The Craftsman

Robert C. Martin

New Edition, Revision 0.1.7, May 07, 2022

Foreword

> In the far-future year of ~~2364~~ 2002, generation-ships carry humanity to distant worlds.
>
> Somehow, we're still using Java.
>
> — anonymous reviewer

This is a book about the craft of software engineering, in story form.

This book is a community-edited omnibus collection of a series of short-fiction articles originally published serially in the (now-defunct) *ObjectMentor* and *InformIT* magazines, written by Robert C. Martin, a.k.a. "Uncle Bob." Some of the later articles come from Martin's own blog, *The Clean Coder*.

The story of *The Craftsman* is told as a series of Socratic dialogues between a new apprentice programmer named Alphonse, and the more-experienced programmers he works with. It's mostly just a framing device for talking about software, but there's also a little bit of *bildungsroman* to it.

This book is intended to be a companion to nonfiction titles like *Design Patterns* and Martin's own *Clean Code* series. Where non-fiction books can advocate for software-engineering practices and explain their benefits with retrospective object-lessons, the fictionalized treatment given here can give the reader a better sense of how it feels to apply — or fail to apply! — these processes in the moment; and can even build within readers a level of intuition for applying these skills.

Foreword

That being said — please, do not try to learn Java from this book.

Much of the source material for this book was written more than 15 years ago. When the earliest articles in the *Craftsman* series were being written, the current version of the Java language was *J2SE 1.4*. Many language features we take for granted today did not yet exist; and many practices we now avoid (e.g. direct spawning of `Thread`s) were still considered idiomatic.

The code in this book will likely eventually be modernized — or even translated into other languages with less boilerplate — to streamline the reading experience.

However, keep in mind that the code examples presented in each chapter *aren't the point of this book*. Your focus as a reader should not be on *what* is being written, but on *how* it is being written — how source code is being taken from one revision to the next. The *process of software engineering* demonstrated in this book needs no modernization; it is as applicable today as it was 15 years ago.

(Also, some of the code is bad on purpose. In this story, inexperienced developers will write some rather bad code. And, as well, inexperienced code-reviewers won't necessarily catch those devs' mistakes! Remember while reading, that a journeyman is not a master…)

Table of Contents

Chapter 1 – Opening Disaster

(Collects serial chapters *#1 Opening Disaster*, *#2 Crash Diet*, and *#3 Clarity and Collaboration*.)

13 February 2002

Dear Diary,

Today was a disaster — I really messed it up. I wanted so much to impress the journeymen here, but all I did was make a mess.

It was my first day on the job as an apprentice to Mr. C. I was lucky to get this apprenticeship. Mr. C is a well recognized master of software development. The competition for this position was fierce. Mr. C's apprentices often become journeymen in high demand. It *means* something to have worked with Mr. C.

I thought I was going to meet him today, but instead a journeyman named Jerry took me aside. He told me that Mr. C always puts his apprentices through an orientation during their first few days. He said this orientation was to introduce apprentices to the practices that Mr. C insists we use, and to the level of quality he expects from our code.

This excited me greatly. It was an opportunity to show them how good a programmer I am. So I told Jerry I couldn't wait to start. He responded by asking me to write a simple program for him. He wanted me to use the Sieve of Eratosthenes to calculate prime numbers. He told me to have the program, including all unit tests, ready for review just after lunch.

This was great! I had almost four hours to whip together a simple program like the Sieve. I was determined to do a really impressive job. Commit 1 shows what I wrote. I made sure it was well commented, and neatly formatted.

Commit 1, `GeneratePrimes.java`

```
/**
 * This class Generates prime numbers up to a user specified
 * maximum. The algorithm used is the Sieve of Eratosthenes.
 * <p>
 * Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --
 * d. c. 194, Alexandria. The first man to calculate the
 * circumference of the Earth. Also known for working on
 * calendars with leap years and ran the library at Alexandria.
 * <p>
 * The algorithm is quite simple. Given an array of integers
 * starting at 2. Cross out all multiples of 2. Find the next
 * uncrossed integer, and cross out all of its multiples.
 * Repeat untilyou have passed the square root of the maximum
 * value.
 *
 * @author Alphonse
 * @version 13 Feb 2002 atp
 */
```

```java
import java.util.*;

public class GeneratePrimes {
  /**
   * @param maxValue is the generation limit.
   */

  public static int[] generatePrimes(int maxValue) {
    if (maxValue >= 2) // the only valid case
    {
      // declarations
      int s = maxValue + 1; // size of array
      boolean[] f = new boolean[s];
      int i;

      // initialize array to true.
      for (i = 0; i < s; i++)
        f[i] = true;

      // get rid of known non-primes
      f[0] = f[1] = false;

      // sieve
      int j;
      for (i = 2; i < Math.sqrt(s) + 1; i++) {
        if (f[i]) // if i is uncrossed, cross its multiples.
        {
          for (j = 2 * i; j < s; j += i)
            f[j] = false; // multiple is not prime
        }
      }

      // how many primes are there?
      int count = 0;
      for (i = 0; i < s; i++)
      {
        if (f[i])
          count++; // bump count.
      }

      int[] primes = new int[count];

      // move the primes into the result
      for (i = 0, j = 0; i < s; i++)
      {
        if (f[i]) // if prime
          primes[j++] = i;
      }

      return primes; // return the primes
    }
```

```
    else // maxValue < 2
      return new int[0]; // return null array if bad input.
  }
}
```

Then I wrote a unit test for GeneratePrimes. It is shown in Commit 2. It uses the uses the JUnit framework as Jerry had instructed. It takes a statistical approach; checking to see if the generator can generate primes up to 0, 2, 3, and 100. In the first case there should be no primes. In the second there should be one prime, and it should be 2. In the third there should be two primes and they should be 2 and 3. In the last case there should be 25 primes the last of which is 97. If all these tests pass, then I assumed that the generator was working. I doubt this is foolproof, but I couldn't think of a reasonable scenario where these tests would pass and yet the function would fail.

Commit 2, TestGeneratePrimes.java

```
import junit.framework.*;
import java.util.*;

public class TestGeneratePrimes extends TestCase
{
  public static void main(String args[])
  {
    junit.swingui.TestRunner.main(
      new String[] {"TestGeneratePrimes"});
  }

  public TestGeneratePrimes(String name)
  {
    super(name);
  }

  public void testPrimes() {
    int[] nullArray = GeneratePrimes.generatePrimes(0);
    assertEquals(nullArray.length, 0);

    int[] minArray = GeneratePrimes.generatePrimes(2);
    assertEquals(minArray.length, 1);
    assertEquals(minArray[0], 2);

    int[] threeArray = GeneratePrimes.generatePrimes(3);
    assertEquals(threeArray.length, 2);
    assertEquals(threeArray[0], 2);
    assertEquals(threeArray[1], 3);

    int[] centArray = GeneratePrimes.generatePrimes(100);
    assertEquals(centArray.length, 25);
    assertEquals(centArray[24], 97);
  }
}
```

I got all this to work in about an hour. Jerry didn't want to see me until after lunch, so I spent the rest of my time reading the *Design Patterns* book that Jerry gave me.

After lunch I stopped by Jerry's office, and told him I was done with the program. He looked up at me with a funny grin on his face and said: "Good, let's go check it out."

He took me out into the lab and sat me down in front of a workstation. He sat next to me. He asked me to bring up my program on this machine. So I navigated to my laptop on the network and brought up the source files.

Jerry looked at the two source files for about five minutes. Then he shook his head and said: "You can't show something like this to Mr. C! If I let him see this, he'd probably fire both of us. He's not a very patient man." I was startled, but managed keep my cool enough to ask: "What's wrong with it?"

Jerry sighed. "Let's walk through this together." he said. "I'll show you, point by point, how Mr. C wants things done."

"It seems pretty clear," he continued, "that the main function wants to be three separate functions. The first initializes all the variables and sets up the sieve. The second actually executes the sieve, the third loads the sieved results into an integer array."

I could see what he meant. There *were* three concepts buried in that function. Still, I didn't see what he wanted me to do about it.

He looked at me for awhile, clearly expecting me to do something. But finally he heaved a sigh, shook his head, and continued. "To expose these three concepts more clearly, I want you to extract them into three separate methods. Also get rid of all the unnecessary comments and pick a better name for the class. When you are done with that, make sure all the tests still run."

You can see what I did in Commit 3. (I've highlighted the lines with changes, just like Martin Fowler does in his *Refactoring* book.) I changed the name of the class to a noun, got rid of all the comments about Eratosthenes, and made three methods out of the three concepts in the `generatePrimes` function.

Extracting the three functions forced me to promote some of the variables of the function to `static` fields of the class. Jerry said that this made it much clearer which variables are local and which have wider influence.

Commit 3, `PrimeGenerator.java`

```
/**
 * This class Generates prime numbers up to a user specified
 * maximum. The algorithm used is the Sieve of Eratosthenes.
 * Given an array of integers starting at 2:
 * Find the first uncrossed integer, and cross out all its
 * multiples. Repeat until the first uncrossed integer exceeds
 * the square root of the maximum value.
 */

import java.util.*;
```

```
public class PrimeGenerator
{
  private static int s;
  private static boolean[] f;
  private static int[] primes;

  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else {
      initializeSieve(maxValue);
      sieve();
      loadPrimes();
      return primes; // return the primes
    }
  }

  private static void loadPrimes()
  {
    int i;
    int j;

    // how many primes are there?
    int count = 0;
    for (i = 0; i < s; i++)
    {
      if (f[i])
        count++; // bump count.
    }

    primes = new int[count];

    // move the primes into the result
    for (i = 0; j = 0; i < s; i++)
    {
      if (f[i]) // if prime
        primes[j++] = i;
    }
  }

  private static void sieve()
  {
    int i;
    int j;
    for (i = 2; i < Math.sqrt(s) + 1; i++)
    {
      if (f[i]) // if i is uncrossed, cross out its multiples.
      {
        for (j = 2 * i; j < s; j += i)
```

```
          f[j] = false; // multiple is not prime
        }
      }
    }

    private static void initializeSieve(int maxValue)
    {
      // declarations
      s = maxValue + 1; // size of array
      f = new boolean[s];
      int i;

      // initialize array to true.
      for (i = 0; i < s; i++)
        f[i] = true;

      // get rid of known non-primes
      f[0] = f[1] = false;
    }
}
```

Jerry told me that this was a little messy, so he took the keyboard and showed me how to clean it up. Commit 4 shows what he did. First he got rid of the `s` variable in `initializeSieve` and replacing it with `f.length`. Then he changed the names of the three functions to something he said was a bit more expressive. Finally he rearranged the innards of `initializeArrayOfIntegers` (née `initializeSieve`) to be a little nicer to read. The tests all still ran.

Commit 4, `PrimeGenerator.java` *(partial)*

```
public class PrimeGenerator
{
  private static boolean[] f;
  private static int[] result;

  // ...

  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      initializeArrayOfIntegers(maxValue);
      crossOutMultiples();
      putUncrossedIntegersIntoResult();
      return result;
    }
  }

  private static void initializeArrayOfIntegers(int maxValue)
  {
    f = new boolean[maxValue + 1];
    f[0] = f[1] = false; //neither primes nor multiples.
    for (int i = 2; i < f.length; i++)
      f[i] = true;
  }

  // ...
}
```

I had to admit, this was a bit cleaner. I'd always thought it was a waste of time to give functions long descriptive names, but his changes really did make the code more readable.

Next Jerry pointed at `crossOutMultiples`. He said he thought the `if(f[i] == true)` statements could be made more readable. I thought about it for a minute. The intent of those statements was to check to see if `i` was uncrossed; so I changed the name of `f` to `unCrossed`.

Jerry said that this was better, but still wasn't pleased with it because it lead to double negatives like `unCrossed[i] = false`. So he changed the name of the array to `isCrossed` and changed the sense of all the booleans. Then he ran all the tests.

Jerry got rid of the initialization that set `isCrossed[0]` and `isCrossed[1]` to `true`. He said it was good enough to just made sure that no part of the function used the `isCrossed` array for indexes less than 2. The tests all still ran.

Jerry extracted the inner loop of the `crossOutMultiples` function and called it `crossOutMultiplesOf`. He said that statements like if `(isCrossed[i] == false)` were confusing so he created a function

called `notCrossed` and changed the if statement to `if (notCrossed(i))`. Then he ran the tests.

Then Jerry asked me what that square root was all about. I spent a bit of time writing a comment that tried to explain why you only have to iterate up to the square root of the array size. I tried to emulate Jerry by extracting the calculation into a function where I could put the explanatory comment. In writing the comment I realized that the square root is the maximum prime factor of any of the integers in the array. So I chose that name for the variables and functions that dealt with it. Finally, I made sure that the tests all still ran. The result of all these changes are shown in Commit 5.

Commit 5, `PrimeGenerator.java` *(partial)*

```
public class PrimeGenerator
{
  private static boolean[] isCrossed;
  private static int[] result;

  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      initializeArrayOfIntegers(maxValue);
      crossOutMultiples();
      putUncrossedIntegersIntoResult();
      return result;
    }
  }

  private static void initializeArrayOfIntegers(int maxValue)
  {
    isCrossed = new boolean[maxValue + 1];
    for (int i = 2; i < isCrossed.length; i++)
      isCrossed[i] = false;
  }

  private static void crossOutMultiples()
  {
    int maxPrimeFactor = calcMaxPrimeFactor();
    for (int i = 2; i <= maxPrimeFactor; i++)
      if (notCrossed(i))
        crossOutMultiplesOf(i);
  }

  private static int calcMaxPrimeFactor()
  {
    // We cross out all multiples of p, where p is prime.
    // Thus, all crossed out multiples have p and q for
    // factors. If p > sqrt of the size of the array, then
    // q will never be greater than 1. Thus p is the
```

```
    // largest prime factor in the array, and is also
    // the iteration limit.
    double maxPrimeFactor = Math.sqrt(isCrossed.length) + 1;
    return (int) maxPrimeFactor;
  }

  private static void crossOutMultiplesOf(int i)
  {
    for (int multiple = 2*i;
         multiple < isCrossed.length;
         multiple += i)
      isCrossed[multiple] = true;
  }

  private static boolean notCrossed(int i)
  {
    return isCrossed[i] == false;
  }
}
```

I was starting to get the hang of this so I took a look at the `putUncrossedIntegersIntoResult` method. I saw that this method had two parts. The first counts the number of uncrossed integers in the array, and creates the result array of that size. The second moves the uncrossed integers into the result array. So, as you can see in Commit 6, I extracted the first part into its own function and did some miscellaneous cleanup. The tests all still ran. Jerry was just barely nodding his head. Did he actually like what I did?

Commit 6, `PrimeGenerator.java` *(partial)*

```
private static void putUncrossedIntegersIntoResult()
{
  result = new int[numberOfUncrossedIntegers()];
  for (int j = 0, i = 2; i < isCrossed.length; i++)
    if (notCrossed(i))
      result[j++] = i;
}

private static int numberOfUncrossedIntegers()
{
  int count = 0;
  for (int i = 2; i < isCrossed.length; i++)
    if (notCrossed(i))
      count++;

  return count;
}
```

Next Jerry made pass over the whole program, reading it from beginning to end, rather like he was reading a geometric proof. He told me that this was a real important step. “So far”, he said, “We’ve been refactoring fragments. Now we want to see if the whole program hangs together as a readable

whole."

I asked: "Jerry, do you do this with your own code too?"

Jerry scowled: "We work as a team around here, so there is no code I call my own. Do you consider this code yours now?"

"Not anymore." I said, meekly. "You've had a big influence on it."

"We *both* have." he said, "And that's the way that Mr. C likes it. He doesn't want any single person owning code. But to answer your question: Yes. We all practice this kind of refactoring and code clean up around here. It's Mr. C's way."

During the read-through, Jerry realized that he didn't like the name `initializeArrayOfIntegers`.

"What's being initialized is not, in fact, an array of integers;", he said, "it's an array of booleans. But `initializeArrayOfBooleans` is not an improvement. What we are really doing in this method is uncrossing all the relevant integers so that we can then cross out the multiples."

"Of course!" I said. So I grabbed the keyboard and changed the name to `uncrossIntegersUpTo`. I also realized that I didn't like the name `isCrossed` for the array of booleans. So I changed it to `crossedOut`. The tests all still run. I was starting to enjoy this; but Jerry showed no sign of approval.

Then Jerry turned to me and asked me what I was smoking when I wrote all that `maxPrimeFactor` stuff. (See Commit 6.) At first I was taken aback. But as I looked the code and comments over I realized he had a point. Yikes, I felt stupid! The square root of the size of the array is not necessarily prime. That method did not calculate the maximum prime factor. The explanatory comment was just wrong. So I sheepishly rewrote the comment to better explain the rationale behind the square root, and renamed all the variables appropriately. The tests all still ran.

Commit 7, `PrimeGenerator.java` *(partial)*

```
private static int calcMaxPrimeFactor() {
  // We cross out all multiples of p, where p is prime.
  // Thus, all crossed out multiples have p and q for
  // factors. If p > sqrt of the size of the array, then
  // q will never be greater than 1. Thus p is the
  // largest prime factor in the array, and is also
  // the iteration limit.
  double maxPrimeFactor = Math.sqrt(isCrossed.length) + 1;
  return (int) maxPrimeFactor;
}
```

"What the devil is that `+1` doing in there?" Jerry barked at me.

I gulped, looked at the code, and finally said: "I was afraid that a fractional square root would convert to an integer that was one too small to serve as the iteration limit."

"So you're going to litter the code with extra increments just because you are paranoid?" he asked. "That's silly, get rid of the increment and run the tests."

I did, and the tests all ran. I thought about this for a minute, because it made me nervous. But I decided that maybe the true iteration limit was the largest prime less than or equal to the square root of the size of the array.

“That last change makes me pretty nervous.” I said to Jerry. “I understand the rationale behind the square root, but I’ve got a nagging feeling that there may be some corner cases that aren’t being covered.”

“OK,” he grumbled. “So write another test that checks that.”

“I suppose I could check that there are no multiples in any of the prime lists between 2 and 500.”

“OK, if it’ll make you feel better, try that.” he said. He was clearly becomming impatient.

So I wrote the `testExhaustive` function shown in Commit 9. The new test passed, and my fears were allayed.

Then Jerry relented a bit. “It’s always good do know why something works;” said Jerry, “and it’s even better when you show you are right with a test.”

Then Jerry scrolled one more time through all the code and tests (shown in Commit 8). He sat back, thinking for a minute, and said: “OK, I think we’re done. The code looks reasonably clean. I’ll show it to Mr. C.”

Then he looked me dead in the eye and said: “Remember this. From now on when you write a module, get help with it and keep it clean. If you hand in anything below those standards you won’t last long here.”

And with that, he strode off.

Commit 8, `PrimeGenerator.java`

```
/**
 * This class Generates prime numbers up to a user specified
 * maximum. The algorithm used is the Sieve of Eratosthenes.
 * Given an array of integers starting at 2:
 * Find the first uncrossed integer, and cross out all its
 * multiples. Repeat until there are no more multiples
 * in the array.
 */

public class PrimeGenerator
{
  private static boolean[] crossedOut;
  private static int[] result;

  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
```

```
  {
    uncrossIntegersUpTo(maxValue);
    crossOutMultiples();
    putUncrossedIntegersIntoResult();
    return result;
  }
}

private static void uncrossIntegersUpTo(int maxValue)
{
  crossedOut = new boolean[maxValue + 1];
  for (int i = 2; i < crossedOut.length; i++)
    crossedOut[i] = false;
}

private static void crossOutMultiples()
{
  int limit = determineIterationLimit();
  for (int i = 2; i <= limit; i++)
    if (notCrossed(i))
      crossOutMultiplesOf(i);
}

private static int determineIterationLimit()
{
  // Every multiple in the array has a prime factor that
  // is less than or equal to the sqrt of the array size,
  // so we don't have to cross out multiples of numbers
  // larger than that root.
  double iterationLimit = Math.sqrt(crossedOut.length);
  return (int) iterationLimit;
}

private static void crossOutMultiplesOf(int i)
{
  for (int multiple = 2*i;
       multiple < crossedOut.length;
       multiple += i)
    crossedOut[multiple] = true;
}

private static boolean notCrossed(int i)
{
  return crossedOut[i] == false;
}

private static void putUncrossedIntegersIntoResult()
{
  result = new int[numberOfUncrossedIntegers()];
  for (int j = 0, i = 2; i < crossedOut.length; i++)
    if (notCrossed(i))
```

```
        result[j++] = i;
    }

    private static int numberOfUncrossedIntegers()
    {
      int count = 0;
      for (int i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
          count++;

      return count;
    }
}
```

Commit 8, TestGeneratePrimes.java

```
import junit.framework.*;

public class TestGeneratePrimes extends TestCase
{
  public static void main(String args[])
  {
    junit.swingui.TestRunner.main(
      new String[] {"TestGeneratePrimes"});
  }

  public TestGeneratePrimes(String name)
  {
    super(name);
  }

  public void testPrimes()
  {
    int[] nullArray = PrimeGenerator.generatePrimes(0);
    assertEquals(nullArray.length, 0);

    int[] minArray = PrimeGenerator.generatePrimes(2);
    assertEquals(minArray.length, 1);
    assertEquals(minArray[0], 2);

    int[] threeArray = PrimeGenerator.generatePrimes(3);
    assertEquals(threeArray.length, 2);
    assertEquals(threeArray[0], 2);
    assertEquals(threeArray[1], 3);

    int[] centArray = PrimeGenerator.generatePrimes(100);
    assertEquals(centArray.length, 25);
    assertEquals(centArray[24], 97);
  }

  public void testExhaustive()
```

```
  {
    for (int i = 2; i<500; i++)
      verifyPrimeList(PrimeGenerator.generatePrimes(i));
  }

  private void verifyPrimeList(int[] list)
  {
    for (int i=0; i<list.length; i++)
      verifyPrime(list[i]);
  }

  private void verifyPrime(int n)
  {
    for (int factor=2; factor<n; factor++)
      assert(n%factor != 0);
  }
}
```

What a disasater! I thought sure that my original solution had been top-notch. In some ways I still feel that way. I had tried to show off my brilliance, but I guess Mr. C values collaboration and clarity more than individual brilliance.

I had to admit that the program reads much better than it did at the start. It also works a bit better. I was pretty pleased with the outcome. Also, in spite of Jerry's gruff attitude, it had been *fun* working with him. I had learned a lot.

Still, I was pretty discouraged with my performance. I don't think the folks here are going to like me very much. I'm not sure they'll ever think I'm good enough for them. This is going to be a lot harder than I thought.

Chapter 2 – A Test of Patience

(Collects serial chapters *#4 A Test of Patience* and *#5 Baby Steps*.)

Dear Diary,

Last night I stared out the window for hours watching the stars drift through the spectrum. I felt conflicted about the work I did with Jerry yesterday. I learned a lot from working with Jerry on the prime generator, but I don't think I impressed him very much. And, frankly, I wasn't all that impressed with him. he spent a lot of time polishing a piece of code that worked just fine.

Today Jerry came to me with a new exercise. He asked me to write a program that calculates the prime factors of an integer. He said he'd work with me from the start. So the two of us sat down and began to program.

I was pretty sure I knew how to do this. We had written the prime generator yesterday. Finding prime factors is just a matter of walking through a list of primes and seeing if any are factors of the given integer. So I grabbed the keyboard and began to write code. After about half an hour of writing and testing I had produced the following.

Commit 1, `PrimeFactorizer.java`

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class PrimeFactorizer {
  public static void main(String[] args) {
    int[] factors = findFactors(Integer.parseInt(args[0]));
    for (int i = 0; i < factors.length; i++)
      System.out.println(factors[i]);
  }

  public static int[] findFactors(int multiple) {
    List factors = new LinkedList();
    int[] primes = PrimeGenerator.generatePrimes((int) Math.sqrt(multiple));
    for (int i = 0; i < primes.length; i++)
      for (; multiple % primes[i] == 0; multiple /= primes[i])
        factors.add(new Integer(primes[i]));
    return createFactorArray(factors);
  }

  private static int[] createFactorArray(List factors) {
    int factorArray[] = new int[factors.size()];
    int j = 0;
    for (Iterator fi = factors.iterator(); fi.hasNext();) {
      Integer factor = (Integer) fi.next();
      factorArray[j++] = factor.intValue();
    }
    return factorArray;
  }
}
```

I tested the program by running the main program with several different arguments. They all seemed to work. Running it with 100 gave me 2, 2, 5, and 5. Running it with 32767 gave me 7, 31, and 151. Running it with 32768 gave me fifteen twos.

Jerry just sat there and watched me. He didn't say a word. This made me nervous, but I kept on massaging and testing the code until I was happy with it. Then I started writing the unit tests.

"What are you doing?" asked Jerry.

"The program works, so I'm writing the unit tests." I replied.

"Why do you need unit tests if the program already works?" he said?

I hadn't thought of it that way. I just knew that you were supposed to write unit tests. I ventured a guess: "So that other programmers can see that it works?"

Jerry looked at me for about thirty seconds. Then he shook his head and said: "What are they teaching you guys in school nowadays?"

I started to answer, but he stopped me with a look.

“OK”, he said, “delete what you’ve done. I’ll show you how we do things around here.”

I wasn’t prepared for this. He wanted me to delete what I had just spent thirty minutes creating. I just sat there in disbelief.

Finally, Jerry said: “Go ahead, delete it.”

“But it works.” I said.

“So what?” Said Jerry.

I was starting to get testy. “There’s nothing wrong with it!” I asserted. “Really.” he grumbled; and he grabbed the keyboard and deleted my code.

I was dumbfounded. No, I was furious. He had just reached over and deleted my work. For a minute I stopped caring about the prestige of being an apprentice of Mr. C. What good was that apprenticeship if it meant I had to work with brutes like Jerry? These, and other less complimentary, thoughts raced behind my eyes as I glared at him.

“Ah. I see that upset you.” Jerry said calmly.

I sputtered, but couldn’t say anything intelligent.

“Look.” Jerry said, clearly trying to calm me down. “Don’t become vested in your code. This was just thirty minutes worth of work. It’s not that big a deal. You need to be ready to throw away a lot more code than that if you want to become any kind of a programmer. Often the best thing you can do with a batch of code is throw it out.”

“But that’s such a waste!” I blurted.

“Do you think the value of a program is in the code?” he asked. “It’s not. The value of a program is in your head.”

He looked at me for a second, and then went on. “Have you ever accidentally deleted something you were working on? Something that took a few days of effort?”

“Once, at school.” I said. “A disk crashed and the latest backup was two days old.

He winced and nodded knowingly. Then he asked: “How long did it take you to recreate what you had lost?”

“I was pretty familiar with it, so it only took me about half a day to recreate it.”

“So you didn’t really lose two days worth of work.”

I didn’t care for his logic. I couldn’t refute it, but I didn’t like it. It had *felt* like I had lost two days worth of work!

“Did you notice whether the new code was better or worse than the code you lost?” he asked.

“Oh, it was much better.” I said, regretting my words the instant I said them. “I was able to use a much better structure the second time.”

He smiled. “So for an extra 25% effort, you wound up with a better solution.”

His logic was annoying me. I shook my head and nearly shouted: “Are you suggesting that we always throw away our code when we are done?”

To my astonishment he nodded his head and said: “Almost. I’m suggesting that throwing away code is a valid and useful operation. I’m suggesting that you should not view it as a loss. I’m suggesting that you not get vested in your code.”

I didn’t like this; but I didn’t have an argument to use against him. I just sat there in silent disagreement. “OK”, he said, “Let’s start over. The way we work around here is to write our unit tests *first*.”

This was patently absurd. I reacted with an intelligent: “Huh?”

“Let me show you.” he said. “Our task is to create an array of prime factors from a single integer. What is the simplest test case you can think of?”

“The first valid case is 2. And it should return an array with just a single 2 in it.”

“Right.” he said. And he wrote the following unit test.

Commit 2, `TestPrimeFactors.java` *(partial)*

```
public void testTwo() throws Exception {
  int factors[] = PrimeFactorizer.factor(2);
  assertEquals(1, factors.length);
  assertEquals(2, factors[0]);
}
```

Then he wrote the simplest code that would allow the test case to compile.

Commit 3, `PrimeFactorizer.java`

```
public class PrimeFactorizer {
  public static int[] factor(int multiple) {
    return new int[0]; }
  }
}
```

He ran the test, and it failed saying: `testTwo(TestPrimeFactors): expected: <1> but was: <0>`.

Now he looked at me and he said: “Do the simplest thing possible to make that test case pass.”

This was absurdity upon absurdity. “What do you mean?” I said. “The simplest thing would be to return an array with a 2 in it.”

With a straight face, he said, “OK, do that.”

“But that’s silly.” I said. “It’s the wrong code. The real solution isn’t going to just return a 2.”

“Yes, that’s true.” he said, “But just humor me for a bit.”

I sighed, rolled my eyes, huffed and puffed a bit, and then wrote:

Commit 4, `PrimeFactorizer.java` *(partial)*

```
public static int[] factor(int multiple) {
  return new int[] {2};
}
```

I ran the tests, and — of course — they passed.

“What did that prove?” I asked.

“It proved that you could write a function that finds the prime factors of two.” He said. “It also proves that the test passes when the function responds correctly to a two.”

I rolled my eyes again. This was beneath my intelligence. I thought being an apprentice here was supposed to *teach* me something.

“Now, what’s the simplest test case we can add to this?” he asked me.

I couldn’t help myself. I dripped with sarcasm as I said: “Gosh, Jerry, maybe we should try a three.”

And though I expected it, I was also incredulous. He actually wrote the test case for three:

Commit 5, `TestPrimeFactors.java` *(partial)*

```
public void testThree() throws Exception {
  int factors[] = PrimeFactorizer.factor(3);
  assertEquals(1, factors.length);
  assertEquals(3, factors[0]);
}
```

Running it produced the expected failure:

```
testThree(TestPrimeFactors): expected: <3> but was: <2>
```

“OK, Alphonse, do the simplest thing that will make this test case pass.”

Impatiently, I took the keyboard and typed the following:

Commit 6, `PrimeFactorizer.java` *(partial)*

```
public static int[] factor(int multiple) {
  if (multiple == 2)
    return new int[] {2};
  else
    return new int[] {3};
}
```

I ran the tests, and they passed.

Jerry looked at me with an odd kind of smile. He said: “OK, that passes the tests. However, it’s not very bright, is it?”

He’s the one who started this nonsense and now he’s asking *me* if this is *bright*? “I think this whole exercise is pretty dim,” I said.

He ignored me and continued. “Every time you add a new test case, you have to make it pass by making the code more general. Now go back and make the simplest change that is more general than your first solution.”

I thought about this for a minute. At last Jerry had asked me something that might require a few brain cells. Yes, there was a more general solution. I took the keyboard and typed:

Commit 7, `PrimeFactorizer.java` *(partial)*

```
public static int[] factor(int multiple) {
  return new int[] {multiple};
}
```

The tests passed, and Jerry smiled. But I still couldn’t see how this was getting us any closer to generating prime factors. As far as I could tell, this was a ridiculous waste of time. Still, I wasn’t surprised when Jerry asked me: “Now what’s the simplest test case we can add?”

“Clearly that would be the case for four.” I said impatiently. And I grabbed the keyboard and wrote:

Commit 8, `TestPrimeFactors.java` *(partial)*

```
public void testFour() throws Exception {
  int factors[] = PrimeFactorizer.factor(4);
  assertEquals(2, factors.length);
  assertEquals(2, factors[0]);
  assertEquals(2, factors[1]);
}
```

“I expect the first assert will fail because an array of size 1 will be returned.” I said.

Sure enough, when I ran the test it reported:

```
testFour(TestPrimeFactors): expected <2> but was <1>
```

“I presume you’d like me to make the simplest modification that will make all these tests pass, and will make the factor method more general?” I asked.

Jerry just nodded.

I made a concerted effort to solve only the test case at hand, ignoring the test cases I knew would be next. This galled me, but it was what Jerry wanted. The result was:

Commit 9, PrimeFactorizer.java

```
public class PrimeFactorizer {
  public static int[] factor(int multiple) {
    int currentFactor = 0;
    int factorRegister[] = new int[2];

    for (; (multiple % 2) == 0; multiple /= 2)
      factorRegister[currentFactor++] = 2;

    if (multiple != 1)
      factorRegister[currentFactor++] = multiple;

    int factors[] = new int[currentFactor];

    for (int i = 0; i < currentFactor; i++)
      factors[i] = factorRegister[i];

    return factors;
  }
}
```

This passed all the tests, but was pretty messy. Jerry scrunched up his face as though he smelled something rotten. He said: “We have to refactor this before we go any further.”

“Wait.” I objected. “I agree that it’s a bit messy. But shouldn’t we get it all working first and then refactor it if there’s time?”

“Egad! No!” said Jerry. “We need to refactor it *now* so that we can see the true structure as it evolves. Otherwise we’ll just keep piling mess upon mess, and we’ll lose the sense of what we’re doing.”

“OK.” I sighed. “Let’s clean this up.”

So the two of us did a little refactoring. The result follows:

Commit 10, PrimeFactorizer.java

```
public class PrimeFactorizer {
  private static int factorIndex;
  private static int[] factorRegister;

  public static int[] factor(int multiple) {
    initialize();
    findPrimeFactors(multiple);
    return copyToResult();
  }

  private static void initialize() {
    factorIndex = 0;
    factorRegister = new int[2];
  }

  private static void findPrimeFactors(int multiple) {
    for (; (multiple % 2) == 0; multiple /= 2)
      factorRegister[factorIndex++] = 2;
    if (multiple != 1)
      factorRegister[factorIndex++] = multiple;
  }

  private static int[] copyToResult() {
    int factors[] = new int[factorIndex];
    for (int i = 0; i < factorIndex; i++)
      factors[i] = factorRegister[i];
    return factors;
  }
}
```

“Time for the next test case.” Said Jerry; and he passed me the keyboard.

I still couldn’t see where this was going, but there was no way out of it. following test case:

Commit 11, TestPrimeFactors.java *(partial)*

```
public void testFive() throws Exception {
  int factors[] = PrimeFactorizer.factor(5);
  assertEquals(1, factors.length);
  assertEquals(5, factors[0]);
}
```

“That’s interesting.”, I said as I stared at the green bar, “That one works without change.”

“That *is* interesting.” Said Jerry. “Let’s try the next test case.”

Now I was intrigued. I hadn’t expected the test case to just work. As I thought about it, it was obvious why it worked, but I still hadn’t anticipated it. I was pretty sure the next test case would

fail, so I typed it in and ran it.

Commit 12, `TestPrimeFactors.java` *(partial)*

```
public void testSix() throws Exception {
  int factors[] = PrimeFactorizer.factor(6);
  assertEquals(2, factors.length);
  assertContains(factors, 2);
  assertContains(factors, 3);
}

private void assertContains(int factors[], int n) {
  String error = "assertContains:" + n;
  for (int i = 0; i < factors.length; i++) {
    if (factors[i] == n) return;
  }
  fail(error);
}
```

“Yikes! That one passed too!” I cried.

“Interesting.” Nodded Jerry. “Seven is going to work too, isn’t it?”

“Yeah, I think it is.”

“Then let’s skip it and go for eight. That one can’t pass!”

He was right. Eight had to fail because the `factorRegister` array was too small.

Commit 13, `TestPrimeFactors.java` *(partial)*

```
public void testEight() throws Exception {
  int factors[] = PrimeFactorizer.factor(8);
  assertEquals(3, factors.length);
  assertContainsMany(factors, 3, 2);
}

private void assertContainsMany(int factors[], int n, int f) {
  String error = "assertContains(" + n + "," + f + ")";
  int count = 0;

  for (int i = 0; i < factors.length; i++) {
    if (factors[i] == f) count++;
  }

  if (count != n)
    fail(error);
}
```

“What a relief! It failed!”

“Yeah,” said Jerry, “for an array out of bounds exception. You could get it to pass by increasing the size of `factorRegister`, but that wouldn’t be more general.”

“Let’s try it anyway, and then we’ll solve the general problem of the array size.”

So I changed the 2 to a 3 in the `initialize` function, and got a green bar.

“OK,” I said, “what is the maximum number of factors that a number can have?”

“I think it’s something like the log2 of the number.” said Jerry.

“Wait!” I said, “Maybe we’re chasing our tail. What is the largest number we can handle? Isn’t it 2^{64}?”

“I’m pretty sure it can’t be larger than that.” said Jerry.

“OK, then let’s just make the size of the `factorRegister` 100. That’s big enough to handle any number we throw at it.

“Fine by me.” said Jerry. “A hundred integers is nothing to worry about.”

We tried it, and the tests still ran.

I looked at Jerry and said: “The next test case is nine. That’s certainly going to fail.”

“Let’s try it.” he said.

So I typed in the following:

Commit 14, `TestPrimeFactors.java` *(partial)*

```
public void testNine() throws Exception {
  int factors[] = PrimeFactorizer.factor(9);
  assertEquals(2, factors.length);
  assertContainsMany(factors, 2, 3);
}
```

“Good, that failed.” I said. “Making it pass should be simple. I just need to remove 2 as a special number in `findPrimeFactors`, and use both 2 and 3 with some general algorithm.” So I modified `findPrimeFactors` as follows:

Commit 15, `PrimeFactorizer.java` *(partial)*

```
private static void findPrimeFactors(int multiple) {
  for (int factor = 2; multiple != 1; factor++)
    for (; (multiple % factor) == 0; multiple /= factor)
      factorRegister[factorIndex++] = factor;
}
```

“OK, that passes.” Said Jerry. “Now what’s the next failing test case?”

“Well, the simple algorithm I used will divide by non-primes as well as primes. That won’t work right. That’s why my first version of the program divided *only* by primes. The first non-prime the algorithm will divide by is four, so I imagine 4×4 will fail.”

Commit 16, `TestPrimeFactors.java` *(partial)*

```
public void testSixteen() throws Exception {
  int factors[] = PrimeFactorizer.factor(16);
  assertEquals(4, factors.length);
  assertContainsMany(factors, 4, 2);
}
```

“Ouch! That passes.” I said. “How could that pass?”

“It passes, because all the twos have been removed before you try to divide by four, so four is never found as a factor. Remember, it also wasn’t found as a factor or 8 or 4!”

“Of course!” I said. “All the primes are removed before their composites. The fact that the algorithm checks the composites is irrelevant. But that means I never needed the array of prime numbers that I had in my first version.”

“Right.” said Jerry. “That’s why I deleted it.”

“Is this it then? Are we done?”

“Can you think of a failing test case?” asked Jerry?

“I don’t know.” I said. “Let’s try 1000.”

“Ah, the shotgun approach. OK, give it a try.”

Commit 17, `TestPrimeFactors.java` *(partial)*

```
public void testThousand() throws Exception {
  int factors[] = PrimeFactorizer.factor(1000);
  assertEquals(6, factors.length);
  assertContainsMany(factors, 3, 2);
  assertContainsMany(factors, 3, 5);
}
```

“That worked! OK, how about…”

We tried several other test cases, but they all passed. This version of the program was much simpler than my first version, and was faster too. No wonder Jerry deleted the first one.

What amazed me, and still amazes me, is that we snuck up on the better solution one test case at a time. I don’t think I would ever have stumbled upon this simple approach had we not been inching forward one simple test case at a time. I wonder if that happens in bigger projects?

I *learned* something today.

Chapter 3 – Once is Not Enough

(Collects serial chapters *#6–7 Once Is Not Enough* and *#8 Test Document*.)

Yesterday left me drained. Jerry and I had solved the prime factors problem by sneaking up on it one tiny test case at a time. It was the strangest way to solve a problem that I have ever seen; but it worked better than my original solution.

I wandered aimlessly around the corridors thinking about it again and again. I don't remember dinner, or even being in the galley. I fell asleep early and dreamed of sequences of tiny test cases.

When I reported to Jerry this morning he said:

"Good Morning Alphonse. Are you ready to start working on a *real* project?"

"You bet I am!" Farb, yes, I was ready! I was tired of these play examples.

"Good! We've got a program called SMC that compiles a finite state machine grammar into Java. Mr. C wants us to turn that program into a service delivered over the net."

"What do you mean?" I asked.

Jerry turned to a sketch-wall and began to talk and draw at the same time.

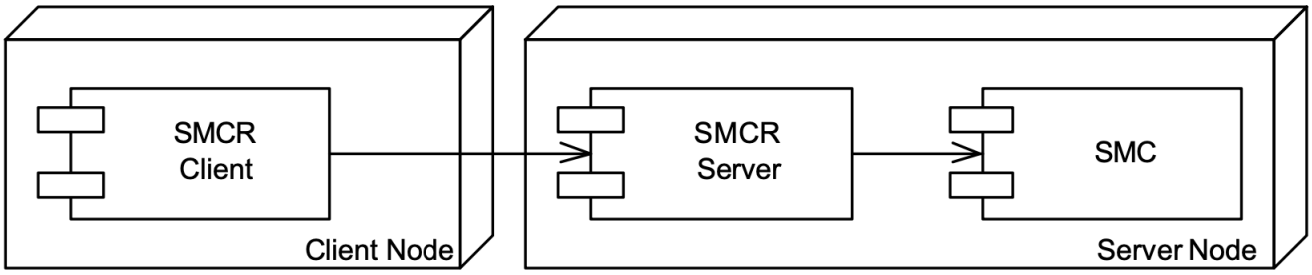

Figure 1. SMCR service diagram

"We're going to write two programs. One called SMCR Client, and the other called SMCR Server. The user who wants to compile a finite state machine will invoke SMCR Client with the name of the file to be compiled. SMCR Client will send that file to a special computer where SMCR Server is running. SMCR Server will run the SMC compiler and then send the resulting compiled files back to SMCR Client. SMCR Client will then write them in the user's directory. As far as the user is concerned, it'll be no different than using SMC directly."

"OK, I think I understand." I said. "It sounds pretty simple."

"It is *pretty* simple." Jerry said. "But working with sockets is always just a little interesting."

So we sat down at a workstation and, as usual, got ready to write our first unit test. Jerry thought for a minute and then went back to the sketch-wall and drew the following diagram.

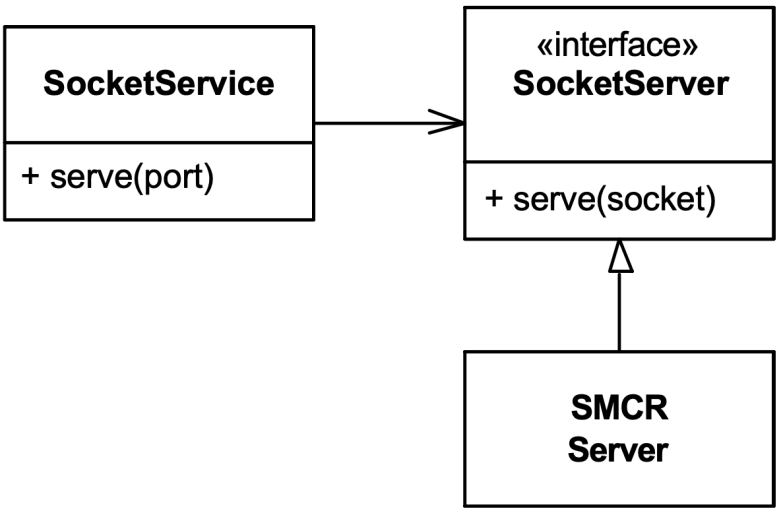

Figure 2. SMCR Server type hierarchy

"Here's what I have in mind for SMCR Server." He said. "We'll put the socket management code in a class called `SocketService`. This class will catch, and manage, connections coming from the outside. When `serve(port)` is called, it'll create the service socket with the given port number and start accepting connections. Whenever a connection comes in it will create a new thread and pass control to the `serve(socket)` method of the `SocketServer` interface. That way we separate socket management code from the code that performs the services we desire."

Not knowing whether this was a good thing or not, I simply nodded. Clearly he had some reason for thinking the way he did. I just went along with it.

Next he wrote the following test.

Commit 1, `TestSocketServer.java` *(partial)*

```
public void testOneConnection() throws Exception {
  SocketService ss = new SocketService();
  ss.serve(999);
  connect(999);
  ss.close();
  assertEquals(1, ss.connections());
}
```

"What I'm doing here is called *Intentional Programming*." Jerry said. I'm calling code that doesn't yet exist. As I do so, I express my intent about what that code should look like, and how it should behave."

"OK." I responded. "You create the `SocketService`. Then you ask it to accept connections on port 999. Next it looks like you are connecting to the service you just created on port 999. Finally you close the `SocketService` and assert that it got one connection."

"Right." Jerry confirmed.

"But how do you know `SocketService` will need the connections method?"

"Oh, it probably doesn't. I'm just putting it there so I can test it."

“Isn’t that wasteful?” I queried.

Jerry looked me sternly in the eye and replied: “Nothing that makes a test easy is wasted, Alphonse. We often add methods to classes simply to make the classes easier to test.”

I didn’t like the `connections()` method, but I held my tongue for the moment.

We wrote just enough of the `SocketService` constructor, and the `serve`, `close`, and `connect` methods to get everything to compile. They were just empty functions. When we ran the test, it failed as expected.

Next Jerry wrote the `connect` method as part of the test case class.

Commit 2, `TestSocketServer.java` *(partial)*

```
private void connect(int port) {
  try {
    Socket s = new Socket("localhost", port);
    s.close();
  } catch (IOException e) {
    fail("could not connect");
  }
}
```

Running this produced the following error:

```
testOneConnection: could not connect
```

I said: “It’s failing because it can’t find port 999 anywhere, right?”

“Right!” said Jerry. “But that’s easy to fix. Here, why don’t you fix it?”

I’d never written socket code before, so I wasn’t sure what to do next. Jerry pointed me to the `ServerSocket` entry in the Javadocs. The examples there made it look pretty simple. So I fleshed in the `SocketService` methods as shown below.

Commit 3, `SocketService.java`

```
import java.net.ServerSocket;

public class SocketService {
  private ServerSocket serverSocket = null;

  public void serve(int port) throws Exception {
    serverSocket = new ServerSocket(port);
  }

  public void close() throws Exception {
    serverSocket.close();
  }

  public int connections() {
    return 0;
  }
}
```

Running this gave us:

```
testOneConnection: expected:<1> but was:<0>
```

“Aha!” I said. “It found port 999. Cool! Now we just need to count the connection!”

So I changed the `SocketService` class as follows:

Commit 4, `SocketService.java`

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class SocketService {
  private ServerSocket serverSocket = null;
  private int connections = 0;

  public void serve(int port) throws Exception {
    serverSocket = new ServerSocket(port);
    try {
      Socket s = serverSocket.accept();
      s.close();
      connections++;
    } catch (IOException e) {
    }
  }

  public void close() throws Exception {
    serverSocket.close();
  }

  public int connections() {
    return connections;
  }
}
```

But this didn't work. It didn't even fail. When I ran the test, the test hung.

"What's going on?" I asked.

Jerry smiled. "See if you can figure it out, Alphonse. Trace it through."

"OK, let's see. The test program calls `serve`, which creates the socket and calls `accept`. Oh! `accept` doesn't return until it gets a connection! And so `serve` never returns, and we never get to call `connect`."

Jerry nodded. "So how are you going to fix this, Alphonse?"

I thought about this for a minute. I needed to call the `connect` function after calling `accept`; but when you call `accept`, it won't return until you call `connect`. At first this sounded impossible.

"It's not impossible, Alphonse." said Jerry. "You just have to create a thread."

I thought about this a little more. Yes, I could put the call to accept in a separate thread. Then I could invoke that thread and then call `connect`.

"I see what you mean about socket code being a little interesting." I said. And then I made the following changes:

Commit 5, `SocketService.java` *(partial)*

```
private Thread serverThread = null;

public void serve(int port) throws Exception {
  serverSocket = new ServerSocket(port);
  serverThread = new Thread(new Runnable() {
    public void run() {
      try {
        Socket s = serverSocket.accept();
        s.close();
        connections++;
      } catch (IOException e) {
      }
    }
  });
  serverThread.start();
}
```

“Nice use of the anonymous inner class, Alphonse.” said Jerry.

“Thanks.” It felt good to get a compliment from him. “But it does make for a lot of monkey tails at the end of the function.”

“We’ll refactor that later. First, let’s run the test.”

The test ran just fine, but Jerry looked pensive, like he’d just been lied to.

“Run the test again, Alphonse.”

I happily hit the run button, and it worked again.

“Again.” he said.

I looked at him for a second to see if he was joking. Clearly he was not. His eyes were locked on the screen, as though he were hunting a dribin. So I hit the button again and saw:

```
testOneConnection: expected:<1> but was:<0>
```

“Now wait a minute!” I hollered. “That can’t be!”

“Oh, yes it can.” Jerry said. “I was expecting it.”

I repeatedly pushed the button. In ten attempts I saw three failures. Was I losing my mind? How can a program behave like that?

“How did you know, Jerry? Are you related to the oracle of Aldebran?”

“No, I’ve just written this kind of code before, and I know a little bit about what to expect. Can you work out what’s happening? Think it through carefully.”

My brain was already hurting, but I started piecing things together. I went to the sketch-wall and began to draw.

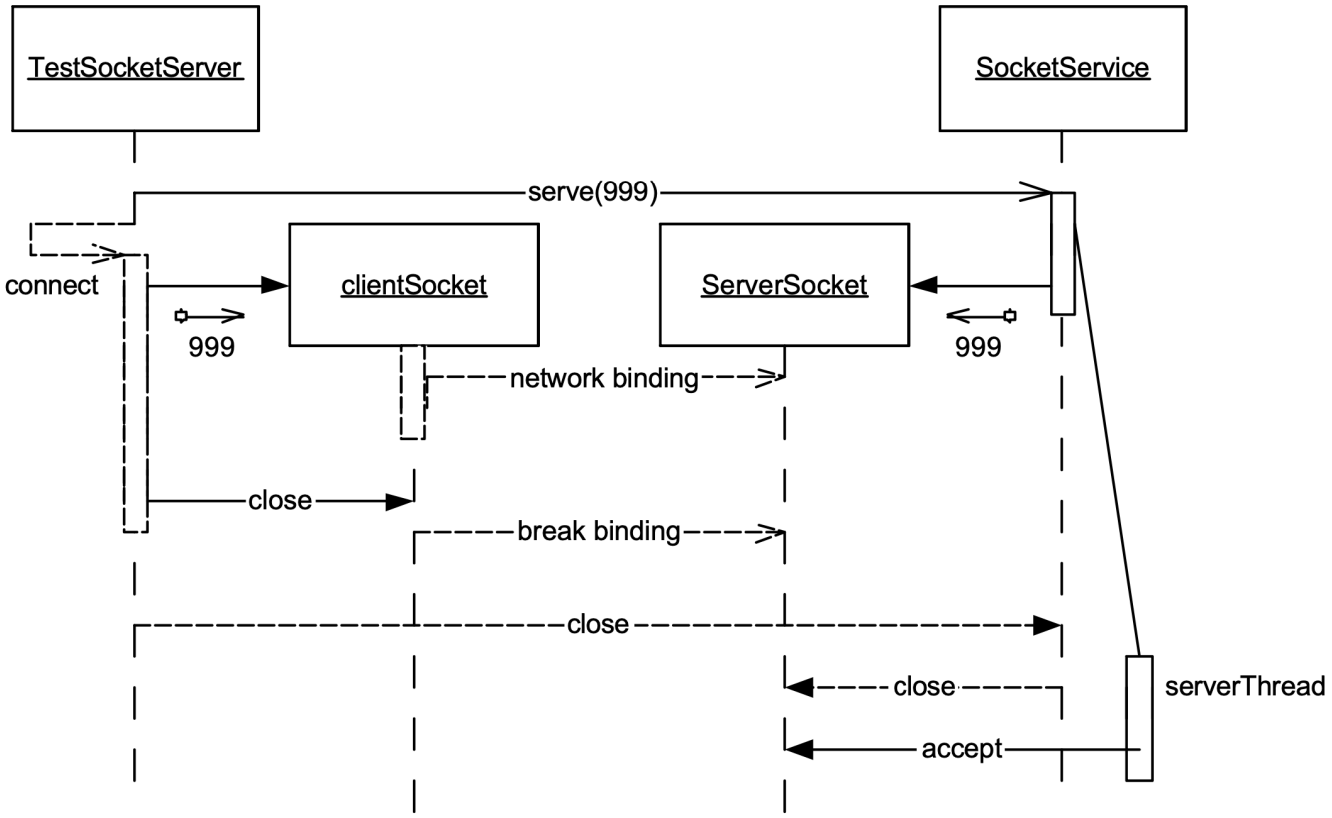

Figure 3. SocketServer protocol sequence diagram

When I had worked it out, I recited the scenario to Jerry. "`TestSocketServer` sent the `serve(999)` message to `SocketService`. `SocketService` created the `ServerSocket` and the `serverThread` and then returned. `TestSocketServer` then called `connect` which created the client socket. The two sockets must have found each other because we didn't get a `could not connect` error. The `ServerSocket` must have accepted the connection, but perhaps `serverThread` hadn't had a chance to run yet. And while `serverThread` was blocked, the `connect` function closed the client socket. Then `TestSocketServer` sent the close message to the `SocketService`, which closed the `serverSocket`. By the time the `serverThread` got a chance to call the `accept` function, the server socket was closed."

"I think you're right." Jerry said. "The two events — accept and close — are asynchronous, and the system is sensitive to their order. We call that a *race condition*. We have to make sure we always win the race."

We decided to test my hypothesis by putting a print statement in the catch block after the accept call. Sure enough, three times in ten, we saw the message.

"So how can we get our unit test to run?" Jerry asked me.

"It seems to me that the test can't just open the client socket and then immediately close it." I replied. "It needs to wait for the accept."

"We could just wait for 100ms real time before closing the client socket." he said. "Yeah, that would probably work." I replied. "But it's a bit ugly."

"Let's see if we can get it to work, and then we'll consider refactoring it."

So I made the following changes to the `connect` method:

Commit 6, `SocketService.java` *(partial)*

```
private void connect(int port) {
  try {
    Socket s = new Socket("localhost", port);
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
    }
    s.close();
  } catch (IOException e) {
    fail("could not connect");
  }
}
```

This made the tests pass 10 for 10.

“This is scary.” I said. “The fact that a test passes once doesn’t mean the code is working.”

“Yeah.” Jerry agreed. “When you are dealing with multiple threads, you have to keep an eye out for possible race conditions. Hitting the test button multiple times is a good habit to get into.”

“It’s a good thing we found this in our test cases.” I said. “Finding it after the system was running would have been a *lot* harder.”

Jerry just nodded.

The project so far:

Commit 7, `TestSocketServer.java`

```
import junit.framework.TestCase;
import junit.swingui.TestRunner;
import java.io.IOException;
import java.net.Socket;

public class TestSocketServer extends TestCase {
  public static void main(String[] args) {
    TestRunner.main(new String[]{"TestSocketServer"});
  }

  public TestSocketServer(String name) {
    super(name);
  }

  public void testOneConnection() throws Exception {
    SocketService ss = new SocketService();
    ss.serve(999);
    connect(999);
    ss.close();
    assertEquals(1, ss.connections());
  }

  private void connect(int port) {
    try {
      Socket s = new Socket("localhost", port);
      try {
        Thread.sleep(100);
      } catch (InterruptedException e) {
      }
      s.close();
    } catch (IOException e) {
      fail("could not connect");
    }
  }
}
```

Commit 7, SocketService.java

```
import java.io.IOException;
import java.net.*;

public class SocketService {
  private ServerSocket serverSocket = null;
  private int connections = 0;
  private Thread serverThread = null;

  public void serve(int port) throws Exception {
    serverSocket = new ServerSocket(port);
    serverThread = new Thread(new Runnable() {
      public void run() {
        try {
          Socket s = serverSocket.accept();
          s.close();
          connections++;
        } catch (IOException e) {
        }
      }
    });
    serverThread.start();
  }

  public void close() throws Exception {
    serverSocket.close();
  }

  public int connections() {
    return connections;
  }
}
```

We came back from our break, ready to continue the SocketService.

“We’ve proven that we can connect once. Let’s try to connect several times.” said Jerry.

“Sounds good,” I said. So I wrote the following test case:

Commit 8, `TestSocketServer.java` *(partial)*

```
public void testManyConnections() throws Exception {
  SocketService ss = new SocketService();
  ss.serve(999);
  for (int i = 0; i < 10; i++)
    connect(999);
  ss.close();
  assertEquals(10, ss.connections());
}
```

“OK, this fails,” I said.

“As it should.” Jerry replied. “The `SocketService` only calls `accept` once. We need to put that call into a loop.”

“When should the loop terminate?” I asked.

Jerry thought for a second and said: “When we call `close` on the `SocketService`.”

“Like this?” I said, making the following changes:

Commit 9, `SocketService.java`

```
import java.io.IOException;
import java.net.*;

public class SocketService {
  private ServerSocket serverSocket = null;
  private int connections = 0;
  private Thread serverThread = null;
  private boolean running = false;

  public void serve(int port) throws Exception {
    serverSocket = new ServerSocket(port);
    serverThread = new Thread(new Runnable() {
      public void run() {
        running = true;
        while (running) {
          try {
            Socket s = serverSocket.accept();
            s.close();
            connections++;
          } catch (IOException e) {
          }
        }
      }
    });
    serverThread.start();
  }

  public void close() throws Exception {
    running = false;
    serverSocket.close();
  }

  public int connections() {
    return connections;
  }
}
```

I ran the tests, and they both passed.

“Good.” I said. “Now we can connect as many times as we like. Unfortunately the `SocketService` doesn’t do very much when we connect to it. It just closes.”

“Yeah, let’s change that.” said Jerry. “Let’s have the `SocketService` send us a “Hello” message when we connect to it.”

I didn’t care for that. I said: “Why should we pollute `SocketService` with a “Hello” message just to satisfy our tests? It would be good to test that the `SocketService` can send a message, but we don’t want the message to be part of the `SocketService` code!”

“Right!” Jerry agreed. “We want the message to be specified by, and verified by, the test.”

“How do we do that?” I asked.

Jerry smiled and said: “We’ll use the *Mock Object* pattern. In short, we create an interface that the `SocketService` will execute after receiving a connection. We’ll have the test implement that interface to send the “Hello” message. Then we’ll have the test read the message from the client socket and verify that it was sent correctly.”

I didn’t know what the Mock Object pattern was, and his description of interfaces confused me. “Can you show me?” I asked.

So Jerry took the keyboard and began to type.

“First we’ll write the test.”

Commit 10, `TestSocketServer.java` *(partial)*

```
public void testSendMessage() throws Exception {
  SocketService ss = new SocketService();
  ss.serve(999, new HelloServer());
  Socket s = new Socket("localhost", 999);
  InputStream is = s.getInputStream();
  InputStreamReader isr = new InputStreamReader(is);
  BufferedReader br = new BufferedReader(isr);
  String answer = br.readLine();
  s.close();
  assertEquals("Hello", answer);
}
```

I examined this code carefully. “OK, so you’re creating something called a `HelloServer` and passing it into the `serve` method. That’s going to break all the other tests!”

“Good point!” Jerry exclaimed. “That means we need to refactor those other tests before we continue.”

“But the services in the other two tests don’t *do* anything.” I complained.

“Of course they do — they count connections! Remember how much you didn’t like that connections variable, and its accessor? Well now we’re going to get rid of them.”

“We are?”

“Just watch.” Jerry laughed. “First we’ll change the two tests, and add a *connections* variable to the test case.”

Commit 11, `TestSocketServer.java` (partial)

```
public void testOneConnection() throws Exception
{
  ss.serve(999, connectionCounter);
  connect(999);
  assertEquals(1, connections);
}

public void testManyConnections() throws Exception
{
  ss.serve(999, connectionCounter);
  for (int i=0; i<10; i++)
    connect(999);
  assertEquals(10, connections);
}
```

“Next we’ll make the interface.”

Commit 12, `SocketServer.java`

```
import java.net.Socket;

public interface SocketServer {
  public void serve(Socket s);
}
```

“Next we’ll create the `connectionCounter` variable and initialize it in the `TestSocketServer` constructor with an anonymous inner class that bumps the `connections` variable.

Commit 13, `TestSocketServer.java` (partial)

```
public class TestSocketServer extends TestCase {
  private int connections = 0;
  private SocketServer connectionCounter;

  // ...

  public TestSocketServer(String name) {
    super(name);
    connectionCounter = new SocketServer() {
      public void serve(Socket s) {
        connections++;
      }
    };
  }

  // ...
}
```

“Finally, we’ll make it all compile by adding the extra argument to the `serve` method of the `SocketService`, and commenting out the new test.

Commit 14, `SocketService.java`

```
public void serve(int port, SocketServer server) throws Exception {
  // ...
}
```

“OK, I see what you are doing.” I said. The two old tests should fail now because `SocketService` never invokes the `serve` method of its `SocketServer` argument.”

Sure enough, the tests failed for exactly that reason.

I knew what to do next. I grabbed the keyboard and made the following change:

Commit 15, `SocketService.java` *(partial)*

```
public class SocketService {
  private ServerSocket serverSocket = null;
  private int connections = 0;
  private Thread serverThread = null;
  private boolean running = false;
  private SocketServer itsServer;

  public void serve(int port, SocketServer server) throws Exception {
    itsServer = server;
    serverSocket = new ServerSocket(port);
    serverThread = new Thread(new Runnable() {
      public void run() {
        running = true;
        while (running) {
          try {
            Socket s = serverSocket.accept();
            itsServer.serve(s);
            s.close();
            connections++;
          } catch (IOException e) {
          }
        }
      }
    });
    serverThread.start();
  }

  // ...
```

This made all the tests run.

“Great!” Said Jerry. “Now we’ve got to make the new test work.”

So I uncommented the test and compiled it. It complained about `HelloServer`.

“Oh, right. We need to implement the `HelloServer`. It’s just going to spit the word “hello” out the socket, right?”

“Right,” Jerry confirmed.

So I added the following new class to the `TestSocketServer.java` file:

Commit 16, `TestSocketServer.java` *(partial)*

```
class HelloServer implements SocketServer {
  public void serve(Socket s) {
    try {
      PrintStream ps = new PrintStream(s.getOutputStream());
      ps.println("Hello");
    } catch (IOException e) {
    }
  }
}
```

The tests all passed.

“That was pretty easy,” said Jerry.

“Yeah. That *Mock Object* pattern is pretty useful. It let us keep all the test code in the test module. The `SocketService` doesn’t know anything about it.”

“It’s even more useful than that.” Jerry replied. “Real servers will also implement the `SocketServer` interface.”

“I can see that.” I said. “Interesting that the needs of a unit test drove us to create a design that would be generally useful."

“That happens all the time.” Said Jerry. “Tests are users too. The needs of the tests are often the same as the needs of the real users.”

“But why is it called *Mock Object*?”

“Think of it this way. The `HelloServer` is a substitute for, or a mock-up of, a real server. The pattern allows us to substitute test mock-ups into real application code.”

“I see.” I said. “Well, we should clean this code up now and get rid of that useless `connections` variable.”

“Agreed.”

So we did a little cleanup and took another break. The resulting `SocketService` is shown below.

Commit 17, SocketService.java

```
import java.io.IOException;
import java.net.*;

public class SocketService {
  private ServerSocket serverSocket = null;
  private Thread serverThread = null;
  private boolean running = false;
  private SocketServer itsServer;

  public void serve(int port, SocketServer server) throws Exception {
    itsServer = server;
    serverSocket = new ServerSocket(port);
    serverThread = makeServerThread();
    serverThread.start();
  }

  private Thread makeServerThread() {
    return new Thread(new Runnable() {
      public void run() {
        running = true;
        while (running) {
          acceptAndServeConnection();
        }
      }
    });
  }

  private void acceptAndServeConnection() {
    try {
      Socket s = serverSocket.accept();
      itsServer.serve(s);
      s.close();
    } catch (IOException e) {
    }
  }

  public void close() throws Exception {
    running = false;
    serverSocket.close();
  }
}
```

The turbo on level 17 was down again, so I had to use the ladder-shaft. While sliding down the ladder I got to thinking about how remarkable it was to use tests as a design tool. Lost in my thoughts I got a bit careless with the coriolis forces and bumped my elbow against the shaft. It was still stinging when I rejoined Jerry in the lab.

"Nicely done on the cleanup," said Jerry. "And with the `HelloServer`, we now know that we can send a message through the socket."

I knew Jerry was going to start thinking about refactoring now. I wanted to get the jump on him. So I looked carefully at the code, and I remembered what he told me about duplication. "There's some duplicated code in the unit tests." I said. "In each of the three tests we create and close the `SocketService`. We should get rid of that."

"Good eye!" He replied. "Let's move that into the Setup and Teardown functions." He grabbed the keyboard and made the following changes.

Commit 18, `TestSocketServer.java` *(partial)*

```
private SocketService ss;

public void setUp() throws Exception {
  ss = new SocketService();
}

public void tearDown() throws Exception {
  ss.close();
}
```

Then he removed all the `ss = newSocketService();` and `ss.close();` lines from the three tests.

"That's a little better," I said. "So now let's see if we can send a message the other direction."

"Exactly what I was thinking." Said Jerry. "And I've got just the way to do it."

Jerry grabbed the keyboard and began to type a new test case.

Commit 19, `TestSocketServer.java` *(partial)*

```
public void testReceiveMessage() throws Exception {
  ss.serve(999, new EchoService());
  Socket s = new Socket("localhost", 999);
  InputStream is = s.getInputStream();
  InputStreamReader isr = new InputStreamReader(is);
  BufferedReader br = new BufferedReader(isr);
  OutputStream os = s.getOutputStream();
  PrintStream ps = new PrintStream(os);
  ps.println("MyMessage");
  String answer = br.readLine();
  s.close();
  assertEquals("MyMessage", answer);
}
```

"Ouch! That's pretty ugly." I said.

"Yeah, it is." Replied Jerry. "Let's get it working and then we'll clean it up. We don't want a mess like

that laying around for long! You see where I am going with this though, don't you?"

"Yes, I thnk so." I said. "`EchoService` will recieve a message from the socket and just send it right back. So your test just sends `'MyMessage'`, then reads it back again."

"Right. Care to take a crack at writing `EchoService`?"

"Sure," I said, and I grabbed the keyboard.

Commit 20, `TestSocketServer.java` *(partial)*

```
class EchoService implements SocketServer {
  public void serve(Socket s) {
    try {
      InputStream is = s.getInputStream();
      InputStreamReader isr = new InputStreamReader(is);
      BufferedReader br = new BufferedReader(isr);
      OutputStream os = s.getOutputStream();
      PrintStream ps = new PrintStream(os);
      String token = br.readLine();
      ps.println(token);
    } catch (IOException e) {
    }
  }
}
```

"Ick." I said. "More of the same kind of ugly code. We keep on having to create `PrintStream` and `BufferedReader` objects from the socket. We really need to clean that up."

"We'll do that just as soon as the test works," said Jerry. And then he looked at me expectantly. "Oh!" I said, "I forgot to run the test."

Sheepishly, I pushed the test button and watched it pass.

"That wasn't hard to get working," I said; "now let's get rid of that ugly code."

Keeping the keyboard, I extracted several functions from the previous mess in `EchoService`:

Commit 21, `TestSocketServer.java` *(partial)*

```
class EchoService implements SocketServer {
  public void serve(Socket s) {
    try {
      BufferedReader br = getBufferedReader(s);
      PrintStream ps = getPrintStream(s);
      String token = br.readLine();
      ps.println(token);
    } catch (IOException e) {
    }
  }

  private PrintStream getPrintStream(Socket s) throws IOException {
    OutputStream os = s.getOutputStream();
    PrintStream ps = new PrintStream(os);
    return ps;
  }

  private BufferedReader getBufferedReader(Socket s) throws IOException {
    InputStream is = s.getInputStream();
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);
    return br;
  }
}
```

"Yes." Said Jerry. "That makes the serve method of `EchoService` much better, but it clutters the class quite a bit. Also, it doesn't really help the `testRecieveMessage` function. That function has the same kindofugliness. I wonder if `getBufferedReader` and `getPrintStream` are in the right place."

"This is going to be a recurring problem," I said. "Anybody who wants to use SocketService is going to have to convert the socket into a `BufferedReader` and a `PrintStream`."

"I think that's your answer!" Jerry replied. "The `getBufferedReader` and `getPrintStream` methods really belong in `SocketService`."

This made a lot of sense to me. So I moved the two functions into the `SocketService` class and changed `EchoService` accordingly.

Commit 22, `SocketService.java` *(partial)*

```
public class SocketService {
  // ...

  public static PrintStream getPrintStream(Socket s) throws IOException {
    OutputStream os = s.getOutputStream();
    PrintStream ps = new PrintStream(os);
    return ps;
  }

  public static BufferedReader getBufferedReader(Socket s) throws IOException {
    InputStream is = s.getInputStream();
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);
    return br;
  }
}
```

Commit 22, `TestSocketServer.java` *(partial)*

```
class EchoService implements SocketServer {
  public void serve(Socket s) {
    try {
      BufferedReader br = SocketService.getBufferedReader(s);
      PrintStream ps = SocketService.getPrintStream(s);
      String token = br.readLine();
      ps.println(token);
    } catch (IOException e) {
    }
  }
}
```

The tests all still ran. Keeping my momentum, I said: "Now I should be able to fix the `testReceiveMessage` method too." So, while Jerry watched, I made that change too.

Commit 23, `TestSocketServer.java` *(partial)*

```
public void testReceiveMessage() throws Exception {
  ss.serve(999, new EchoService());
  Socket s = new Socket("localhost", 999);
  BufferedReader br = SocketService.getBufferedReader(s);
  PrintStream ps = SocketService.getPrintStream(s);
  ps.println("MyMessage");
  String answer = br.readLine();
  s.close();
  assertEquals("MyMessage", answer);
}
```

"Yeah, that's a lot better," Jerry said.

"Not only that, but the tests still pass." I said. Then I noticed something. "Oops, there's another one in `testSendMessage`." And so I made that change too.

Commit 24, `TestSocketServer.java` *(partial)*

```
public void testSendMessage() throws Exception {
  ss.serve(999, new HelloServer());
  Socket s = new Socket("localhost", 999);
  BufferedReader br = SocketService.getBufferedReader(s);
  String answer = br.readLine();
  assertEquals("Hello", answer);
}
```

The tests all still ran. I sat there looking through the TestSocketServer class, trying to find any other code I could eliminate.

"Are you done?" asked Jerry.

After a few more seconds of scanning, I relaxed, and said, "I think so."

"Good." He said. "I was starting to see smoke coming out of your ears. I think it's time for a quick break."

"OK, but I have a question first."

"Shoot."

I looked back and forth between Jerry and the code for a few seconds, and then I said: "We haven't made a single change to `SocketService`. We've added two new tests — `testSendMessage`, and `testReceiveMessage` — and they both just worked. And yet we spent a lot of time writing those tests, and also refactoring. What good did it do us? We didn't change the production code at all!"

Jerry raised an eyebrow and gave me an impenetrable regard. "It's interesting that you noticed that. Do you think we wasted our time?"

"It didn't feel like a waste, because we proved to ourselves that you could send and receive messages. Still, we wrote and refactored a lot of code that didn't help us add anything to the production code."

"You don't think that `getBufferedReader` and `getPrintStream` were worth putting into production?"

"They're pretty trivial, and all they're really doing is supporting the tests."

Jerry sighed and looked away for a minute. Then he turned back to me and said: "If you were just coming onto this project, and I showed you these tests, what would they teach you?"

That was a strange question. My first reaction was to answer by saying that they'd teach me that the authors had wanted to prove their code worked. But I held that thought back. Jerry wouldn't have asked me that question if he hadn't wanted me to carefully think about the answer.

What would I learn by reading those tests? I'd learn how to create a `SocketService`, and how to bind a `SocketServer` derivative to it. I'd also learn how to send and receive messages. I'd learn the names and locations of the classes in the framework, and how to use them.

So I gathered up my courage and said: "You mean we wrote these tests as examples to show to others?"

"That's part of the reason, Alphonse. Yes, others will be able to read these tests and see how to work the code we're writing. They'll also be able to work through our reasoning. Moreover, they'll be able to compile and execute these tests in order to prove to themselves that our reasoning was sound."

"There's more to it than that—" he continued, "but we'll leave that for another time. Now let's take our break."

My elbow still twinged, so I was glad to see that the turbo was repaired. On the way up the lift I kept rolling same thought around in my head. "Tests are a form of documentation. Compilable, executable, and always in sync."

Index