Refactoring a Test

Why Refactor Tests?

Tests can quickly become a bottleneck in an agile development process. This may not be immediately obvious to those who have never experienced the difference between simple, easily understood tests and complex, obtuse, hard-to-maintain tests. The productivity difference can be staggering!

This section of the book acts as a "motivating example" for the entire book by showing you how much of a difference refactoring tests can make. It walks you through an example starting with a complex test and, step by step, refactors it to a simple, easily understood test. Along the way, I will point out some key smells and the patterns that we can use to remove them. Ideally, this exercise will whet your appetite for more.

A Complex Test

Here is a test that is not atypical of some of the tests I have seen on various projects:

```
public void testAddItemQuantity_severalQuantity_v1(){
   Address billingAddress = null;
   Address shippingAddress = null;
   Customer customer = null;
   Product product = null;
   Invoice invoice = null;
   try {
      //   Set up fixture
      billingAddress = new Address("1222 1st St SW",
            "Calgary", "Alberta", "T2N 2V2","Canada");
      shippingAddress = new Address("1333 1st St SW",
            "Calgary", "Alberta", "T2N 2V2", "Canada");
      customer = new Customer(99, "John", "Doe",
                              new BigDecimal("30"),
                              billingAddress,
                              shippingAddress);
      product = new Product(88, "SomeWidget",
                            new BigDecimal("19.99"));
      invoice = new Invoice(customer);
```

```
         // Exercise SUT
         invoice.addItemQuantity(product, 5);
         // Verify outcome
         List lineItems = invoice.getLineItems();
         if (lineItems.size() == 1) {
            LineItem actItem = (LineItem) lineItems.get(0);
            assertEquals("inv", invoice, actItem.getInv());
            assertEquals("prod", product, actItem.getProd());
            assertEquals("quant", 5, actItem.getQuantity());
            assertEquals("discount", new BigDecimal("30"),
                            actItem.getPercentDiscount());
            assertEquals("unit price",new BigDecimal("19.99"),
                               actItem.getUnitPrice());
            assertEquals("extended", new BigDecimal("69.96"),
                            actItem.getExtendedPrice());
         } else {
            assertTrue("Invoice should have 1 item", false);
         }
      } finally {
         // Teardown
         deleteObject(invoice);
         deleteObject(product);
         deleteObject(customer);
         deleteObject(billingAddress);
         deleteObject(shippingAddress);
      }
   }
```

This test is quite long[1] and is much more complicated than it needs to be. This *Obscure Test* (page 186) is difficult to understand because the sheer number of lines in the test makes it hard to see the big picture. It also suffers from a number of other problems that we will address individually.

Cleaning Up the Test

Let's look at each of the various parts of the test.

Cleaning Up the Verification Logic

First, let's focus on the part that verifies the expected outcome. Maybe we can infer from the assertions which test conditions this test is trying to verify.

[1] While the need to wrap lines to keep them at 65 characters makes this code look even longer than it really is, it is still unnecessarily long. It contains 25 executable statements including initialized declarations, 6 lines of control statements, 4 in-line comments, and 2 lines to declare the test method—giving a total of 37 lines of unwrapped source code.

```
      List lineItems = invoice.getLineItems();
      if (lineItems.size() == 1) {
         LineItem actItem = (LineItem) lineItems.get(0);
         assertEquals("inv", invoice, actItem.getInv());
         assertEquals("prod", product, actItem.getProd());
         assertEquals("quant", 5, actItem.getQuantity());
         assertEquals("discount", new BigDecimal("30"),
                        actItem.getPercentDiscount());
         assertEquals("unit price",new BigDecimal("19.99"),
                          actItem.getUnitPrice());
         assertEquals("extended", new BigDecimal("69.96"),
                        actItem.getExtendedPrice());
      } else {
         assertTrue("Invoice should have 1 item", false);
      }
```

A simple problem to fix is the obtuse assertion on the very last line. Calling `assertTrue` with an argument of `false` should always result in a test failure, so why don't we say so directly? Let's change this to a call to `fail`:

```
      List lineItems = invoice.getLineItems();
      if (lineItems.size() == 1) {
         LineItem actItem = (LineItem) lineItems.get(0);
         assertEquals("inv", invoice, actItem.getInv());
         assertEquals("prod", product, actItem.getProd());
         assertEquals("quant", 5, actItem.getQuantity());
         assertEquals("discount", new BigDecimal("30"),
                        actItem.getPercentDiscount());
         assertEquals("unit price",new BigDecimal("19.99"),
                          actItem.getUnitPrice());
         assertEquals("extended", new BigDecimal("69.96"),
                        actItem.getExtendedPrice());
      } else {
         fail("Invoice should have exactly one line item");
      }
```

We can think of this move as an Extract Method [Fowler] refactoring, because we are replacing the *Stated Outcome Assertion* (see *Assertion Method* on page 362) with a hard-coded parameter with a more intent-revealing call to a *Single Outcome Assertion* (see *Assertion Method*) method that encapsulates the call.

Of course, this set of assertions suffers from several more problems. For example, why do we need so many of them? It turns out that many of these assertions are testing fields set by the constructor for the `LineItem`, which is itself covered by another unit test. So why repeat these assertions here? It will just create more test code to maintain when the logic changes.

One solution is to use a single assertion on an *Expected Object* (see *State Verification* on page 462) instead of one assertion per object field. First, we define an object that looks exactly how we expect the result to look. In this case, we create an expected `LineItem` with the fields filled in with the expected values, including the `unitPrice` and `extendedPrice` initialized from the `product`.

```
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
   LineItem expected =
      new LineItem(invoice, product, 5,
                   new BigDecimal("30"),
                   new BigDecimal("69.96"));
   LineItem actItem = (LineItem) lineItems.get(0);
   assertEquals("invoice", expected.getInv(),
                           actItem.getInv());
   assertEquals("product", expected.getProd(),
                           actItem.getProd());
   assertEquals("quantity",expected.getQuantity(),
                           actItem.getQuantity());
   assertEquals("discount",
                expected.getPercentDiscount(),
                actItem.getPercentDiscount());
   assertEquals("unit pr", new BigDecimal("19.99"),
                           actItem.getUnitPrice());
   assertEquals("extend pr",new BigDecimal("69.96"),
                            actItem.getExtendedPrice());
} else {
   fail("Invoice should have exactly one line item");
}
```

Once we have created our *Expected Object*, we can then assert on it using `assertEquals`:

```
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
   LineItem expected =
      new LineItem(invoice, product,5,
                   new BigDecimal("30"),
                   new BigDecimal("69.96"));
   LineItem actItem = (LineItem) lineItems.get(0);
   assertEquals("invoice", expected, actItem);
} else {
   fail("Invoice should have exactly one line item");
}
```

Clearly, the Preserve Whole Object [Fowler] refactoring makes the code a lot simpler and more obvious. But wait! Why do we have an `if` statement in a test? If there are several paths through a test, how do we know which one is actually being executed? It would be a lot better if we could eliminate this *Conditional Test Logic* (page 200). Luckily for us, the pattern *Guard Assertion* (page 490) is designed to handle exactly this case. We simply use a Replace Conditional with Guard Clause [Fowler] refactoring to replace the `if ... else fail() ...` sequence with an assertion on the same condition. This *Guard Assertio*n halts execution if the condition is not met without introducing *Conditional Test Logic*.

```
      List lineItems = invoice.getLineItems();
      assertEquals("number of items", 1,lineItems.size());
      LineItem expected =
         new LineItem(invoice, product, 5,
                      new BigDecimal("30"),
                      new BigDecimal("69.96"));
      LineItem actItem = (LineItem) lineItems.get(0);
      assertEquals("invoice", expected, actItem);
```

So far, we have reduced 11 lines of verification code to just 4, and those 4 lines are a lot simpler code to boot.[2] Some people might suggest that this refactoring is good enough. But can't we make this assertion even more obvious? What are we really trying to verify? We are trying to say that there should be only one line item and it should look exactly like our `expectedLineItem`. We can say this explicitly by using an Extract Method refactoring to define a *Custom Assertion* (page 474).

```
      LineItem expected =
         new LineItem(invoice, product, 5,
                      new BigDecimal("30"),
                      new BigDecimal("69.96"));
      assertContainsExactlyOneLineItem(invoice, expected);
```

That is better! Now we have the verification part of the test down to just two lines. Let's review what the whole test looks like:

```
   public void testAddItemQuantity_severalQuantity_v6(){
      Address billingAddress = null;
      Address shippingAddress = null;
      Customer customer = null;
      Product product = null;
      Invoice invoice = null;
      try {
         //   Set up fixture
         billingAddress = new Address("1222 1st St SW",
                "Calgary", "Alberta", "T2N 2V2", "Canada");
         shippingAddress = new Address("1333 1st St SW",
                "Calgary", "Alberta", "T2N 2V2", "Canada");
         customer = new Customer(99, "John", "Doe",
                                 new BigDecimal("30"),
                                 billingAddress,
                                 shippingAddress);
         product = new Product(88, "SomeWidget",
                               new BigDecimal("19.99"));
         invoice = new Invoice(customer);
         // Exercise SUT
         invoice.addItemQuantity(product, 5);
```

[2] It's a good thing we are not being rewarded for the number of lines of code we write! This is yet another example of why KLOC is such a poor measure of productivity.

```
         // Verify outcome
         LineItem expected =
            new LineItem(invoice, product, 5,
                         new BigDecimal("30"),
                         new BigDecimal("69.96"));
         assertContainsExactlyOneLineItem(invoice, expected);
      } finally {
         // Teardown
         deleteObject(invoice);
         deleteObject(product);
         deleteObject(customer);
         deleteObject(billingAddress);
         deleteObject(shippingAddress);
      }
   }
```

Cleaning Up the Fixture Teardown Logic

Now that we have cleaned up the result verification logic, let's turn our attention to the `finally` block at the end of the test. What is *this* code doing?

```
      } finally {
         // Teardown
         deleteObject(invoice);
         deleteObject(product);
         deleteObject(customer);
         deleteObject(billingAddress);
         deleteObject(shippingAddress);
      }
```

Most modern languages have an equivalent construct to the `try/finally` block that can be used to ensure that code gets run even when an error or exception occurs. In a *Test Method* (page 348), the `finally` block ensures that any cleanup code gets run regardless of whether the test passed or failed. A failed assertion throws an exception, which would transfer control back to the *Test Automation Framework's* (page 298) exception-handling code, so we use the `finally` block to clean up first. This approach means that we avoid having to catch the exception and then rethrow it.

In this test, the `finally` block calls the `deleteObject` method on each of the objects created by the test. Unfortunately, this code suffers from a fatal flaw. Have you noticed it yet?

Things could go wrong during the teardown itself. What happens if the first call to `deleteObject` throws an exception? As coded here, none of the other calls to `deleteObject` would be executed. The solution is to use a nested `try/finally` block around this first call, thereby ensuring that the second call to `deleteObject` always executes. But what if the second call fails? In this case, we would need a total of six nested try/finally blocks to make this maneuver work. That would almost double the length of the test, and we cannot afford to write and maintain so much code in each test.

```
      } finally {
         //      Teardown
         try {
            deleteObject(invoice);
         } finally {
            try {
               deleteObject(product);
            } finally {
               try {
                  deleteObject(customer);
               } finally {
                  try {
                     deleteObject(billingAddress);
                  } finally {
                     deleteObject(shippingAddress);
                  }
               }
            }
         }
```

The problem is that we now have a *Complex Teardown* (see *Obscure Test*). What are the chances of getting this code right? And how do we test the test code? Clearly, our current approach is not going to be very effective.

Of course, we could move this code into the tearDown method. That would have the advantage of removing it from the *Test Method*. Also, because the tearDown method acts as a finally block, we would get rid of the outermost try/finally. Unfortunately, this strategy doesn't address the root of the problem: the need to write detailed teardown code in each test.

We could try to avoid creating the objects in the first place by using a *Shared Fixture* (page 317) that is not torn down between tests. Unfortunately, this approach is likely to lead to a number of test smells, including *Unrepeatable Test* (see *Erratic Test* on page 228) and *Interacting Tests* (see *Erratic Test*), caused by interactions via the shared fixture. Another issue is that the references to objects used from the shared fixture are often *Mystery Guests* (see *Obscure Test*).[3]

The best solution is to use a *Fresh Fixture* (page 311) but to avoid writing teardown code for every test. To do so, we can use an in-memory fixture that is automatically garbage collected. This approach won't work, however, if the objects we create are persistent (e.g., if they are saved in a database). While it is best to construct the system architecture so that most of our tests can

[3] The test reader cannot see the objects being used by the test.

be executed without the database, we almost always have some tests that need it. In these cases, we can extend the *Test Automation Framework* to do most of the work for us. We can add a means to register each object we create with the framework so that it can do the deleting for us.

First, we need to register each object as we create it:

```
   //   Set up fixture
   billingAddress = new Address("1222 1st St SW", "Calgary",
                     "Alberta", "T2N 2V2", "Canada");
   registerTestObject(billingAddress);
   shippingAddress = new Address("1333 1st St SW", "Calgary",
                      "Alberta","T2N 2V2", "Canada");
   registerTestObject(shippingAddress);
   customer = new Customer(99, "John", "Doe",
                           new BigDecimal("30"),
                           billingAddress,
                           shippingAddress);
   registerTestObject(shippingAddress);
   product = new Product(88, "SomeWidget",
                         new BigDecimal("19.99"));
   registerTestObject(shippingAddress);
   invoice = new Invoice(customer);
   registerTestObject(shippingAddress);
```

Registration consists of adding the object to a collection of test objects:

```
List testObjects;

protected void setUp() throws Exception {
   super.setUp();
   testObjects = new ArrayList();
}

protected void registerTestObject(Object testObject) {
   testObjects.add(testObject);
}
```

In the `tearDown` method, we iterate through the list of test objects and delete each one:

```
public void tearDown() {
   Iterator i = testObjects.iterator();
   while (i.hasNext()) {
      try {
         deleteObject(i.next());
      } catch (RuntimeException e) {
         // Nothing to do; we just want to make sure
         // we continue on to the next object in the list
      }
   }
}
```

Now our test looks like this:

```
public void testAddItemQuantity_severalQuantity_v8(){
   Address billingAddress = null;
   Address shippingAddress = null;
   Customer customer = null;
   Product product = null;
   Invoice invoice = null;
   //   Set up fixture
   billingAddress = new Address("1222 1st St SW", "Calgary",
                  "Alberta", "T2N 2V2", "Canada");
   registerTestObject(billingAddress);
   shippingAddress = new Address("1333 1st St SW", "Calgary",
                   "Alberta","T2N 2V2", "Canada");
   registerTestObject(shippingAddress);
   customer = new Customer(99, "John", "Doe",
                        new BigDecimal("30"),
                        billingAddress,
                        shippingAddress);
   registerTestObject(shippingAddress);
   product = new Product(88, "SomeWidget",
                      new BigDecimal("19.99"));
   registerTestObject(shippingAddress);
   invoice = new Invoice(customer);
   registerTestObject(shippingAddress);
   // Exercise SUT
   invoice.addItemQuantity(product, 5);
   // Verify outcome
   LineItem expected =
      new LineItem(invoice, product, 5,
                   new BigDecimal("30"),
                   new BigDecimal("69.96"));
   assertContainsExactlyOneLineItem(invoice, expected);
}
```

We have been able to remove the `try/finally` block and, except for the additional calls to `registerTestObject`, our code is much simpler. But we can still clean this code up a bit more. Why, for example, do we need to declare the variables and initialize them to null, only to reinitialize them later? This action was needed with the original test because they had to be accessible in the `finally` block; now that we have removed this block, we can combine the declaration with the initialization:

```
public void testAddItemQuantity_severalQuantity_v9(){
   //   Set up fixture
   Address billingAddress = new Address("1222 1st St SW",
            "Calgary", "Alberta", "T2N 2V2", "Canada");
   registerTestObject(billingAddress);
   Address shippingAddress = new Address("1333 1st St SW",
            "Calgary", "Alberta", "T2N 2V2", "Canada");
```

```
      registerTestObject(shippingAddress);
      Customer customer = new Customer(99, "John", "Doe",
                                       new BigDecimal("30"),
                                       billingAddress,
                                       shippingAddress);
      registerTestObject(shippingAddress);
      Product product = new Product(88, "SomeWidget",
                                    new BigDecimal("19.99"));
      registerTestObject(shippingAddress);
      Invoice invoice = new Invoice(customer);
      registerTestObject(shippingAddress);
      // Exercise SUT
      invoice.addItemQuantity(product, 5);
      // Verify outcome
      LineItem expected =
         new LineItem(invoice, product, 5,
                      new BigDecimal("30"),
                      new BigDecimal("69.95"));
      assertContainsExactlyOneLineItem(invoice, expected);
   }
```

Cleaning Up the Fixture Setup

Now that we have cleaned up the assertions and the fixture teardown, let's turn our attention to the fixture setup. One obvious "quick fix" would be to take each of the calls to a constructor, take the subsequent call to `registerTestObject`, and use an Extract Method refactoring to define a *Creation Method* (page 415). This will make the test a bit simpler to read and write. The use of *Creation Methods* has another advantage: They encapsulate the API of the SUT and reduce the test maintenance effort when the various object constructors change by allowing us to modify only a single place rather than having to change each test.

```
   public void testAddItemQuantity_severalQuantity_v10(){
      //   Set up fixture
      Address billingAddress =
         createAddress( "1222 1st St SW", "Calgary", "Alberta",
                        "T2N 2V2", "Canada");
      Address shippingAddress =
         createAddress( "1333 1st St SW", "Calgary", "Alberta",
                        "T2N 2V2", "Canada");
      Customer customer =
         createCustomer( 99, "John", "Doe", new BigDecimal("30"),
                         billingAddress, shippingAddress);
      Product product =
         createProduct( 88,"SomeWidget",new BigDecimal("19.99"));
      Invoice invoice = createInvoice(customer);
      // Exercise SUT
      invoice.addItemQuantity(product, 5);
      // Verify outcome
      LineItem expected =
         new LineItem(invoice, product,5, new BigDecimal("30"),
                      new BigDecimal("69.96"));
      assertContainsExactlyOneLineItem(invoice, expected);
   }
```

This fixture setup logic still suffers from several problems. The first problem is that it is difficult to tell how the fixture is related to the expected outcome of the test. Do the customer's particulars affect the outcome in some way? Does the customer's address affect the outcome? What is this test really verifying?

The other problem is that this test exhibits *Hard-Coded Test Data* (see *Obscure Test*). Given that our SUT persists all objects we create in a database, the use of *Hard-Coded Test Data* may result in an *Unrepeatable Test*, an *Interacting Test*, or a *Test Run War* (see *Erratic Test*) if any of the fields of the customer, product, or invoice must be unique.

We can solve this problem by generating a unique value for each test and then using that value to seed the attributes of the objects we create for the test. This approach will ensure that the test creates different objects each time the test is run. Because we have already moved the object creation logic into *Creation Methods*, this step is relatively easy; we just put this logic into the *Creation Method* and remove the corresponding parameters. This is another application of the Extract Method refactoring, in which we create a new, parameterless version of the *Creation Method*.

```
   public void testAddItemQuantity_severalQuantity_v11(){
      final int QUANTITY = 5;
      //   Set up fixture
      Address billingAddress = createAnAddress();
      Address shippingAddress = createAnAddress();
      Customer customer = createACustomer(new BigDecimal("30"),
               billingAddress, shippingAddress);
      Product product = createAProduct(new BigDecimal("19.99"));
      Invoice invoice = createInvoice(customer);
      // Exercise SUT
      invoice.addItemQuantity(product, QUANTITY);
      // Verify outcome
      LineItem expected =
         new LineItem(invoice, product, 5, new BigDecimal("30"),
                      new BigDecimal("69.96"));
      assertContainsExactlyOneLineItem(invoice, expected);
   }
   private Product createAProduct(BigDecimal unitPrice) {
      BigDecimal uniqueId = getUniqueNumber();
      String uniqueString = uniqueId.toString();
      return new Product(uniqueId.toBigInteger().intValue(),
                         uniqueString, unitPrice);
   }
```

We call this pattern an *Anonymous Creation Method* (see *Creation Method*) because we are declaring that we don't care about the particulars of the object. If the expected behavior of the SUT depends on a particular value, we can either pass the value as a parameter or imply it in the name of the creation method.

This test looks a lot better now, but we are not done yet. Does the expected outcome depend in any way on the addresses of the customer? If not, we can hide their construction completely by using an Extract Method refactoring (again!) to create a version of the `createACustomer` method that fabricates them for us.

```
public void testAddItemQuantity_severalQuantity_v12(){
   //  Set up fixture
   Customer cust = createACustomer(new BigDecimal("30"));
   Product prod = createAProduct(new BigDecimal("19.99"));
   Invoice invoice = createInvoice(cust);
   // Exercise SUT
   invoice.addItemQuantity(prod, 5);
   // Verify outcome
   LineItem expected = new LineItem(invoice, prod, 5,
         new BigDecimal("30"), new BigDecimal("69.96"));
   assertContainsExactlyOneLineItem(invoice, expected);
}
```

By moving the calls that create the addresses into the method that creates the customer, we have made it clear that the addresses do not affect the logic that we are verifying in this test. The outcome does depend on the customer's discount, however, so we pass the discount percentage to the customer creation method.

We still have one or two things to clean up. For example, the *Hard-Coded Test Data* for the unit price, quantity, and customer's discount is repeated twice in the test. We can clarify the meaning of these numbers by using a Replace Magic Number with Symbolic Constant [Fowler] refactoring to give them role-describing names. Also, the constructor we are using to create the `LineItem` is not used anywhere in the SUT itself because the `LineItem` normally calculates the `extendedCost` when it is constructed. We should turn this test-specific code into a Foreign Method [Fowler] implemented within the test harness. We have already seen examples of how to do so with the `Customer` and `Product`: We use a *Parameterized Creation Method* (see *Creation Method*) to return the expected `LineItem` based on only those values of interest.

```
public void testAddItemQuantity_severalQuantity_v13(){
   final int QUANTITY = 5;
   final BigDecimal UNIT_PRICE = new BigDecimal("19.99");
   final BigDecimal CUST_DISCOUNT_PC = new BigDecimal("30");
```

```
      //   Set up fixture
      Customer customer = createACustomer(CUST_DISCOUNT_PC);
      Product product = createAProduct( UNIT_PRICE);
      Invoice invoice = createInvoice(customer);
      // Exercise SUT
      invoice.addItemQuantity(product, QUANTITY);
      // Verify outcome
      final BigDecimal EXTENDED_PRICE = new BigDecimal("69.96");
      LineItem expected =
         new LineItem(invoice, product, QUANTITY,
                  CUST_DISCOUNT_PC, EXTENDED_PRICE);
      assertContainsExactlyOneLineItem(invoice, expected);
   }
```

One final point: Where did the value "69.96" come from? If this value comes from the output of some reference system, we should say so. Because it was just manually calculated and typed into the test, we can show the calculation in the test for the **test reader's** benefit.

The Cleaned-Up Test

Here is the final cleaned-up version of the test:

```
   public void testAddItemQuantity_severalQuantity_v14(){
      final int QUANTITY = 5;
      final BigDecimal UNIT_PRICE = new BigDecimal("19.99");
      final BigDecimal CUST_DISCOUNT_PC =  new BigDecimal("30");
      // Set up fixture
      Customer customer = createACustomer(CUST_DISCOUNT_PC);
      Product product = createAProduct( UNIT_PRICE);
      Invoice invoice = createInvoice(customer);
      // Exercise SUT
      invoice.addItemQuantity(product, QUANTITY);
      // Verify outcome
      final BigDecimal BASE_PRICE =
         UNIT_PRICE.multiply(new BigDecimal(QUANTITY));
      final BigDecimal EXTENDED_PRICE =
         BASE_PRICE.subtract(BASE_PRICE.multiply(
               CUST_DISCOUNT_PC.movePointLeft(2)));
      LineItem expected =
         createLineItem(QUANTITY, CUST_DISCOUNT_PC,
                     EXTENDED_PRICE, product, invoice);
      assertContainsExactlyOneLineItem(invoice, expected);
   }
```

We have used an Introduce Explaining Variable [Fowler] refactoring to better document the calculation of the `BASE_PRICE` (price*quantity) and `EXTENDED_PRICE` (the price with discount). The revised test is now much smaller and clearer than the bulky code we started with. It fulfills the role of *Tests as Documentation* (see page 23) very well. So what did we discover that this test verifies? It confirms that the line items added to an invoice are, indeed, added to the invoice and that the extended cost is based on the product price, the customer's discount, and the quantity ordered.

Writing More Tests

It seemed like we went to a lot of effort to refactor this test to make it clearer. Will we have to spend so much effort on every test?

I should hope not! Much of the effort here related to the discovery of which *Test Utility Methods* (page 599) were required for writing the test. We defined a *Higher-Level Language* (see page 41) for testing our application. Once we have those methods in place, writing other tests becomes much simpler. For example, if we want to write a test that verifies that the extended cost is recalculated when we change the quantity of a `LineItem`, we can reuse most of the *Test Utility Methods*.

```
public void testAddLineItem_quantityOne(){
   final BigDecimal BASE_PRICE = UNIT_PRICE;
   final BigDecimal EXTENDED_PRICE = BASE_PRICE;
   //   Set up fixture
   Customer customer = createACustomer(NO_CUST_DISCOUNT);
   Invoice invoice = createInvoice(customer);
   //   Exercise SUT
   invoice.addItemQuantity(PRODUCT, QUAN_ONE);
   // Verify outcome
   LineItem expected =
      createLineItem( QUAN_ONE, NO_CUST_DISCOUNT,
                      EXTENDED_PRICE, PRODUCT, invoice);
   assertContainsExactlyOneLineItem( invoice, expected );
}

public void testChangeQuantity_severalQuantity(){
   final int ORIGINAL_QUANTITY = 3;
   final int NEW_QUANTITY = 5;
   final BigDecimal BASE_PRICE =
      UNIT_PRICE.multiply(   new BigDecimal(NEW_QUANTITY));
   final BigDecimal EXTENDED_PRICE =
      BASE_PRICE.subtract(BASE_PRICE.multiply(
                 CUST_DISCOUNT_PC.movePointLeft(2)));
   //   Set up fixture
   Customer customer = createACustomer(CUST_DISCOUNT_PC);
   Invoice invoice = createInvoice(customer);
   Product product = createAProduct( UNIT_PRICE);
   invoice.addItemQuantity(product, ORIGINAL_QUANTITY);
```

```
    // Exercise SUT
    invoice.changeQuantityForProduct(product, NEW_QUANTITY);
    // Verify outcome
    LineItem expected = createLineItem( NEW_QUANTITY,
        CUST_DISCOUNT_PC, EXTENDED_PRICE, PRODUCT, invoice);
    assertContainsExactlyOneLineItem( invoice, expected );
 }
```

This test was written in about two minutes and did not require adding any new *Test Utility Methods*. Contrast that with how long it would have taken to write a completely new test in the original style. And the effort saved in writing the tests is just part of the equation—we also need to consider the effort we saved understanding existing tests each time we need to revisit them. Over the course of a development project and the subsequent maintenance activity, this cost savings will really add up.

Further Compaction

Writing these additional tests revealed a few more sources of *Test Code Duplication* (page 213). For example, it seems that we always create both a `Customer` and an `Invoice`. Why not combine these two lines? Similarly, we continually define and initialize the `QUANTITY` and `CUSTOMER_DISCOUNT_PC` constants inside our test methods. Why can't we do these tasks just once? The `Product` does not seem to play any roles in these tests; we always create it exactly the same way. Can we factor this responsibility out, too? Certainly! We just apply an Extract Method refactoring to each set of duplicated code to create more powerful *Creation Methods*.

```
 public void testAddItemQuantity_severalQuantity_v15(){
    // Set up fixture
    Invoice invoice = createCustomerInvoice(CUST_DISCOUNT_PC);
    // Exercise SUT
    invoice.addItemQuantity(PRODUCT, SEVERAL);
    // Verify outcome
    final BigDecimal BASE_PRICE =
       UNIT_PRICE.multiply(new BigDecimal(SEVERAL));
    final BigDecimal EXTENDED_PRICE =
       BASE_PRICE.subtract(BASE_PRICE.multiply(
             CUST_DISCOUNT_PC.movePointLeft(2)));
    LineItem expected = createLineItem( SEVERAL,
        CUST_DISCOUNT_PC, EXTENDED_PRICE, PRODUCT, invoice);
    assertContainsExactlyOneLineItem(invoice, expected);
 }

 public void testAddLineItem_quantityOne_v2(){
    final BigDecimal BASE_PRICE = UNIT_PRICE;
    final BigDecimal EXTENDED_PRICE = BASE_PRICE;
```

```
      //   Set up fixture
      Invoice invoice = createCustomerInvoice(NO_CUST_DISCOUNT);
      //   Exercise SUT
      invoice.addItemQuantity(PRODUCT, QUAN_ONE);
      // Verify outcome
      LineItem expected = createLineItem( SEVERAL,
           CUST_DISCOUNT_PC, EXTENDED_PRICE, PRODUCT, invoice);
      assertContainsExactlyOneLineItem( invoice, expected );
   }

   public void testChangeQuantity_severalQuantity_V2(){
      final int NEW_QUANTITY = SEVERAL + 2;
      final BigDecimal BASE_PRICE =
         UNIT_PRICE.multiply(   new BigDecimal(NEW_QUANTITY));
      final BigDecimal EXTENDED_PRICE =
         BASE_PRICE.subtract(BASE_PRICE.multiply(
                     CUST_DISCOUNT_PC.movePointLeft(2)));
      //   Set up fixture
      Invoice invoice = createCustomerInvoice(CUST_DISCOUNT_PC);
      invoice.addItemQuantity(PRODUCT, SEVERAL);
      // Exercise SUT
      invoice.changeQuantityForProduct(PRODUCT, NEW_QUANTITY);
      // Verify outcome
      LineItem expected = createLineItem( NEW_QUANTITY,
            CUST_DISCOUNT_PC, EXTENDED_PRICE, PRODUCT, invoice);
      assertContainsExactlyOneLineItem( invoice, expected );
   }
```

We have now reduced the number of lines of code we need to understand from 35 statements in the original test to just 6 statements.[4] We are left with just a bit more than one sixth of the original code to maintain! We could go further by factoring out the fixture setup into a `setUp` method, but that effort would be worthwhile only if a lot of tests needed the same Customer/Discount/Invoice configuration. If we wanted to reuse these *Test Utility Methods* from other *Testcase Classes* (page 373), we could use an Extract Superclass [Fowler] refactoring to create a *Testcase Superclass* (page 638), and then use a Pull Up Method [Fowler] refactoring to move the *Test Utility Methods* to it so they can be reused.

[4] Ignoring wrapped lines, we have 6 executable statements surrounded by the two lines of method declarations/end.