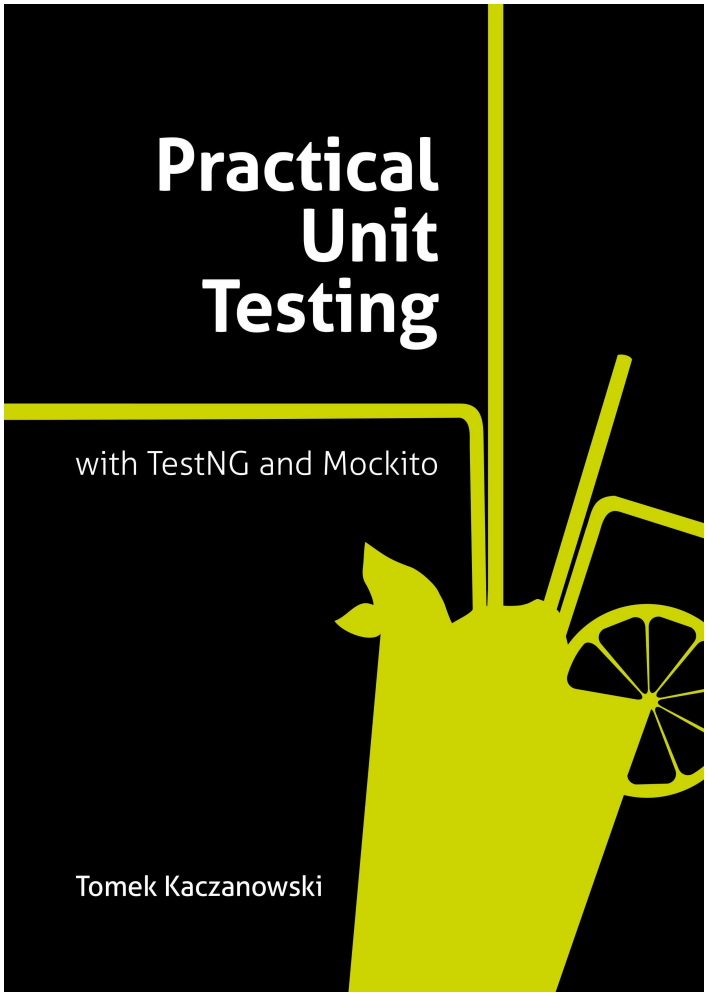

http://practicalunittesting.com

Sample chapter

Chapter 10: Maintainable Tests

Chapter 10. Maintainable Tests

Applications maintainability - the holy grail of software development! We write code every day, trying to make it so good it will withstand the test of time. We hope that we, or our colleagues, working with this code sometime in the future, will be able to understand it at a glance. We hope to be able to introduce changes easily without causing chaos throughout the entire application.

We should write our tests with the same attitude, trying to make them maintainable. Why? Because, as we have already discussed, they play a crucial role in supporting and documenting our production code. In this section we will discuss various aspects of the maintainability of tests.

10.1. Test Behaviour, not Methods

The rule we will be discussing in this section is very simple: *"Test behaviour, not methods!"*. This means that when writing tests, we should think about the SUT in terms of its responsibilities - in terms of the contract it has with its client. We should abstract from the SUT's implementation, which is of only secondary importance. What matters is that the SUT should fulfill the requirements for which it was designed. And to make sure it really does, we should write these requirements in the form of test cases. The requirements know nothing about the actual implementation, and neither should our tests.

This may seem trivial, but unfortunately I frequently see this simple rule being violated, which leads me to be think that it is, after all, worth discussing.

Below, in Listing 10.1, an example of a suboptimal test of the `BankAccount` class is presented. Each test method attempts to test a single method of the public API of `BankAccount`: `getBalance()`, `deposit()` and `withdraw()`.

In order to better present the main issue of this section, I have decided to keep all tests truncated to a very limited number of test cases. In reality, I would use many more test cases, probably employing parametrized tests (see Section 3.6).

Listing 10.1. One test method per one production code method

```
@Test
public class BankAccountTest {

    private BankAccount account;

    @BeforeMethod
    public void setUp() {
        account = new BankAccount();
    }

    public void testBalance() { ❶
        account.deposit(200);
        assertEquals(account.getBalance(), 200);
    }

    public void testDeposit() { ❷
        account.deposit(100);
        assertEquals(account.getBalance(), 100);
        account.deposit(100);
        assertEquals(account.getBalance(), 200);
    }

    public void testWithdraw() { ❸
        account.deposit(100);
        account.withdraw(30);
        assertEquals(account.getBalance(), 70);
        account.withdraw(20);
        assertEquals(account.getBalance(), 50);
    }
}
```

❶ Test for the `getBalance()` method. Note that it also uses a `deposit()` method.
❷ Test for the `deposit()` method. It also calls `getBalance()`.
❸ Test for the `withdraw()` method, which likewise also calls `getBalance()`.

As Listing 10.1 shows, isolation is not possible in unit tests at the level of methods. Each test method calls various methods of the SUT - not only the one they have pretensions to testing. It has to be like that, because you really cannot test the `deposit()` method without checking the account's balance (using the `getBalance()` method).

There are also some other issues with this approach. Let us list them:

- If any of the test methods should fail, then an error message (e.g. *"testDeposit has failed"*) will not be informative enough for us to instantly understand which of the SUT's requirements has not been fulfilled (where this is really important from the client's point of view).

- Each of the SUT's methods is involved in multiple user-stories, so it is very hard to keep a *"one test method per production code method"* pattern. For example, how might we add a test to the existing code, which would verify that after creation an account has a balance of zero? We could enhance the `testBalance()` method with an additional assertion, but that would make it prone to fail for more than one reason. Which is not good, and leads to confusion when the test does fail.

- Test methods tend to grow as the SUT is enhanced to reflect new requirements.

- Sometimes it is hard to decide which of the SUT's methods is really being tested in a certain scenario (because more than one is being used).

- Test methods overlap with each other - e.g. `testBalance()` is a repetition of what will be tested by `testDeposit()` and `testWithdraw()`. In fact, it is hard to say why `testBalance()` is there at all - probably because a developer felt she/he *"needed to have a test for the `getBalance()` method"*.

When I see test code like this, I know for sure that it was written after the SUT had already been implemented. The structure of the test reflects the structure (implementation) of the SUT code, which is a clear sign of this approach. From what I have observed, such tests rarely cover everything required of the SUT. They check what obviously needs to be checked, given the SUT's implementation, but do not try to test anything more (thus avoiding solving some of the dilemmas listed above).

What is interesting is that this test is good enough to achieve 100% code coverage of a valid implementation of the `BankAccount` class. This is one more reason **not** to trust the code coverage (see also Section 11.3).

Is there a better approach? Yes, and - what is really nice - it does not require any additional work. It only requires us to concentrate on the SUT's behaviour (which reflects its responsibilities) and write it down in the form of tests.

An example of this approach is shown in the two listings below. As can be seen, some of its methods are identical to the previous approach, but the test as a whole has been created with a completely different mindset, and it covers a broader set of the SUT's responsibilities.

Listing 10.2. Testing behaviour, not implementation

```
@Test
public class BankAccountTest {

    private BankAccount account;

    @BeforeMethod
    public void setUp() {
        account = new BankAccount();
    }

    public void shouldBeEmptyAfterCreation() { ❶
        assertEquals(account.getBalance(), 0);
    }

    public void shouldAllowToDepositMoney() { ❷
        account.deposit(100);
        assertEquals(account.getBalance(), 100);
        account.deposit(100);
        assertEquals(account.getBalance(), 200);
    }

    public void shouldAllowToWithdrawMoney() { ❸
        account.deposit(100);
        account.withdraw(30);
        assertEquals(account.getBalance(), 70);
        account.withdraw(20);
        assertEquals(account.getBalance(), 50);
    }

...
```

❶ There is no test for the `getBalance()` method, because its proper functioning is validated by other tests.

❷ This is identical to the previous `testDeposit()` method, with the exception of the method name, which is much more informative.
❸ As above - identical to the `testWithdraw()` method, but better named.

Listing 10.3. Testing behaviour, not implementation

```
...

    @Test(expectedExceptions = NotEnoughMoneyException.class)
    public void shouldNotAllowToWithdrawFromEmptyAccount() { ❶
        // implementation omitted
    }

    @Test(expectedExceptions = InvalidAmountException.class)
    public void shouldNotAllowToUseNegativeAmountForWithdraw() { ❷
        // implementation omitted
    }

    @Test(expectedExceptions = InvalidAmountException.class)
    public void shouldNotAllowToUseNegativeAmountForDeposit() { ❸
        // implementation omitted
    }
}
```

❶❷❸ New methods added. This was possible, because the developer was thinking in terms of the SUT's responsibility.

The two versions of the `BankAccountTest` test class differ substantially when it comes to test methods naming. Good test method names include information about the scenario they verify. This topic is discussed in detail in Section 9.3.2.

Table 10.1 compares what was tested and how, with both approaches.

Table 10.1. Comparison of two approaches to testing

use-case scenario	testing implementation	testing behaviour
when opening a new account, its balance should be zero	oops, forgot about this one!	`shouldBeEmptyAfterCreation()`
it is possible to credit an account	`testDeposit()` and `testBalance()`	`shouldAllowToDepositMoney()`
it is possible to debit an account	`testWithdraw()`	`shouldAllowToWithdrawMoney()`
should not allow for accounts misuse	oops, forgot about these too!	`shouldNotAllowToWitrdrawFromEmptyAccount()`, `shouldNotAllowToUseNegativeAmountForDeposit()` and `shouldNotAllowToUseNegativeAmountForWithdraw()`

One might be tempted to claim that this is just a single example, and a biased one at that. Well actually, no. I have witnessed this far too many times to have any doubts about it being how things are: **when testing implementation only a subset of scenarios is being verified**, test methods are overlapping, and are prone to grow to include all possible scenarios for each test method. The key is to think about

test methods as about mini user stories: each of them should ensure that some functionality important from the client's point of view is working properly.

So, as a rule of thumb, forget about implementation. Think about requirements. TDD might make it easier for you to code like this.

Some IDEs offer "a feature" which generates test methods based on production code (so if your class has a `doSomething()` method, the tool will generate a `testDoSomething()` method). This can lead you down the wrong path - that of methods testing rather than class responsibilities testing. Avoid such solutions. Stay on the safe side by following the test-first approach.

10.2. Complexity Leads to Bugs

> Controlling complexity is the essence of computer programming.
>
> — Brian Kernighan

Do not put any complexity into your tests! No `if` structure, no `switch` statements, no decision making. Otherwise you risk finding yourself in a situation where the results of tests are influenced by two factors at the same time: the quality of the logic of production code and the quality of the logic of test code. This is one too many.

If any test fails, you need to discover where the bug is - in the production code or the test code. A worse thing can also happen - it is possible that tests pass thanks to errors in test code unintentionally rectifying errors in production code. (Yes, two wrongs sometimes make a right!) This is a serious danger.

Another thing you lose out on by putting logic inside your test code is that it can no longer serve as documentation. Who wants to read documentation that requires the solving of logical puzzles?

Remember, what you are supposed to be doing is testing the correctness of production code. Do not make it any harder than necessary.

10.3. Follow the Rules or Suffer

> Procedural code gets information, then makes decisions. Object-oriented code tells objects to do things.
>
> — Alec Sharp

> Daughter, do not talk with strangers!
>
> — Demeter *Ancient Greece (700 BC)*

The two quotes which open this section refer to two famous principles of good design, both of them notorious for being breached: *"Tell, Don't Ask!"*[1] and *"Law of Demeter"*[2]. The first one states that the object should ask others to do whatever it wants, rather than doing the job based on the data they are willing to provide. The second principle dictates with whom the object is allowed to talk.

This section gives an example of what happens when you break these two rules.

[1] See http://pragprog.com/articles/tell-dont-ask for details.
[2] See http://en.wikipedia.org/wiki/Law_of_Demeter for a more detailed description.

10.3.1. Real Life is Object-Oriented

Imagine you get in a taxi. *"To the airport, please!"*, you say, and the driver nods his head. Now, you want to know how long it will take to get there. What question would you rather ask:

1. How long will it take?
2. Please tell me (so I can do the maths myself):
   a. How far is the airport?
   b. What is your average speed travelling to the airport from here?

I have never heard of anyone who used the second approach. In real life we act quite smartly, asking people who know (or at least should do) and only caring about the result (i.e. leaving the boring details to them). So why on earth do we write code that follows the second approach? And we really do! I see this happening all the time.

Let us have a look at an example from the domain of finance[3] in order to illustrate the difference between these two approaches. The example is as follows. There is a function that calculates the value of all assets of a client. It takes a collection of funds as a parameter and returns a single number as an output. Each fund consists of two registers. A client has a number of entities within each register.

10.3.2. The Non-Object-Oriented Approach

A possible implementation of a `Client` class is shown in Listing 10.4. Some details have been omitted, so we can concentrate on the crucial part: calculating the value of a client's assets.

Listing 10.4. Client class written using a non-object-oriented approach

```
public class Client {

    private final List<IFund> funds;

    ...

    public BigDecimal getValueOfAllFunds() {
        BigDecimal value = BigDecimal.ZERO;
        for (IFund f : funds) {
            value = value.add(f.getCurrentValue().getValue().multiply(
                new BigDecimal(
                    f.getRegisterX().getNbOfUnits()
                    + f.getRegisterY().getNbOfUnits()
                )
            ));
        }
        return value;
    }
}
```

As shown in Listing 10.4, a client has to do some complex calculations in order to obtain the result. For each fund it needs to:

[3]The example is only a slightly modified version of a real business domain problem and real code that once got implemented as part of some long-forgotten project.

- get the current fund value (`f.getCurrentValue().getValue()`), which is a two-step process, because `IFund` returns `ICurrentValue` object, which contains the real value,
- multiply this value by the number of units in both registers.

Then, the results for all funds must be added together to obtain the final amount.

If you are seriously into object-oriented programming, you will surely have noticed that the code in Listing 10.4 breaches both of the principles mentioned at the beginning of this section:

- *"Tell, Don't Ask!"* has been broken, because `Client` asks for data instead of telling others to give him results,
- *"Law of Demeter"* has been broken, because `Client` talks with friends of his friends (i.e. with registers and current value, both of which are accessed as friends of funds).

This makes it obvious that we are in trouble. The client seems to know everything about everything, when in fact all they should be interested in is the value of each fund they own. The details of the internal structure of funds should be completely hidden from them, but are not. Based on this observation, we can say that the types used in this example have a serious problem with information hiding[4]: they reveal their internal design. This goes against the norms of good practice in programming, and will cause problems when the code needs to be changed.

…but the **main problem** with such code is… that it works! The results obtained are correct. This code really calculates what it should. This leads people to conclude that the code itself is also correct. The widespread *"If it works, don't fix it!"* approach[5] results in such code being left as it is. The problems come later - usually when the code should be changed. This is when the troubles begin.

So, right now we will attempt to test it. There are many test cases that should be verified (with different combinations of number of funds and values), but for our purposes it will suffice to choose just one: a client having two funds. This does not sound like a difficult task, does it? Well, let us take a closer look.

> Okay, so here is what I will need for my test: two test doubles of the `IFund` type, each of them having a value; so two `ICurrentValue` test doubles will also be required. Each fund also has two registers, so another four test doubles will be required (of the `IRegister` type). And it seems like all of these test doubles will be stubs. I only need them because I want them to return some canned values. Anything else? No, these are the main points. So let us get started.
>
> — Tomek *Thinking Aloud about How to Test Non-Object-Oriented Code*

The listing is divided into two parts, so it renders better.

[4] See http://en.wikipedia.org/wiki/Information_hiding

[5] Please consult [martin2008] for a different approach - the **Boy Scout Rule** rule: *"Leave the campground cleaner than you found it."*.

Listing 10.5. Test of the non-object-oriented Client class - setup

```
public class ClientTest {

    private int NB_OF_UNITS_AX = 5; ❶
    private int NB_OF_UNITS_AY = 1;
    private int NB_OF_UNITS_BX = 4;
    private int NB_OF_UNITS_BY = 1;
    private BigDecimal FUND_A_VALUE = new BigDecimal(3);
    private BigDecimal FUND_B_VALUE = new BigDecimal(2);

    @Test
    public void totalValueShouldBeEqualToSumOfAllFundsValues() {
        Client client = new Client(); ❷
        IFund fundA = mock(IFund.class); ❸
        IFund fundB = mock(IFund.class);
        IRegister regAX = mock(IRegister.class);
        IRegister regAY = mock(IRegister.class);
        IRegister regBX = mock(IRegister.class);
        IRegister regBY = mock(IRegister.class);
        ICurrentValue currentValueA = mock(ICurrentValue.class);
        ICurrentValue currentValueB = mock(ICurrentValue.class);

        ...
```

❶ Some primitive values that are also required for this test.
❷ A client: our SUT.
❸ The SUT's collaborators - direct and indirect.

Listing 10.6. Test of the non-object-oriented Client class - actual tests

```
        ...

        when(fundA.getRegisterX()).thenReturn(regAX); ❶
        when(fundA.getRegisterY()).thenReturn(regAY);
        when(fundB.getRegisterX()).thenReturn(regBX);
        when(fundB.getRegisterY()).thenReturn(regBY);
        when(regAX.getNbOfUnits()).thenReturn(NB_OF_UNITS_AX);
        when(regAY.getNbOfUnits()).thenReturn(NB_OF_UNITS_AY);
        when(regBX.getNbOfUnits()).thenReturn(NB_OF_UNITS_BX);
        when(regBY.getNbOfUnits()).thenReturn(NB_OF_UNITS_BY);

        when(fundA.getCurrentValue()).thenReturn(currentValueA); ❷
        when(fundB.getCurrentValue()).thenReturn(currentValueB);
        when(currentValueA.getValue()).thenReturn(FUND_A_VALUE);
        when(currentValueB.getValue()).thenReturn(FUND_B_VALUE);

        client.addFund(fundA); ❸
        client.addFund(fundB);

        assertEquals(client.getValueOfAllFunds(),
            BigDecimal.valueOf((5 + 1) * 3 + (4 + 1) * 2)); ❹
    }
}
```

❶ Instructing stubs on what they should return.
❷ Hmm, interesting - instructing a stub to return a stub…
❸ Setting the SUT in the desired state - it should own two funds.
❹ Verification.

This test is very long, and there are some really disturbing and confusing features to it:

- the test class knows all about the internalities of funds and registers,
  - the algorithm of calculation,
  - the internalities of all types involved in calculations (e.g. that a register has units),
- a number of test doubles are required for this test,
- the test methods consist mostly of instructions for stubs concerning the values they should return,
- stubs are returning stubs.

All this makes our test **hard to understand and maintain**, and also **fragile** (it needs to be rewritten every time we change anything in the funds value calculation algorithm).

And now some really **bad news**: we would need more than one test like this. We need a test for 0 funds, for 1 fund, and for 7 funds (when the marketing guys come up with a brilliant idea of some extra bonus for people who have invested in more than 6 funds), and all this multiplied by various values of funds. Uh, that would hurt really bad.

Do We Need Mocks?

In the example as presented so far, we have used test doubles for all collaborators of the `Client` class. In fact, a few lines of test code could have been spared, if we had used real objects instead of classes. True, but on the other hand:

- as discussed in Section 5.5, this would be no more than a short-term solution,
- in real life, the values of funds might be fetched from some external source (e.g. a web service), which would make it much harder to test.

Because of this, replacing all collaborators with test doubles seems a valid choice.

10.3.3. The Object-Oriented Approach

Mentally scarred - no doubt - by what we have just witnessed, let us now try out a different implementation of the `Client` class, and compare the effort required to test it with that involved in the previous example. This time we shall make our `Client` more object-oriented.

Listing 10.7. The Client class - object-oriented version

```
public class Client {

    private final List<IFund> funds;

    ....

    public BigDecimal getValueOfAllFunds() {
        BigDecimal value = BigDecimal.ZERO;
        for (IFund f : funds) {
            value = value.add(f.getValue()); ❶
        }
        return value;
    }
}
```

❶ This time all calculation of fund value is encapsulated within a `getValue()` method of the `IFund` type. All the client does is add up the results.

Writing a test for such a class is straightforward - we need only two stubs for this (one per each fund that the client has, and we have decided in advance that for the first test, the client will have two funds)

Listing 10.8. Test of the object-oriented Client class

```
public class ClientTest {

    private final static BigDecimal VALUE_A = new BigDecimal(9);
    private final static BigDecimal VALUE_B =  new BigDecimal(2);

    public void totalValueShouldBeEqualToSumOfAllFundsValues() {
        Client client = new Client();
        IFund fundA = mock(IFund.class);
        IFund fundB = mock(IFund.class);

        when(fundA.getValue()).thenReturn(VALUE_A);
        when(fundB.getValue()).thenReturn(VALUE_B);

        client.addFund(fundA);
        client.addFund(fundB);

        assertEquals(client.getValueOfAllFunds(), VALUE_A.add(VALUE_B));
    }
}
```

Wow, this differs substantially from what we were seeing before. The test is concise and does not contain any information on the internalities of funds. Pursuing this object-oriented approach further, we would have to write tests for each and every class (e.g. we need a test for implementation of the `IFund` interface, and also for the `IRegister` interface), but all of them would be very, very simple indeed. Each of these tests would also depend only on the SUT. No information about other classes would be used within the test code. This is very different from what we saw in Listing 10.5.

Coming back to the question we asked when discussing a non-object-oriented version of this test, would it be hard to write tests for 0, 1 and 7 funds? This time the answer is *no*. It would not be.

10.3.4. How To Deal with Procedural Code?

We have just witnessed the (disastrous) impact that procedural code can have on testing. If your code does not adhere to basic rules of object-oriented design, it will be hard to test. Now, let us discuss what is the right way to deal with such code.

As usual, the best thing you can do is to **act before the damage has been done**. Do not let procedural code creep into your codebase! TDD seems to be very good at deterring procedural code. As discussed previously, it is very painful to write tests for such code. If you start out with the tests themselves, you will definitely end up coming up with solutions that are more object-oriented (and less procedural).

The above advice will not ne of much use, though, if you have just inherited 100k lines of procedural code. There are techniques that can help you deal with such an unfortunate situation, but the topic goes beyond the scope of this book. Please refer to the excellent work of [feathers2004] for guidance.

10.3.5. Conclusions

As the code examples within this section have demonstrated, **bad code makes it hard to write tests**. Allow me to back up this claim with two quotes, illustrating the most important points connected with what we have just been discussing.

> Consistency. It is only a virtue, if you are not a screwup.
>
> — Wisdom of the Internet ;)

The misery begins with a single, innocent-seeming line such as *"ask object `x` for the value of `y` (`x.getY()`) and make some decisions based on the value of `y`"*. If you encounter code which breaches the *"Tell, Don't Ask!"* principle, then do not copy and paste it into your code. What you should do, instead, is clean it, usually by adding methods in places where they ought to be[6]. Then proceed - writing clean, well-designed code.

Do not copy other sloppy work! Do not become one of the blind led by the blind! An abyss awaits you if you do. (Wow, that has really got you scared, hasn't it?)

> Every time a mock returns a mock, a fairy dies.
>
> — Twitter @damianguy *2009 Oct 19*

When writing a test requires you to have a test double which returns another test double, then you know you are about to do something very bad indeed. Such a situation indicates that the code you are working with contravenes *"Law of Demeter"*, which is really most regrettable. Repair the code, and only then get down to testing it. After all…you do not want fairies to die, do you?

10.4. Rewriting Tests when the Code Changes

A change in the requirements occurs. Developers analyze it and implement the required changes. Then tests are run and some of them fail. You can see the disappointment written all over the developers' faces when they sit down to *"fix these *(&(#$ failed tests!"*.

Have you ever witnessed such a scenario? Have you ever had the feeling that your tests are a major nuisance, and that their existence makes the process of introducing changes a good deal longer and harder than it would be without them? Well, I have certainly seen this many times, and have personally become angry at the fact that after having performed some updates of production code I also had to take care of the tests (instead of moving on to another task).

There are two explanations of why this situation is so common. The first relates to the quality of your tests, the second to the code-first approach.

Let us agree on something, before we begin. If you rewrite part of your implementation, then **it is normal that some of your tests will start to fail**. In fact, in the majority of cases this is even desirable: if no tests fail, then it means your tests were not good enough![7] The real problems arise if:

[6] If you need more information about this, please read about the "Feature Envy" code smell.
[7] This is exactly the behaviour that mutation testing takes advantage of; see Section 11.4.

- the change which made the tests fail is really a refactoring - it does not influence the observable external behaviour of the SUT,
- the failed tests do not seem to have anything to do with the functionality that has changed,
- a single change results in many tests failing.

The last of the above highlights the fact of there being some duplication in respect of tests – with the result that multiple tests are verifying the same functionality. This is rather simple to spot and fix. The other two issues are more interesting, and will be discussed below.

10.4.1. Avoid Overspecified Tests

> The most important rule of thumb we follow to keep our tests flexible is: Specify exactly what you want to happen and no more.
>
> — JMock tutorial

What is an overspecified test? There is no consensus about this, and many examples that can be found describe very different features of tests. For the sake of this discussion, let us accept a very simple "definition": **a test is overspecified if it verifies some aspects which are irrelevant to the scenario being tested**.

Now, which parts of the tests are relevant and which are not? How can we distinguish them just by looking at the test code?

Well, good test method names are certainly very helpful in this respect. For example, if we analyze the test in the listing below, we find that it is a little bit overspecified.

Listing 10.9. Overspecified test - superfluous verification

```
@Test
public void itemsAvailableIfTheyAreInStore() {
    when(store.itemsLeft(ITEM_NAME)).thenReturn(2); ❶

    assertTrue(shop.isAvailable(ITEM_NAME)); ❷

    verify(store).itemsLeft(ITEM_NAME); ❸
}
```

❶ stubbing of a DOC,
❷ asserting on the SUT's functionality,
❸ verifying the DOC's behaviour.

If this test truly sets out to verify that *"items are available if they are in store"* (as the test method name claims), then what is the last verification doing? Does it really help to achieve the goal of the test? Not really. If this cooperation with the store collaborator is really a valuable feature of the SUT (is it?), then maybe it would be more appropriate to have a second test to verify it:

Listing 10.10. Two better-focused tests

```
@Test
public void itemsAvailableIfTheyAreInStore() {
    when(store.itemsLeft(ITEM_NAME)).thenReturn(2);

    assertTrue(shop.isAvailable(ITEM_NAME));
}

@Test
public void shouldCheckStoreForItems() {
    shop.isAvailable(ITEM_NAME);

    verify(store).itemsLeft(ITEM_NAME);
}
```

Each of the tests in Listing 10.10 has only one reason to fail, while the previous version (in Listing 10.9) has two. The tests are no longer overspecified. If we refactor the SUT's implementation, it may turn out that only one fails, thus making it clear which functionality was broken.

This example shows the importance of good naming. It is very hard to decide which part of the `testShop()` method is not relevant to the test's principal goal.

Another test-double based example is the use of **specific parameter values** (`"my item"`, `7` or `new Date(x,y,z)`) when something more generic would suffice (`anyString()`, `anyInt()`, `anyDate()`)[8]. Again, the question we should ask is whether these specific values are really important for the test case in hand. If not, let us use more relaxed values.

Also, you might be tempted to test very defensively, to **verify that some interactions have not happened**. Sometimes this makes sense. For example in Section 5.4.3 we verified whether no messages had been sent to some collaborators. And such a test was fine - it made sure that the unsubscribe feature worked fine. However, do not put such verifications in when they are not necessary. You could guard each and every one of the SUT's collaborators with verifications that none of their methods have been called[9], but do not do so, unless they are important relative to the given scenario. Likewise, checking whether certain calls to collaborators happened in the order requested (using Mockito's `inOrder()` method) will usually just amount to overkill.

We can find numerous examples of overspecified tests outside of the interactions testing domain, as well. A common case is to expect a certain exact form of text, where what is in fact important is only that it should contain several statements. Like with the example discussed above, it is usually possible to divide such tests into two smaller, more focused ones. For example, the first test could check whether a message that has been created contains the user's name and address, while the second one might perform a full text-matching. This is also an example of when test dependencies make sense: there is no point in bothering with an exact message comparison (which is what the second test verifies), if you know that it does not contain any vital information (verified by the first test).

Based on what we have learned so far, we can say that a good rule of thumb for writing decent, focused tests is as follows: **test only the minimally necessary set of features using each test method**.

[8] See Section 6.6 for discussion and more examples.
[9] Mockito provides some interesting functions for this - `verifyZeroInteractions()` and `verifyNoMoreInteractions()`.

As is by no means unusual where problems connected with tests are concerned, the real culprit may be the production code. If your test really needs to repeat the petty details of the SUT's implementation (which will certainly lead to it being overspecified), then maybe the problem lies with how the SUT works with its collaborators. Does the SUT respect the *"Tell-Don't-Ask!"* principle?

10.4.2. Are You Really Coding Test-First?

So the change request came. A developer updated the production code, and then also worked on the failed tests which stopped working because of the implemented change. Wait! What? By implementing changes in production code first, we have just reverted to code-first development, with all its issues! The price that we pay is that now we will have to rewrite some tests looking at the code we wrote a few minutes ago. But this is boring: such tests will probably not find any bugs, and they themselves will most probably be very closely linked to the implementation of the production code (as was already discussed in Chapter 4, *Test Driven Development*).

Much better results (and less frustration for developers) can be achieved by trying to mimic the TDD approach, following the order of actions given below:

- requirements change,
- developers analyze which tests should be updated to reflect the new requirements,
- tests are updated (and fail because code does not meet the new requirements),
- developers analyze what changes should be introduced into the production code,
- code is updated and tests pass.

This is somewhat different from the TDD approach as we have observed it so far. If we write a new functionality, then we ensure that each individual test that fails is dealt with at once. However, when the requirements of an existing functionality change, we may find ourselves forced to rewrite several tests at once, and then have to deal with all of them failing.

We may sum things up here by saying that in order to avoid having to fix tests after code changes (which is pretty annoying, let's face it), you should:

- write good tests (i.e. loosely coupled to implementation), minimizing the number of failed tests,
- use test-first in all phases of the development process - both when working on new features and when introducing changes to the existing codebase.

10.4.3. Conclusions

> The mime: Developing the code first and then repeating what the code does with expectations mocks. This makes the code drive the tests rather than the other way around. Usually leads to excessive setup and poorly named tests that are hard to see what they do.
>
> — James Carr

As with many other things related to quality, how you start makes a difference. If you start with production code, then your tests will (inevitably, as experience proves) contain too many implementation

details, and thus become fragile. They will start to fail every time you touch the production code. But you can also start from the other end: writing tests first or, rather, designing your production code using tests. Do that and your tests will not really be testing classes, so much as the functionalities embedded within them, and as such will have more chances of staying green when classes change.

Of course, it would be naive to expect that your tests can survive any changes to production code. We already know that many of our tests focus on interactions of objects (and to do so, use knowledge about the internal implementation of those objects), so such false hopes should be abandoned. The question remains, how many tests will be undermined by a single change in your production code, and how easy will it be to update them so they meet the altered requirements.

Probably the most important lesson we should remember is that **we should write tests which verify the expected outcome of systems behaviour, and not the behaviour itself**. If possible, let us verify that the system works properly by analyzing returned values. Only when this is not possible should we resort to interactions testing. As Gerard Meszaros puts it (see [meszaros2007]): *"use the front door"*.

The focus should be on the goal of the test. There is usually a single feature that is tested by each test. We should put aside anything that is not really essential to its verification. For example, by using stubs (whose behaviour is not verified) instead of mocks (which are verified) whenever possible. Another example is the use of more argument matchers - both in stubbing and verification (see Section 6.6).

And finally, now for some very obvious advice: your tests should be run very frequently. If not, then one day you will learn that 50% of them need to be rewritten. And then there will be nothing you can do - except wail in despair!

10.5. Things Too Simple To Break

> Yep, you should be unit testing every breakable line of code.
>
> — Bob Gregory

> It’s necessary to be very good at testing to decide correctly when you don’t need it, and then very good at programming to get away with it.
>
> — Twitter @RonJeffries *2012 Jan 31*

After reading about the benefits of developer tests, you should be tempted to test **everything**. Very good, this is the right attitude. If you code **test-first**, then you get high code coverage "for free". Everything is tested as a result of writing methods to satisfy a failed test. But if you follow the **code-first** approach, then you might quickly start questioning the idea of testing everything. In the case of some code parts, writing unit tests seems superfluous. This section is devoted to exactly these sorts of doubt or uncertainty.

Please note, that only a minority of methods are too simple to be considered *"unbreakable"*. Most of the code you write calls for decent tests!

Let us take the example of simple getter/setter methods, as shown in Listing 10.11.

Listing 10.11. Getters/Setters - too simple to break

```
public class User {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Yes, you definitely **can** write a test for this code - but please ask yourself: what kind of bugs, current or future, do you expect to catch by having such a test?

In my opinion there is no sense to writing tests for such code after the code has already been written. There are two reasons for this, which are as follows:

- there is no logic there worth testing,
- the code has probably been generated by the IDE (which then eliminates the threat of a silly copy&paste error).

However, if the getter and setter methods are to be changed, entailing that some complexity will be added (even of the simplest sort), then a test should be created. For example, if the `setName()` method evolves and takes care of validation, along the lines shown in Listing 10.12, then it surely should be tested.

Listing 10.12. Getters/Setters with validation - not so simple anymore

```
public void setName(String name) {
    if (name == null || name.isEmpty()) {
        throw new IllegalArgumentException();
    }
    this.name = name;
}
```

Many people argue that because of the possible future evolution of code (which is hard to predict when the first version is actually being written), you should write a test even for such trivial cases as the first version of the `setName()` method (the one without validation). I tend to disagree, and I would encourage you to refrain from writing such tests. On the other hand, once things get complicated it is crucial to write them. Then there is no excuse, and tests have to be written.

> It is true that adding tests for even these simple methods guards against the possibility that someone refactors and makes the methods "not-so-simple" anymore. In that case, though, the refactorer needs to be aware that the method is now complex enough to break, and should write tests for it - and preferably before the refactoring.
>
> — J.B. Raisenberg *JUnit FAQ*

However, none of this matters if you write code **test-first**. In that case, every method will be preceded with a case of a test failing. The complexity does not matter. If it exists, there must be a test for it. It does not necessarily mean that your test code will be full of trivial getter/setter tests. On the contrary, when your design is being guided by tests, you might well find yourself writing less getters and setters than

you used to. This is one of the benefits of allowing design to be driven by functionality requirements (expressed in the form of tests).

Returning to the **code-first** approach, let us take a look at another example, which shows a piece of code often included in the "too simple to break" category. Listing 10.13 shows a simple delegator - a method whose main task is to tell some other object to do the job.

Listing 10.13. Delegator - too simple to break

```
public class DelegatorExample {

    private Collaborator collaborator;

    public void delegate() {
        collaborator.doSomething();
    }
}
```

True, proper testing of such simple code does require some effort. If you are to use test doubles (which you probably should do), then the test will probably be longer, and even more complicated, than the tested method itself. This will definitely discourage us from writing unit tests - especially in cases where the benefits are not clearly visible. There is no easy answer to the question of whether you should write a test for such a method. It depends on (at least) three factors, namely:

- the type (i.e. specific features) of the `Collaborator` class,
- the complexity of the delegating method,
- the existence of other types of test.

Let us concentrate on these three factors, and run through a few comments that seem relevant to the issue:

- there is usually (if not always) something more involved than simply telling the collaborator to do the job. A delegating method will take some arguments and pass them to the collaborator, often performing some actions before it does so (validation of parameters, creation of some objects based on received parameters, etc.).
- the collaborator's `doSomething()` method will often return some values being used by the SUT in diverse ways,
- a collaborator's `doSomething()` method might throw exceptions, which will somehow have to be handled by the SUT,
- other types of test - e.g. integration tests - might cover this functionality. For example, an integration test might check if a class of service layer delegates tasks to a class of dao layer. However, it is rare for integration tests to cover all the possible scenarios (i.e. exceptions thrown by collaborators), so there might still be some gasps to be filled by unit tests.

My point is that the rather simple appearance of such delegating methods may be deceptive. There can be much more to them than meets the eye. By thinking about possible usage scenarios and interactions between the SUT and collaborators, you can reveal this hidden complexity and test it. But, as has already been said, every instance of such code can be considered individually, and there might be cases where writing a test is a waste of time.

10.6. Conclusions

Among the topics discussed in this chapter, there are two fundamental things that I would like you to remember. The first is the *"Test behaviour, not implementation!"* rule: if you stick to this, it will guide you towards high-quality testing (on every level, not only for unit tests). The second can be expressed by two rules that apply to production code, commonly known as the **Law of Demeter** and the **Tell, Don't Ask!** principle. Expect trouble when testing code that does not abide by either of them.

The rest of this chapter has been devoted to problems of logic within test code, the notion of "things that are too simple to break", and to the problem of test maintenance.

10.7. Exercises

10.7.1. A Car is a Sports Car if …

After three months of analysis a team of business analysts have decided that a car can be marked with the "sports" tag if it satisfies all of the following requirements:

- it is red,
- it was manufactured by Ferrari,
- its engine has more than 6 cylinders.

Based on these detailed requirements a team of top-notch developers have come up with the following implementation of the `CarSearch` class:

Listing 10.14. CarSearch class implementation

```
public class CarSearch {

    private List<Car> cars = new ArrayList<Car>();

    public void addCar(Car car) {
        cars.add(car);
    }

    public List<Car> findSportCars() {
        List<Car> sportCars = new ArrayList<Car>();
        for (Car car : cars) {
            if (car.getEngine().getNbOfCylinders() > 6
                    && Color.RED.equals(car.getColor())
                    && "Ferrari".equals(car.getManufacturer().getName())) {
                sportCars.add(car);
            }
        }
        return sportCars;
    }
}
```

The `Car`, `Engine` and `Manufacturer` interfaces are presented below:

Listing 10.15. Car interface

```
public interface Car {
    Engine getEngine();
    Color getColor();
    Manufacturer getManufacturer();
}
```

Listing 10.16. Engine interface

```
public interface Engine {
    int getNbOfCylinders();
}
```

Listing 10.17. Manufacturer interface

```
public interface Manufacturer {
    String getName();
}
```

Your task is to write some tests for the `findSportCars()` method of the `CarSearch` class. Basically, what you have to do is pass some cars to the `CarSearch` class (using its `addCar()`) method, and then verify, whether only sports cars are being returned by the `findSportsCars()` method.

Initially, do this for the original implementation of the `CarSearch` class. Then, redesign the `Car` interface, so that the `CarSearch` class does not violate either the "Law of Demeter" or the "Tell, Don’t Ask!" principles, and write the tests once again. Compare the amount of work involved in each case

10.7.2. Stack Test

Based on what was discussed in Section 10.1, implement a `Stack` class and a corresponding `StackTest` class. Please follow the TDD approach, so that the tests are written before the implementation. Make sure you think in terms of class responsibilities!

Thank you for reading!

I hope you have enjoyed it.

Please visit

http://practicalunittesting.com

to learn more about the book.