☰ **Exercise Overview** (0/0 complete) ▼

2.14. Mutation Coverage Advanced Examples

This module presents various examples typically encountered by users of Mutation Testing. These include situations where branches of the code are not covered by the tests, and an example of code and tests that give 100% coverage but still has a bug.

2.14.1. Seeing the Effect: BST Range Query

One misconception for students is that they can get Mutation coverage for a branch by coming up with a test case that executes that branch. In fact, to get mutation testing coverage for a branch, there are two requirements.

1. Execute the branch.
2. Have execution of that branch affect the outcome of some test.

That is, in order to get coverage credit, the branch has to affect some assertion by a test case – typically either by changing the return value of some method whose value is being asserted, or by printing output that is being checked.

Example: Consider doing a range query on a BST, with the goal of visiting the minimum number of nodes. This means adding two checks to limit which children are being visited:

1. Only visit the right child if the root value is less than the range max.
2. Only visit the left child if the root value is greater than or equal to the range min.

A student might carefully construct test cases that properly avoid visiting children in all such cases — that is, executing all the branches. And if the requirement is to print node values in the range, this might be completely correct. Yet, none of that will get credit for mutation coverage of these branches. The reason is that visiting a child unnecessarily would not itself typically change whether the correct values are found during the search. The only likely difference in outcome from the checks on unnecessary visits come in the form of something like the count of the number of nodes visited. Therefore, in order to get credit for covering these branches, it is necessary that the tests actually check the outcome of something like the number of nodes visited, either by having this count returned by the search process (and the correct value verified by an assertion), or by checking that the correct node count is printed.

2.14.2. Over-Constrained Code: Visiting Quadrants

You may have a situation where your unit tests do not cover all branches of your code no matter how hard you try. This is not just an issue with Mutation Testing — in these situations, you wouldn't get code coverage on the affected lines either. In such cases, you should check whether you are writing over-constrained code, where execution of one branch "hides" or makes impossible execution of another.

Consider this example of a comparison of two points. You want to know which quadrant the second point (x2, y2) is in with respect to the first point (x1, y1).

```java
public class Quadrant {
  public static String getQuadrant(int x1, int y1, int x2, int y2) {
    if (x2 >= x1 && y2 >= y1) {
      return "South-East";
    } else if (x2 < x1 && y2 >= y1) {
      return "South-West";
    } else if (x2 < x1 && y2 < y1) {
      return "North-West";
    } else if (x2 >= x1 && y2 < y1) {
      return "North-East";
    }
  }
}
```

This has the virtue of being quite logical and clear. However, it has some problems. For one thing, it is relatively inefficient compared to alternatives, requiring many more arithmetic comparison tests. But our real concern has to do with testing and mutation coverage.

Fact: No series of tests will cover all branches in this code.

You can try yourself to verify this by carefully considering the logic of the code. Note that there are 4 possible outcomes (every test point has to be in one of the four quadrants) but at least 8 branches to test (there are 8 comparison operations). Can you think of test cases that will trigger each of the eight branches? With only four possible inputs, you cannot.

Looking at this more closely, consider that we can write tests that properly check all of the branches for the first `if` statement. These tests will properly fail when the comparisons are mutated to be replaced with `true` or `false`.

But consider what happens when we get to the second `if` statement. Ask yourself the question: Why are we here? The reason is that we have (already!) failed the first `if` test. Which means that there are preconditions to the fact that we are at the second `if` statement. Specifically, we know that we cannot have a point with both small x and small y, or we would not be here. So, consider what happens when we mutate the comparisons in the second `if` statement. In particular, if the comparison of y values is mutated to be `true`, then only a test having small x and y values (that is, in the South-East quadrant) will properly fail (all other points will do the right thing eventually). But no test with those values would be here, because it would have passed the first `if` test and gone down the South-East branch instead.

Note that this is not an issue with Mutation Testing as distinct from code coverage. It is not merely that the branch cannot be checked for the right answer, it cannot even be executed. So code coverage would also not count this branch.

Since we want complete mutation coverage for the four possible inputs We must come up with code that has only four branches. For example, our refactored code could look like this:

```
public class Quadrant {
  public static String getQuadrant(int x1, int y1, int x2, int y2) {
    if (x2 >= x1) {
      if (y2 >= y1) {
        return "South-East";
      } else {
        return "North-East";
      }
    } else {
      if (y2 >= y1) {
        return "South-West";
      } else {
        return "North-West";
      }
    }
  }
}
```

With the refactored code, not only can you test every branch, but this is a lot more efficient. Every branch requires two tests. (In contrast, the original code needed eight tests if it had to go through to the North-East banch.)

Writing overly complicated code is a common problem for many programmers. This is an example of how mutation testing can help you to improve the quality and efficiency of your code, by alerting you to over-constrained code blocks.

2.14.3. Over-Constrained Code: Deleting from a BST Tree

Consider a BST implementation that allows for duplicate key values (but the records are otherwise different). For example, perhaps the BST stores records about a city that includes its name (the BST key value) and location (what is just information carried with the record so far as the BST is concerned). This creates an important distinction for the delete operation: If a record is to be deleted, it has to be the correct record that is deleted, not just any record that happens to match that city name.

Here is a fairly typical approach to implementing the delete operation. Note that in this context, it is guaranteed that the method will not be called unless it has already been verified that this record is in the tree. (Perhaps the caller does a test to see that the record exists before actually calling the delete operation.) Therefore, a common “safety check” that the subtree is null (which happens only when the record is not in the tree) cannot actually be tested at the system level. (It might be testable by a class-specific unit test.)

```
// Return the subtree rooted by rt that has rec removed from it
Node removehelp(Node rt, Record rec) {
  // if (rt == null) { return null; } NOT TESTABLE IF WE KNOW THE RECORD IS THERE
  if (rt.value() > rec) {
    rt.setleft(removehelp(rt.left(), key));
  }
  else if (rt.value() < rec) {
    rt.setright(removehelp(rt.right(), key));
  }
  else if (!rt.value().equals(rec)) { // The names match, but it might not be the
    rt.setleft(removehelp(rt.left(), key)); // Equal valued keys go left in our in
  }
  else {
    // FOUND IT: Replace this node with an appropriate substitute
    ...
  }
```

This seems logical enough. We handle the cases of greater key values, lesser key values, and we have a special case when the names are equal but the records are not. Otherwise, we have found the record and we process it. Four situations, four cases. Unfortunately, we will find that it is not possible to test all branches of this code. This sort of thing can drive students crazy because they are convinced that their tests “cover” all the cases (because they do execute all the branches!), but nothing they do gets them complete mutation coverage. One red flag that there is trouble coming is that we have four branches, but really only three outcomes (note that we go left for two of these four conditions).

What is causing the problem? Consider how MT works: Independently for each of the four tests, it will first replace the expression with TRUE and run all the tests, following which it will replace the expression with FALSE and run all the tests. For each mutation, some test case must fail to get credit for covering that mutation.

Consider what happens with the first test, where small key values should go left. If MT sets this test to TRUE (so the method ALWAYS goes left), then it will fail at some point, which is what we want. But what if MT sets this test to always be FALSE? Then we get to the third test, and of course that happens to be true whenever we have a small key value (because it doesn’t match the search record). So we go left... which we should have done originally. Thus, no test case fails, so the mutant is never covered. The problem is that the third test makes the first test redundant.

Here is a slight revision to the code that IS testable to 100% MT coverage:

```
// Return the subtree rooted by rt that has rec removed from it
Node removehelp(Node rt, Record rec) {
  // if (rt == null) { return null; } DO NOT INCLUDE THIS IF WE KNOW THE RECORD IS
  if ((rt.value() >= rec) && (!rt.value().equals(rec))) { // Combine the two cases
    rt.setleft(removehelp(rt.left(), key));
  }
  else if (rt.value() < rec) {
```

```
  else if (rt.value() < rec) {
    rt.setright(removehelp(rt.right(), key));
  }
  else {
    // FOUND IT: Replace this node with an appropriate substitute
    ...
  }
}
```

Here we have merged the two cases for going left, so as to completely specify the required condition. This code is both testable and a bit simpler.

It is interesting to note that alternatively, we could simply have removed the case for small values going left (the first test). Because the second test sends big records to the right, the third (not equal records) test effectively also picks up the small-key-goes-left behavior. This change to the code is testable. It is a matter of taste as to which approach is easier to understand.

2.14.4. Over-Constrained Code: BigNum Exponents

This example is similar in spirit to the last one, but might be a bit simpler to follow. Consider writing a function that takes in a base value and an exponent where you are to calculate the value of the base to the exponent. We can write this function where we include two base cases (`exponent == 0` and `exponent == 1`), and two recursive calls for if the exponent is even or odd.

```
public int exponentiate(int base, int exponent) {
  if (exponent == 0) {
    return 1;
  }
  else if (exponent == 1) {
    return base;
  }
  else if (exponent % 2 == 0) {
    return exponentiate(base * base, exponent / 2);
  }
  else {
    return base * exponentiate(base, exponent - 1);
  }
}
```

This makes sense, we handle two special cases where the rules of exponentiation may get a little tricky, then we handle two general cases, where if an exponent is even or odd we calculate slightly different numbers. Four situations, four cases. Unfortunately, we will find that it is not possible to test all branches of this code — even if the tests “cover” all the cases (because they do execute all the branches!). As in the last example, we might realize there is a problem by noting that there are four branches, but really only three outcomes: base case, even case, and odd case. This is because while 0 is a true base case (it should not be treated like other even numbers), 1 is really an optimization because the odd case ends up handling it correctly.

Consider what happens with MT on the second condition, `else if (exponent == 1)`. If MT sets this test to TRUE, we always return the base number instead of the result of whatever the calculation intended. For example, exponentiate (8, 2) should return 64, but if the mutant makes the first else if TRUE, the code will return 8, allowing the mutant to be killed. But what happens when MT sets this condition to FALSE? Well that means our code will skip that base case, which seems like a good test should lead to a failure. But consider what happens next. If we modify our example to try exponentiate(8, 1) where MT set the first else if to FALSE, we end up in our else case where we multiply 8 * exponentiate(8, 0). The result of exponentiate is 1 because it hits our base case, thus we end up with 8 * 1 which is 8, which is correct. Therefore we did NOT catch the mutant. The “problem” here is that checking the test for 1 is redundant with (a special instance of) the odd condition. In some circumstances an optimization test like this might lead to runtime