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."*

`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.