The Craftsman: One.

Robert C. Martin
13 February 2002
This chapter is derived from a chapter of Principles, Patterns, and Practices of Agile Software Development*, Robert C. Martin, Prentice Hall, 2002.*

Dear Diary,

13 February 2002,

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 determined to do a really impressive job. Listing 1 shows what I wrote. I made sure it was well commented, and neatly formatted.

Listing 1
`GeneratePrimes.java version 1.`

```
/**
 * 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
 */
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 Listing 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.

Listing 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 didnt' 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, lets 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. "Lets 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...

To Be Continued Next Month

The Craftsman: Two.

Robert C. Martin
13 February 2002
This chapter is derived from a chapter of Principles, Patterns, and Practices of Agile Software Development*, Robert C. Martin, Prentice Hall, 2002.*

Dear Diary,

13 February 2002,

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. ...

In last month's column Alphonse, the apprentice, was asked by Jerry, the Jouneyman, to write a program to generate prime numbers using the sieve of Eratosthenes. Jerry, noting that Alponse implemented the entire algorithm in a single gargantuan function has asked Alphonse to split it by separating the three concepts: inialization, generation, and output preparation;... but Alphonse doesn't know how to start...

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 Listing 3. I've marked my changes in bold, 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.

Listing 3

PrimeGenerator.java, version 2

```
/**
 * 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. Listing 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.

Listing 4

`PrimeGenerator.java, version 3 (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 Listing 5.

Listing 5

PrimeGenerator.java version 4 (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 `putUncrossed-IntegersIntoResult` 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 Listing 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?

Listing 6

`PrimeGenerator.java, version 5 (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;
    }
```

To Be Continued Next Month

The Craftsman: Three.

Robert C. Martin
13 February 2002
This chapter is derived from a chapter of Principles, Patterns, and Practices of Agile Software Development*, Robert C. Martin, Prentice Hall, 2002.*

Previously Alphonse, the apprentice, was asked by Jerry, the Jouneyman, to write a program to generate prime numbers using the sieve of Eratosthenes. Jerry has been reviewing the code and helping Alphonse refactor it. He is not happy with Alphonse's work. In the last column Alphonse had just grabbed the keyboard and done a refactoring that he thought Jerry might approve of...

...Jerry was just barely nodding his head. Did he actually like what I did?

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 Listing 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.

Listing 6

```
TestGeneratePrimes.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 Listing 8. 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 listings 7 and 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.

Listing 7

`PrimeGenerator.java (final)`

```
/**
 * 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;
  }
}
```

Listing 8
TestGeneratePrimes.java (final)

```
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.

To Be Continued Next Month

The Craftsman: 4

Robert C. Martin
12 July 2002

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.

```
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."

The Craftsman: 5

Robert C. Martin
24 July 2002

Jerry had asked me to write a program that generated prime factors. I wrote it, and it worked just fine. Then Jerry just deleted it. I got a bit angry, but Jerry just said: "Don't 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, "Lets 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.

```
  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.

```
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:

```
  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:

```
  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:

```
  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 it is?"

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:

```
  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:

```
  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:

```
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. "Lets clean this up."

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

```
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. Obligingly, I typed in the following test case:

```
  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. "Lets 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.

```
  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 lets 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.

```
  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 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 lets 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."

"Lets try it." he said.

So I typed in the following:

```
  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:

```
  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 4X4 will fail.

```
  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. "Lets try 1000."

"Ah, the shotgun approach. OK, give it a try."

```
  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.