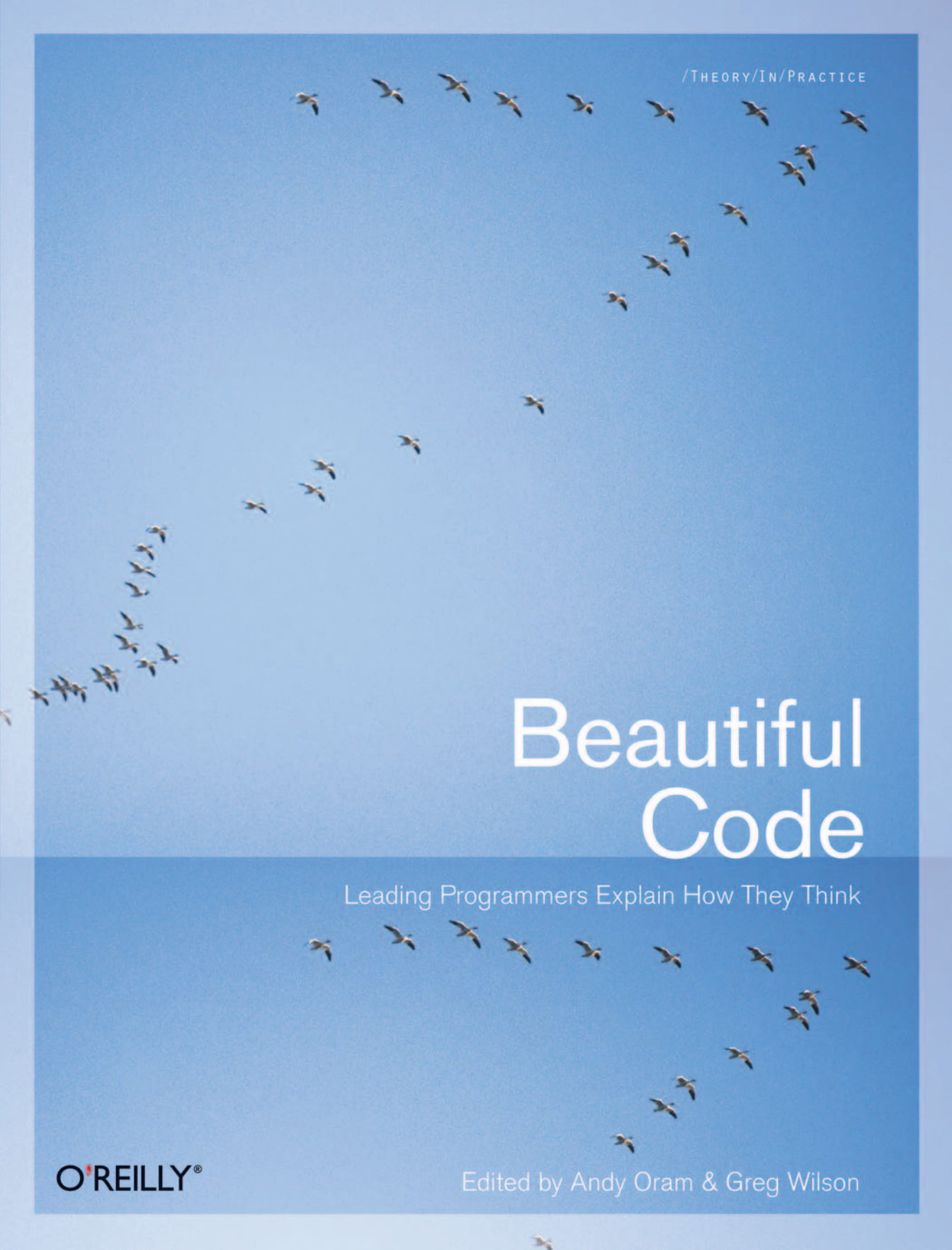
/Theory/In/Practice
Beautiful Code
Leading Programmers Explain How They Think
O'REILLY®
Edited by Andy Oram & Greg Wilson

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