CHAPTER SEVEN

Beautiful Tests

Alberto Savoia

MOST PROGRAMMERS HAVE HAD THE EXPERIENCE OF LOOKING AT A PIECE OF CODE and thinking it was not only functional but also *beautiful*. Code is typically considered beautiful if it does what it's supposed to do with unique elegance and economy.

But what about the tests for that beautiful code—especially the kind of tests that developers write, or *should* write, while they are working on the code? In this chapter, I am going to focus on tests, because tests can be beautiful themselves. More importantly, they can play a key role in helping you create more beautiful code.

As we will see, a combination of things makes tests beautiful. Unlike code, I can't bring myself to consider any *single* test beautiful—at least not in the same way I can look at, say, a sorting routine and call it beautiful. The reason is that testing, by its very nature, is a combinatorial and exploratory problem. Every `if` statement in the code requires at least two tests (one test for when the condition evaluates to true and one when it evaluates to false). An `if` statement with multiple conditions, such as:

```
if ( a || b || c )
```

could require, in theory, up to eight tests—one for each possible combination of the values of `a`, `b`, and `c`. Throw in control loops, multiple input parameters, dependencies on external code, different hardware and software platforms, etc., and the number and types of tests needed increases considerably.

Any nontrivial code, beautiful or not, needs not one, but a *team* of tests, where each test should be focused on checking a specific aspect of the code, similar to the way different players on a sports team are responsible for different tasks and different areas of the playing field.

Now that we have determined that we should evaluate tests in groups, we need to determine what characteristics would make a group of tests *beautiful*—an adjective rarely applied to them.

Generally speaking, the main purpose of tests is to instill, reinforce, or reconfirm our confidence that the code works properly and efficiently. Therefore, to me, the most beautiful tests are those that help me maximize my confidence that the code does, and will continue to do, what it's supposed to. Because different types of tests are needed to verify different properties of the code, the basic criteria for beauty vary. This chapter explores three ways tests can be beautiful:

Tests that are beautiful for their simplicity
: With a few lines of test code, I can document and verify the target code's basic behavior. By automatically running those tests with every build, I can ensure that the intended behavior is preserved as the code evolves. This chapter uses the JUnit testing framework for examples of basic tests that take minutes to write and keep paying dividends for the life of the project.

Tests that are beautiful because they reveal ways to make code more elegant, maintainable, and testable
: In other words, tests that help make code more beautiful. The process of writing tests often helps you realize not only logical problems, but also structural and design issues with your implementation. In this chapter, I demonstrate how, while trying to write tests, I have discovered a way to make my code more robust, readable, and well structured.

Tests that are beautiful for their breadth and depth
: Very thorough and exhaustive tests boost the developer's confidence that the code functions as expected, not only in some basic or handpicked cases, but in *all* cases. This chapter shows how I write and run this category of tests using the concept of *test theories*.

Because most developers are already familiar with basic testing techniques, such as smoke testing and boundary testing, I will spend most of the time on highly effective types of tests and testing techniques that are seldom discussed and rarely practiced.

That Pesky Binary Search

To demonstrate various testing techniques while keeping this chapter reasonably short, I need an example that's simple to describe and that can be implemented in a few lines of code. At the same time, the example must be juicy enough to provide some interesting testing challenges. Ideally, this example should have a long history of buggy implementations, demonstrating the need for thorough testing. And, last but not least, it would be great if this example itself could be considered beautiful code.

It's hard to talk about beautiful code without thinking about Jon Bentley's classic book *Programming Pearls* (Addison-Wesley). As I was rereading the book, I hit the beautiful code example I was looking for: a binary search.

As a quick refresher, a binary search is a simple and effective algorithm (but, as we'll see, tricky to implement correctly) to determine whether a presorted array of numbers `x[0..n-1]` contains a target element `t`. If the array contains `t`, the program returns its position in the array; otherwise, it returns `-1`.

Here's how Jon Bentley described the algorithm to the students:

> Binary search solves the problem by keeping track of the range within the array that holds t (if t is anywhere in the array). Initially, the range is the entire array. The range is shrunk by comparing its middle element to t and discarding half the range. The process continues until t is discovered in the array or until the range in which it must lie is known to be empty.

He adds:

> Most programmers think that with the above description in hand, writing the code is easy. They are wrong. The only way to believe this is by putting down this column right now and writing the code yourself. Try it.

I second Bentley's suggestion. If you have never implemented binary search, or haven't done so in a few years, I suggest you try that yourself before going forward; it will give you greater appreciation for what follows.

Binary search is a great example because it's so simple and yet it's so easy to implement incorrectly. In *Programming Pearls*, Jon Bentley shares how, over the years, he asked hundreds of professional programmers to implement binary search after providing them with a description of the basic algorithm. He gave them a very generous two hours to write it, and even allowed them to use the high-level language of their choice (including pseudocode). Surprisingly, only about 10 percent of the professional programmers implemented binary search correctly.

More surprisingly, in his *Sorting and Searching*,* Donald Knuth points out that even though the first binary search was published in 1946, it took 12 more years for the first binary search *without bugs* to be published.

* *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Second Edition, Addison-Wesley, 1998.

But most surprising of all is that even Jon Bentley's official and *proven* algorithm, which (I must assume) has been implemented and adapted thousands of times, turns out to have a problem that can manifest itself when the array is big enough and the algorithm is implemented in a language with fixed-precision arithmetic.

In Java, the bug manifests itself by throwing an `ArrayIndexOutOfBoundsException`, whereas in C, you get an array index out of bounds with unpredictable results. You can read more about this latest bug in Joshua Bloch's blog: *http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html*.

Here is a Java implementation with the infamous bug:

```
public static int buggyBinarySearch(int[] a, int target) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < target)
            low = mid + 1;
        else if (midVal > target)
            high = mid - 1;
        else
            return mid;
    }
    return -1;
}
```

The bug is in the following line:

```
int mid = (low + high) / 2;
```

If the sum of `low` and `high` is greater than `Integer.MAX_VALUE` (which is $2^{31} - 1$ in Java), it overflows into a negative number and, of course, stays negative when divided by 2—ouch!

The recommended solution is to change the calculation of the midpoint to prevent integer overflow. One way to do it is by subtracting instead of adding:

```
int mid = low + ((high - low) / 2);
```

Or, if you want to show off your knowledge of bit shift operators, the blog (and the official Sun Microsystems bug report*) suggests using the unsigned bit shift, which is probably faster but may be obscure to most Java developers (including me):

```
int mid = (low + high) >>> 1;
```

Considering how simple the idea behind binary search is, and the sheer number and collective brain power of the people that have worked on it over the years, it's a great example of why even the simplest code needs testing—and lots of it. Joshua Bloch expressed this beautifully in his blog about this bug:

* *http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5045582*.

> The general lesson that I take away from this bug is humility: It is hard to write even the smallest piece of code correctly, and our whole world runs on big, complex pieces of code.

Here is the implementation of binary search I want to test. In theory, the fix to the way the `mid` is calculated should resolve the final bug in a pesky piece of code that has eluded some of the best programmers for a few decades:

```
public static int binarySearch(int[] a, int target) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) >>> 1;
        int midVal = a[mid];

        if (midVal < target)
            low = mid + 1;
        else if (midVal > target)
            high = mid - 1;
        else
            return mid;
    }
    return -1;
}
```

This version of `binarySearch` looks right, but there might still be problems with it. Perhaps not bugs, but things that can and should be changed. The changes will make the code not only more robust, but more readable, maintainable, and testable. Let's see whether we can discover some interesting and unexpected opportunities for improvement as we test it.

Introducing JUnit

When speaking of beautiful tests, it's hard not to think of the JUnit testing framework. Because I'm using Java, deciding to build my beautiful tests around JUnit was a very easy decision. But before I do that, in case you are not already familiar with JUnit, let me say a few words about it.

JUnit is the brainchild of Kent Beck and Erich Gamma, who created it to help Java developers write and run automated and self-verifying tests. It has the simple, but ambitious, objective of making it easy for software developers to do what they should have done all along: test their own code.

Unfortunately, we still have a long way to go before the majority of developers are test-infected (i.e., have experimented with developer testing and decided to make it a regular and important part of their development practices). However, since its introduction, JUnit (helped considerably by eXtreme Programming and other Agile methodologies, where developer involvement in testing is nonnegotiable) has gotten more programmers to write

tests than anything else.* Martin Fowler summed up JUnit's impact as follows: "Never in the field of software development was so much owed by so many to so few lines of code."

JUnit is intentionally simple. Simple to learn. Simple to use. This was a key design criterion. Kent Beck and Erich Gamma took great pains to make sure that JUnit was so easy to learn and use that programmers would actually use it. In their own words:

> So, the number one goal is to write a framework within which we have some glimmer of hope that developers will actually write tests. The framework has to use familiar tools, so that there is little new to learn. It has to require no more work than absolutely necessary to write a new test. It has to eliminate duplicate effort.†

The official getting-started documentation for JUnit (the *JUnit Cookbook*) fits in less than two pages: *http://junit.sourceforge.net/doc/cookbook/cookbook.htm*.

Here's the key extract from the cookbook (from the 4.x version of JUnit):

> When you need to test something, here is what you do:
>
> 1. Annotate a method with `@org.junit.Test`
> 2. When you want to check a value, import `org.junit.Assert.*` statically, call `assertTrue( )`, and pass a Boolean that is true if the test succeeds
>
> For example, to test that the sum of two Moneys with the same currency contains a value that is the sum of the values of the two Moneys, write:

```
@Test
public void simpleAdd( ) {
    Money m12CHF= new Money(12, "CHF");
    Money m14CHF= new Money(14, "CHF");
    Money expected= new Money(26, "CHF");
    Money result= m12CHF.add(m14CHF);
    assertTrue(expected.equals(result));
}
```

If you have any familiarity with Java, those two instructions and the simple example are all you need to get started. That's also all you need to understand the tests I will be writing. Beautifully simple, isn't it? So, let's get going.

Nailing Binary Search

Given its history, I am not going to be fooled by the apparent simplicity of binary search, or by the obviousness of the fix, especially because I've never used the unsigned bit shift operator (i.e., >>>) in any other code. I am going to test this *fixed* version of binary search as if I had never heard of it before, nor implemented it before. I am not going to trust anyone's word, or tests, or proofs, that this time it will really work. I want to be confident that it works as it should through my own testing. I want to *nail* it.

* Another indication of JUnit's success and influence is that today there are JUnit-inspired frameworks for most modern programming languages, as well as many JUnit extensions.

† "JUnit: A Cook's Tour," Kent Beck and Erich Gamma: *http://junit.sourceforge.net/doc/cookstour/cookstour.htm*.

Here's my initial testing strategy (or team of tests):

- Start with *smoke tests*.
- Add some *boundary value* tests.
- Continue with various thorough and exhaustive types of tests.
- Finally, add some *performance* tests.

Testing is rarely a linear process. Instead of showing you the finished set of tests, I am going to walk you through my thought processes while I am working on the tests.

Smoking Allowed (and Encouraged)

Let's get started with the smoke tests. These are designed to make sure that the code does the right thing when used in the most basic manner. They are the first line of defense and the first tests that should be written, because if an implementation does not pass the smoke tests, further testing is a waste of time. I often write the smoke tests before I write the code; this is called *test-driven development* (or *TDD*).

Here's my smoke test for binary search:

```
import static org.junit.Assert.*;
import org.junit.Test;

public class BinarySearchSmokeTest {

@Test
public void smokeTestsForBinarySearch( ) {

    int[] arrayWith42 = new int[] { 1, 4, 42, 55, 67, 87, 100, 245 };
    assertEquals(2, Util.binarySearch(arrayWith42, 42));
    assertEquals(-1, Util.binarySearch(arrayWith42, 43));

    }

}
```

As you can tell, this test is really, *really*, basic. Not a huge confidence builder by itself, but still beautiful because it's a very fast and efficient first step toward more thorough tests.

Because this smoke test executes extremely fast (in less than 1/100th of a second on my system), you might ask why I didn't include a few more tests. The answer is that part of the beauty of smoke tests is that they can continue to pay dividends after the bulk of the development is done. To reconfirm my confidence in the code—call it "confidence maintenance"—I like to combine all smoke tests into a suite that I run every time I do a new build (which might be dozens of times a day), and I want this smoke test suite to run fast—ideally in a minute or two. If you have thousands of classes, and thousands of smoke tests, it's essential to keep each one to a bare minimum.

Pushing the Boundaries

As the name implies, boundary testing is designed to explore and validate what happens when the code has to deal with extremes and corner cases. In the case of binary search, the two parameters are the array and the target value. Let's think of some boundary cases for each of these parameters.*

The first set of interesting corner cases that come to mind has to do with the size of the array being searched. I begin with the following basic boundary tests:

```
int[] testArray;

@Test
public void searchEmptyArray( ) {
    testArray = new int[] {};
    assertEquals(-1, Util.binarySearch(testArray, 42));
}

@Test
public void searchArrayOfSizeOne( ) {
    testArray = new int[] { 42 };
    assertEquals(0, Util.binarySearch(testArray, 42));
    assertEquals(-1, Util.binarySearch(testArray, 43));
}
```

It's pretty clear that an empty array is a good boundary case, and so is an array of size 1 because it's the smallest nonempty array. Both of these tests are beautiful because they increase my confidence that the right thing happens at the lower boundary of array size.

But I also want to test the search with a very large array, and this is where it gets interesting (especially with the hindsight knowledge that the bug manifests itself only on arrays with over one billion elements).

My first thought is to create an array large enough to ensure that the integer-overflow bug has been fixed, but I immediately recognize a testability issue: my laptop does not have enough resources to create an array that large in memory. But I know that there are systems that *do* have many gigabytes of memory and keep large arrays in memory. I want to make sure, one way or another, that the `mid` integer does not overflow in those cases.

What can I do?

I know that by the time I am done with some of the other tests I have in mind, I will have enough tests to give me confidence that the basic algorithm and implementation works *provided that the midpoint is calculated correctly and does not overflow into a negative number*. So, here's a summary of my reasoning, leading to a possible testing strategy for enormous arrays:

* The specification for binary search says that the array *must* be sorted prior to making this call, and that if it is not sorted, the results are undefined. We are also assuming that a null array parameter should throw a `NullPointerException`. Because most readers should already be familiar with basic boundary testing techniques, I am going to skip some of those obvious tests.

1. I cannot test `binarySearch` directly with arrays large enough to verify that the overflow bug in the calculation of `mid` does not occur anymore.
2. However, I *can* write enough tests to give me confidence that my `binarySearch` implementation works correctly on smaller arrays.
3. I can also test the way *mid* is calculated when very large values are used, without getting arrays involved.
4. So, if I can gain enough confidence through testing that:
   - My implementation of the basic `binarySearch` algorithm is sound as long as `mid` is calculated correctly, and
   - The way the midpoint is calculated is correct

   then I can have confidence that `binarySearch` will do the right thing on very large arrays.

So the not-so-obvious, but beautiful, testing strategy is to isolate and test the pesky, overflow-prone calculation independently.

One possibility is to create a new method:

```
static int calculateMidpoint(int low, int high) {
    return (low + high) >>> 1;
}
```

then change the following line in the code from:

```
int mid = (low + high) >>> 1;
```

to:

```
int mid = calculateMidpoint(low, high);
```

and then test the heck out of the `calculateMidpoint` method to make sure it always does the right thing.

I can already hear a few of you screaming about adding the overhead of a method call in an algorithm designed for maximum speed. But there's no need to cry foul. Here's why I believe this change to the code is not only acceptable, but the right thing to do:

1. These days, I can trust compiler optimization to do the right thing and inline the method for me, so there is no performance penalty.
2. The change makes the code more readable. I checked with several other Java programmers, and most of them were not familiar with the unsigned bit shift operator, or were not 100 percent sure how it worked. For them, seeing `calculateMidpoint(low, high)` is more obvious than seeing `(low + high) >>> 1`.
3. The change makes the code more testable.

This is actually a good example of how the very act of creating a test for your code will improve its design or legibility. In other words, testing can help you make your code more beautiful.

Here is a sample boundary test for the new `calculateMidpoint` method:

```
@Test
public void calculateMidpointWithBoundaryValues( ) {
    assertEquals(0, calculateMidpoint (0, 1));
    assertEquals(1, calculateMidpoint (0, 2));
    assertEquals(1200000000, calculateMidpoint (1100000000, 1300000000));
    assertEquals(Integer.MAX_VALUE - 2,
        calculateMidpoint (Integer.MAX_VALUE-2, Integer.MAX_VALUE-1));
    assertEquals(Integer.MAX_VALUE - 1,
        calculateMidpoint (Integer.MAX_VALUE-1, Integer.MAX_VALUE));
}
```

I run the tests, and they pass. Good. I am now confident that calculating `mid` using the unfamiliar operator does what it's supposed to do within the range of array sizes I want to handle with this implementation of binary search.

The other set of boundary cases has to do with the position of the target number. I can think of three obvious boundary cases for the target item location: first item in the list, last item in the list, and right smack in the middle of the list. So, I write a simple test to check these cases:

```
@Test
public void testBoundaryCasesForItemLocation( ) {
    testArray = new int[] { -324, -3, -1, 0, 42, 99, 101 };
    assertEquals(0, Util.binarySearch(testArray, -324));  // first position
    assertEquals(3, Util.binarySearch(testArray, 0));      // middle position
    assertEquals(6, Util.binarySearch(testArray, 101));   // last position
}
```

Note that in this test I used some negative numbers and 0, both in the array and for the target number. It had occurred to me, while reading the tests I had already written, that I had used only positive numbers. Since that's not part of the specification, I should introduce negative numbers and 0 in my tests. Which leads me to the following piece of testing wisdom:

> The best way to think of more test cases is to start writing *some* test cases.

Now that I started to think about positive/negative numbers and 0, I realize that it would be good to have a couple of tests that use the minimum and maximum integer values.

```
public void testForMinAndMaxInteger( ) {
    testArray = new int[] {
      Integer.MIN_VALUE, -324, -3, -1, 0, 42, 99, 101, Integer.MAX_VALUE
    };
    assertEquals(0, Util.binarySearch(testArray, Integer.MIN_VALUE));
    assertEquals(8, Util.binarySearch(testArray, Integer.MAX_VALUE));
}
```

So far, all the boundary cases I thought of passed, and I am starting to feel pretty confident. But then I think of the 90 percent of professional programmers in Jon Bentley's class who implemented binary search and thought they had it right but didn't, and my confidence begins to wane a little bit. Did I make any unwarranted assumptions about the inputs? I did not think about negative numbers and 0 until this last test case. What other

unwarranted assumptions have I made? Because I handcrafted the tests, perhaps I subconsciously created cases that would work and missed ones that would fail.

This is a known problem with programmers testing their own code. If they can't think of some scenarios when implementing the code, it's likely that they will not be able to think of them when they switch context and try to *break* the code. Truly beautiful testing requires a developer to make an extra effort, think outside the box, explore weird scenarios, look for weaknesses, and try to break things.

So, what haven't I thought of? My smoke test and boundary tests do not feel sufficient. Is my test set representative enough that I can, through some form of induction,* claim the code will work in all instances? The words of Joshua Bloch echo in my mind: *"…It is hard to write even the smallest piece of code correctly."*

What kind of tests would make me feel confident enough that my implementation will do the right thing with all sorts of inputs—not just the ones I handcrafted?

Random Acts of Testing

So far I've written traditional, tried-and-true types of tests. I used a few concrete examples to test the search code against my expectations of what the correct behavior should be in those cases. Those tests all pass, so I have *some* level of confidence in my code. But I also realize that my tests are very specific and cover only a very small subset of all the possible inputs. What I would like, and what would help me sleep at night knowing my code has been thoroughly covered, is a way of testing over a much broader set of inputs. For this to happen I need two things:

1. A way to generate a large and diverse set of inputs
2. A set of generalized assertions that will work on any input

Let's tackle the first requirement.

What I need here is a way to generate arrays of integers of all shapes and sizes. The only requirement I am going to make is that the resulting arrays are sorted, because that's a precondition. Other than that, anything goes. Here's my initial implementation of the generator:†

```
public int[] generateRandomSortedArray(int maxArraySize, int maxValue) {
    int arraySize = 1 + rand.nextInt(maxArraySize);
    int[] randomArray = new int[arraySize];
    for (int i = 0; i < arraySize; i++) {
        randomArray[i] = rand.nextInt(maxValue);
    }
    Arrays.sort(randomArray);
    return randomArray;
}
```

* By *induction,* I mean deriving general principles from particular facts or instances.

† I say *initial* implementation because I quickly realized that I needed to populate the array with negative as well as positive numbers, and changed the generator accordingly.

For my generator, I take advantage of `java.util`'s random-number generator and `Arrays` utilities. The latter once contained the very same binary-search bug Joshua Bloch mentioned in his blog, but it's fixed in the version of Java I am using. Because I already covered the handling of empty arrays to my satisfaction in my other tests, I use a minimum array size here of 1. The generator is parameterized because I might want to create different sets of tests as I go along: some with small arrays containing big numbers, some with big arrays and small numbers, and so on.

Now I have to come up with some general statements about the desired behavior of the binary search that can be expressed as assertions. By "general," I mean statements that must hold true for any input array and target value. My colleagues Marat Boshernitsan and David Saff call these *theories.* The idea is that we have a theory of how the code should behave, and the more we test the theory, the more confident we can be that what we theorize is actually true. In the following example, I am going to apply a much simplified version of Saff and Boshernitsan's theories.

Let's try to come up with some theories for `binarySearch`. Here we go:

> For all instances of *`testArray`* and *`target`*, where *`testArray`* is a sorted array of integers and is not null, and *`target`* is an integer, the following must always be true of `binarySearch`:
>
> Theory 1:* If `binarySearch(`*`testArray`*`,` *`target`*`)` returns –1, then *`testArray`* does not contain *`target`*.
>
> Theory 2: If `binarySearch(`*`testArray`*`,` *`target`*`)` returns *n*, and *n* is greater than or equal to 0, then *`testArray`* contains *`target`* at position *n*.

Here's my code for testing these two theories:

```
public class BinarySearchTestTheories {

Random rand;

@Before
public void initialize( ) {
    rand = new Random( );
}

@Test
public void testTheories( ) {

    int maxArraySize = 1000;
    int maxValue = 1000;
    int experiments = 1000;
```

* In practice I would use, and recommend using, descriptive names for the theories, such as: *binarySearchReturnsMinusOneImpliesArrayDoesNotContainElement*, but I found that for this chapter, the reasoning is easier to follow if I use Theory1, Theory2, etc.

```
    int[] testArray;
    int target;
    int returnValue;

    while (experiments-- > 0) {
        testArray = generateRandomSortedArray(maxArraySize, maxValue);
        if (rand.nextBoolean()) {
            target = testArray[rand.nextInt(testArray.length)];
        } else {
            target = rand.nextInt();
        }
        returnValue = Util.binarySearch(testArray, target);
        assertTheory1(testArray, target, returnValue);
        assertTheory2(testArray, target, returnValue);
    }
}

public void assertTheory1(int[] testArray, int target, int returnValue) {
    if (returnValue == -1)
        assertFalse(arrayContainsTarget(testArray, target));
}

public void assertTheory2(int[] testArray, int target, int returnValue) {
    if (returnValue >= 0)
        assertEquals(target, testArray[returnValue]);
}

public boolean arrayContainsTarget(int[] testArray, int target) {
    for (int i = 0; i < testArray.length; i++)
        if (testArray[i] == target)
            return true;
    return false;
}
```

In the main test method, `testTheories`, I decide how many experiments I want to run in order to confirm the theories, and use that as my loop counter. Inside the loop, the random-array generator I just wrote gives me a sorted array. I want to test both successful and unsuccessful searches, so I use Java's random number generator again to "toss a coin" (through the `rand.nextBoolean()` code). Based on the virtual coin toss, I decide whether I am going to pick a target number that I *know* is in the array or one that's unlikely to be in the array. Finally, I call `binarySearch`, store the return value, and invoke the methods for the theories I have so far.

Notice that, in order to implement the tests for my theories, I had to write a test helper method, `arrayContainsTarget`, that gives me an alternative way of checking whether `testArray` contains the target element. This is a common practice for this type of testing. Even though the implementation of this helper method provides functionality similar to `binarySearch`, it's a much simpler (albeit much slower) search implementation. I have confidence that the helper does the right thing, so I can use it to test an implementation I am much less sure about.

I start by running 1,000 experiments on arrays of size up to 1,000. The tests take a fraction of a second, and everything passes. Good. Time to explore a little more (remember that testing is an exploratory activity).

I change the experiment and `maxArraySize` values to 10,000, then 100,000. The tests now take closer to a minute, and my CPU maxes out. I feel like I am giving the code a really good workout.

My confidence is building, but one of my beliefs is: *If all your tests pass, chances are that your tests are not good enough.* What other properties should I test now that I have this framework?

I think for a bit and notice that my two theories are both of the form:

> If something is true about the return value of `binarySearch`, then something else must be true about the `testArray` and the `target`.

In other words, I have logic of the form *p* implies *q* (or, $p \rightarrow q$, using logic notation), which means I am only testing half of what I should be testing. I should also have tests of the form $q \rightarrow p$:*

> If something is true about the `testArray` and the `target`, then something else must be true about the return value.

This is a bit tricky, but important, so let me clarify with some specifics. The tests for Theory 1 verify that when the return value is –1, the target element is not in the array. But they don't verify that when the target element is not in the array, the return value is –1. In other words: *if I only had this one theory with which to test*, an implementation that returned –1 sometimes, but not *every* time it should, would still pass all my tests. A similar problem exists with Theory 2.

I can demonstrate this with *mutation testing*, a technique for *testing the tests* invented by Jeff Offut. The basic idea is to mutate the code under tests with some known bugs. If the tests you have still pass despite the bug in the code, then the tests are probably not as thorough as they need to be.

Let me mutate `binarySearch` in some drastic and arbitrary way. I'll try do this: if `target` is greater than 424242 and `target` is not contained in the array, instead of returning –1, I am going to return *–42*. How's that for software vandalism? See the tail end of the following code:

* Of course, either *p*, *q*, or both, could be negated (e.g., $\sim p \rightarrow \sim q$, or $p \rightarrow \sim q$). I am arbitrarily using *p* and *q* as stand-ins for any predicate about the return value and the array parameter, respectively. What's important here is to recognize that when you are programming, you typically think in terms of $p \rightarrow q$ (if *p* is true, then *q* must happen—the so-called *happy path*: the normal, most common usage of the code). When you are testing, however, you must force yourself to think both backward ($q \rightarrow ?$, or if *q* is true what must be true about *p*?), and in negative terms (if p is not true [i.e., $\sim p$], what must be true about *q*?).

```
public static int binarySearch(int[] a, int target) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < target)
            low = mid + 1;
        else if (midVal > target)
            high = mid - 1;
        else
            return mid;
    }
    if (target <= 424242)
        return -1;
    else
        return -42;
}
```

Hopefully, you'll agree that this is a pretty big mutation: the code returns an unexpected and unspecified value if the target is a number greater than 424242 and is not contained in the array. And yet, all the tests we have written so far pass with flying colors.

We definitely need to add at least a couple more theories to make the tests tighter and catch this category of mutations:

> Theory 3: If *testArray* does not contain *target*, then it must return -1.
>
> Theory 4: If *testArray* contains *target* at position *n*, then binarySearch(*testArray*, *target*) must return *n*.

These theories are tested as follows:

```
public void assertTheory3(int[] testArray, int target, int returnValue) {
    if (!arrayContainsTarget(testArray, target))
        assertEquals(-1, returnValue);
}

public void assertTheory4(int[] testArray, int target, int returnValue) {
        assertEquals(getTargetPosition(testArray, target), returnValue);
}

public int getTargetPosition(int[] testArray, int target) {
    for (int i = 0; i < testArray.length; i++)
        if (testArray[i] == target)
            return i;
    return -1;
}
```

Notice that I had to create another helper method, getTargetPosition, which has exactly the same behavior as binarySearch (but I am confident that it works properly, with the huge downside that it requires up to n instead of $\log_2 n$ comparisons). Because

`getTargetPosition` is very similar to `arrayContainsTarget`, and code duplication is bad, I rewrite the latter as follows:

```
public boolean arrayContainsTarget(int[] testArray, int target) {
    return getTargetPosition(testArray, target) >= 0;
}
```

I run these tests again with my random-array generator, and now the `return -42` mutation is caught immediately. Good, that helps my confidence. I remove the intentional bug and run the tests again. I expect them to pass, but they don't. Some tests for Theory 4 are not passing. JUnit is failing with messages of the form:

```
expected:<n> but was:<n + 1>
```

Theory 4 says that:

> If *testArray* contains *target* at position *n*, then `binarySearch`(*testArray*, *target*) must return *n*.

So, in some cases, the search routine is returning a location that's off by one. How's that possible?

I need a bit more data. JUnit's assertions can accept a message of type `String` as the first parameter, so I change Theory 4's `assertEqual` to include some text that will give me more information when it fails:

```
public void assertTheory4(int[] testArray, int target, int returnValue) {
  String testDataInfo = "Theory 4 - Array=" +
        printArray(testArray)
        + " target="
        + target;
  assertEquals(testDataInfo, getTargetPosition(testArray, target), returnValue);
}
```

Now, whenever Theory 4 fails to hold, JUnit will show me the contents of the array as well as the target value. I run the tests again (with small values of `maxArraySize` and `maxValue` to make the output easier to read) and get the following:

```
java.lang.AssertionError: Theory 4 - Array=[2, 11, 36, 66, 104, 108, 108, 108, 122,
155, 159, 161, 191] target=108 expected:<5> but was:<6>
```

I see what's happening. Theory 4 does not take into account duplicate values, and I hadn't thought of that. There are three instances of the number 108. I guess I need to find out what the specification is for handling duplicate values, and fix either the code or my theory and tests. But I'll leave this as an exercise to the reader (I always wanted to say that!) because I am running out of space, and I want to say a few words about performance tests before we wrap up this chapter.

Performance Anxiety

The tests we've already run based on these theories put a pretty tight net around the implementation. It's going to be tough to pass all these tests and still have a buggy implementation. But there is something we overlooked. All the tests we have are good tests for search, but what we are testing is specifically a *binary* search. We need a set of tests for *binary-ness*. We need to see whether the number of comparisons our implementation performs matches the expectations of a maximum of $\log_2 n$ comparisons. How can we go about this?

My first thought is to use the system clock, but I quickly dismiss the idea because the clock I have available does not have enough resolution for this particular challenge (binary search is blazingly fast), and I can't really control the execution environment. So, I use another developer testing trick: I create an alternate implementation of `binarySearch` called `binarySearchComparisonsCount`. This version of the code uses the same logic as the original, but it keeps a count of the comparisons and returns that number instead of –1 or the target location.* Here's that code:

```
public static int binarySearchComparisonCount(int[] a, int target) {
    int low = 0;
    int high = a.length - 1;

    int comparisonCount = 0;

    while (low <= high) {

        comparisonCount++;

        int mid = (low + high) >>> 1;
        int midVal = a[mid];

        if (midVal < target)
            low = mid + 1;
        else if (midVal > target)
            high = mid - 1;
        else
            return comparisonCount;
    }
    return comparisonCount;
}
```

Then I create another theory based on that code:

> Theory 5: If the size of *testArray* is *n*, then *binarySearchComparisonCount*(*testArray*, *target*) must return a number less than, or equal to, 1 + log2 *n*.

* Instead of modifying `binarySearch` to return the comparison count, a better, cleaner, and more object-oriented design (suggested by David Saff) would be to create a `CountingComparator` class that implements Java's generalized Comparator interface and to modify `binarySearch` to take an instance of that class as a third parameter. This would generalize `binarySearch` to work with types other than integers, another example of how testing can lead to better design and more beautiful code.

Here's the code for the theory:

```
public void assertTheory5(int[] testArray, int target) {
    int numberOfComparisons =
        Util.binarySearchComparisonCount(testArray, target);
    assertTrue(numberOfComparisons <= 1 + log2(testArray.length));
}
```

I add this latest theory to my existing list inside the method `testTheories`, which now looks like this:

```
...
    while (experiments-- > 0) {
        testArray = generateRandomSortedArray();
        if (rand.nextInt() % 2 == 0) {
            target = testArray[rand.nextInt(testArray.length)];
        } else {
            target = rand.nextInt();
        }
        returnValue = Util.binarySearch(testArray, target);
        assertTheory1(testArray, target, returnValue);
        assertTheory2(testArray, target, returnValue);
        assertTheory3(testArray, target, returnValue);
        assertTheory4(testArray, target, returnValue);
        assertTheory5(testArray, target);
    }
...
```

I run a few tests with a `maxArraySize` set of a few different values, and I find that Theory 5 seems to be holding strong.

Because it's almost noon, I set the number of `experiments` to 1,000,000 and go to lunch while my computer crunches away and tests each theory a million times.

When I get back, I see that all my tests pass. There are probably a couple more things that I would want to test, but I have made great progress in boosting *my* confidence in this implementation of `binarySearch`. Because different developers have different backgrounds, styles, and levels of experience, you might have focused on different areas of the code. A developer already familiar with the unsigned shift operator, for example, would not feel the same need I had to test it.

In this section, I wanted to give you a flavor of performance testing and show you how you could gain insight into and confidence in your code's performance by combining code instrumentation with test theories. I highly recommend you study Chapter 3, where Jon Bentley gives this important topic the attention and beautiful treatment it deserves.

Conclusion

In this chapter, we have seen that even the best developers and the most beautiful code can benefit from testing. We have also seen that writing test code can be every bit as creative and challenging as writing the target code. And, hopefully, I've shown you that tests themselves can be considered beautiful in at least three different ways.

Some tests are beautiful for their simplicity and efficiency. With a few lines of JUnit code, run automatically with every build, you can document the code's intended behavior and boundaries, and ensure that both of them are preserved as the code evolves.

Other tests are beautiful because, in the process of writing them, they help you improve the code they are meant to test in subtle but important ways. They may not discover proper bugs or defects, but they bring to the surface problems with the design, testability, or maintainability of the code; they help you make your code more beautiful.

Finally, some tests are beautiful for their breadth and thoroughness. They help you gain confidence that the functionality and performance of the code match requirements and expectations, not just on a few handpicked examples, but with a wide range of inputs and conditions.

Developers who want to write beautiful code can learn something from artists. Painters regularly put down their brushes, step away from the canvas, circle it, cock their heads, squint, and look at it from different angles and under different lights. They need to develop and integrate those perspectives in their quest for beauty. If your canvas is an IDE and your medium is code, think of testing as your way of stepping away from the canvas to look at your work with critical eyes and from different perspectives—it will make you a better programmer and help you create more beautiful code.