on, sideMirrors, Reset and Button2*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*seatBottom ∧ ignition = on*	*T*	*3*
	seatBottom ∧ ignition = off	*F*	*2*

(w) **Pre-state**: *5 ⟶ 3*
Predicate: *lumbar ∧ ignition = on*
Prefix values: *Gear = Park, Button2, ignition = on, sideMirrors, Reset and Button2*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*lumbar ∧ ignition = on*	*T*	*3*
	lumbar ∧ ignition = off	*F*	*2*

(x) **Pre-state**: *5 ⟶ 3*
Predicate: *seatBack ∧ ignition = on*
Prefix values: *Gear = Park, Button2, ignition = on, sideMirrors, Reset and Button2*

Coverage Criteria	***Test Case Values***	***P***	***Expected Output*** *(Post-state)*
Predicate coverage	*seatBack ∧ ignition = on*	*T*	*3*
	seatBack ∧ ignition = off	*F*	*2*

2. For the `Memory Seat` finite state machine, complete the test sets for the correlated active clause coverage criterion (CACC) by satisfying the predicates, ensuring reachability, and computing the expected output.

   Instructor Solution Only

3. For the `Memory Seat` finite state machine, complete the test sets for the general inactive active clause coverage criterion (GICC) by satisfying the predicates, ensuring reachability, and computing the expected output.

   Instructor Solution Only

4. Redraw Figure 8.7 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 four resulting predicates. (You can omit the other predicates.) Then derive tests to satisfy CACC for the resulting predicates. How do these tests compare with the tests derived from the original graph?

   Instructor Solution Only

5. 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 (including the transition from Active to Active).

   Instructor Solution Only

6. 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 abstract tests to satisfy Predicate Coverage, Correlated Active Clause Coverage, and General Inactive Clause Coverage. (An abstract test is in terms of the model, not the implementation.)

   Instructor Solution Only

7. Implement the memory seat FSM. Design an appropriate input language to your implementation and turn the tests derived for question 1, 2, and 3 into test scripts. Run the tests.

   Instructor Solution Only

Chapter 9

Exercises, Section 9.1.1

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 9.1.2

1. Define mutation score.

   Instructor Solution Only

2. How is the mutation score related to coverage from Chapter 5?

   Instructor Solution Only

3. Consider the stream BNF in Section 9.1.1 and the ground string "`B 10 06.27.94`." Give three valid and three invalid mutants of the string.

   Instructor Solution Only

4. Consider the following BNF:

```
A  ::= O B | O M | O B M
O  ::= "w" | "x" | "s" | "m"
B  ::= "i" | "f" | "c" | "r"
M  ::= "o" | "t" | "p" | "a" | "h"
```

   (a) How many nonterminal symbols are in the grammar? **Solution:** *4*

   (b) How many terminal symbols are in the grammar? **Solution:** *13*

   (c) Write two strings that are valid according to the BNF. **Solution:** *116 possible strings, including: wio, xf*

   (d) For each of your two strings, give two valid mutants of the string. **Solution:** *wit, xr*

   (e) For each of your two strings, give two invalid mutants of the string. **Solution:** *wif, xw*

5. Consider the following BNF:

```
P  ::= I D Y | I Y D | D I Y | D Y I | Y I D | Y D I
I  ::= "j" | "j"
D  ::= "9" | "21"
Y  ::= "0" | "4"
```

   (a) How many nonterminal symbols are in the grammar? **Instructor Solution Only**

   (b) How many terminal symbols are in the grammar? **Instructor Solution Only**

   (c) Write two strings that are valid according to the BNF. **Instructor Solution Only**

   (d) For each of your two strings, give two valid mutants of the string. **Instructor Solution Only**

   (e) For each of your two strings, give two invalid mutants of the string. **Instructor Solution Only**

Exercises, Section 9.2

1. Provide reachability conditions, infection conditions, propagation conditions, and test case values to kill mutants 2, 4, 5, and 6 in Figure 9.1.

   Instructor Solution Only

2. Answer questions (a) through (d) for the mutant on line 5 in the method `findVal()`.

   (a) If possible, find test inputs that do **not reach** the mutant.

   Solution:
   `findVal`*: The mutant is always reached, even if* `x = null`*.*

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

   Solution:
   `findVal`*: Infection always occurs, even if* `x = null`*, because* `i` *always has the wrong value after initialization in the loop.*

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

   Solution:
   `findVal`*: As long as the last occurrence of* `val` *isn't at* `numbers[0]`*, the correct output is returned. Examples are:* `(numbers, val) = ([1, 1], 1)` *or* `([-1, 1], 1)` *or* `(null, 0)`*.*

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

   Solution:
   `findVal`*: Any input with* `val` *only in* `numbers[0]` *works. An example is:* `(numbers, val) = ([1, 0], 1)`

```
/**
 * Find last index of element
 *
 * @param numbers array to search
 * @param val value to look for
 * @return last index of val in numbers; -1 if absent
 * @throws NullPointerException if numbers is null
 */
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.  }
```

3. Answer questions (a) through (d) for the mutant on line 6 in the method `sum()`.

   (a) If possible, find test inputs that do **not** reach the mutant.

   Instructor Solution Only

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

   Instructor Solution Only

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

   Instructor Solution Only

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

Instructor Solution Only

```
/**
 * Sum values in an array
 *
 * @param x array to sum
 *
 * @return sum of values in x
 * @throws NullPointerException if x is null
 */
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.  }
```

4. Refer to the `patternIndex()` method in the `PatternIndex` program in Chapter 7. Consider Mutant A and Mutant B given below. Implementations are available on the book website in files `PatternIndexA.java` and `PatternIndexB.java`.

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

isPat = false;  // Original  (Inside the loops, not the declaration)
isPat = true;   // Mutant B
```

Answer the following questions for each mutant.

Instructor Solution Only

(a) If possible, design test inputs that do **not reach** the mutants.

Instructor Solution Only

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

Instructor Solution Only

(c) If possible, design test inputs that satisfy reachability and infection, but **not propagation** for the mutants.

Instructor Solution Only

(d) If possible, design test inputs that **strongly kill** the mutants.

Instructor Solution Only

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

Instructor Solution Only

6. 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

7. 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

8. The fundamental premise of mutation 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

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

   Instructor Solution Only

10. Look online for the tool Jester (`jester.sourceforge.net`), which is based on JUnit. Based on your reading, evaluate Jester as a mutation-testing tool.

    Instructor Solution Only

11. Download and install the Java mutation tool *muJava* from the book website. (Direct URL: `http://cs.gmu.edu/~offutt/mujava/`) Enclose the method `cal()` from question 6 inside a class, and use muJava to test `cal()`. Use all the operators. Design tests to kill all non-equivalent mutants. Note that a test case is a method call to `cal()`.

    Instructor Solution Only

    Instructor Solution Only

    (a) How many mutants are there?

    Instructor Solution Only

    Instructor Solution Only

    Instructor Solution Only

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

Instructor Solution Only

(c) What mutation score were you able to achieve before analyzing for equivalent mutants?

Instructor Solution Only

(d) How many equivalent mutants are there?

Instructor Solution Only

Instructor Solution Only

Instructor Solution Only

Exercises, Section 9.4

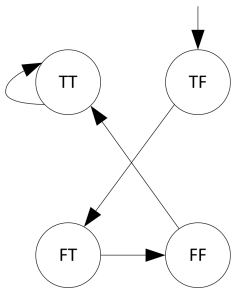
1. Translate the following SMV machine into a finite state machine.

```
MODULE main
#define false 0
#define true  1
VAR
      x, y : boolean;
ASSIGN
      init (x) := true;
      init (y) := true;

      next (x) := case
         x & y   : false;
         x       : true;
         !x & y  : false;
         !x & !y : true
         true    : x;
      esac;

      next (y) := case
         !x & y  : false
         y       : true
         !y      : false
         true    : y;
       esac;
```

Solution:
The graph should have a sequence of four states, in the order shown, with the final state looping back to itself:

TT –> FT –> FF –> TF –> TF

2. Translate the following finite state machine into an SMV machine.



Instructor Solution Only

3. (**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 9.5

1. Generate tests to satisfy TSC for the bank example grammar based on the BNF in Section 9.5.1. 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, state whether the test sequence is either (1) "in" the BNF, and give a derivation, or (2) 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 9.2.2. Succinctly describe any requirements or constraints on the inputs that are hard to model with the BNF.

   Instructor Solution Only

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 mutated version, 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 ordinary
numberPart    ::= ordinary^4
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 (in the grammar). For numbers not in the grammar, state why not.

- 123-4567
- 012-3456
- 109-1212
- 246-9900
- 113-1111

Instructor Solution Only

(b) Consider the following mutation of the grammar:

```
exchangePart ::= special  ordinary  other
```

If possible, give 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. Use the web application program `calculate` to answer the following questions. `calculate` is on the second author's website (at *https://cs.gmu.edu:8443/offutt/servlet/calculate* as of this writing).

   (a) Analyze the inputs for `calculate` and determine and write the grammar for the inputs. You can express the grammar in BNF, an XML schema, or another form if you think it's appropriate. Submit your grammar.

   Instructor Solution Only

   Modeling the `Result` and `Length` text boxes may be non-intuitive to some testers. They are clearly designed as ouput fields, yet the UI implements them as text boxes, allowing users to enter values. Thus they should be part of the grammar. This is a valuable example to discuss in class of something that is easy to overlook with "happy path" tests. Moreover, if the test designer shares the grammar with the software developers, the developers may question the `Result` and `Length` elements in the grammar and decide to change the software on the spot; becoming an opportunity to find faults in the software **before** running any tests.

   (b) Use the mutation ideas in this chapter to generate tests for `calculate`. Submit all tests; be sure to include expected outputs.

   Instructor Solution Only

   (c) Automate your tests using a web testing framework such as HttpUnit or Selenium. Submit screen printouts of any anomalous behavior.

   Instructor Solution Only

8. 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 at least 20 URLs from a small web site (such as course web pages). 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

9. 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.