# 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.