QWAN's Little Book of Test Driven Development

powered by Quality Without a Name

In steering your development process you adopt a simple set of practices that give you the feeling that you want. As development continues, you are constantly aware of which practices enhance and which practices detract from your goal. Each practice is an experiment, to be followed until proven inadequate.

from: Kent Beck, eXtreme Programming Explained 1st ed., p. 28

edition: September 2017

Test Driven Development heuristics

When helping my father doing carpentry, I was struggling to cut a plank in half with a saw. My father made it look so easy. His saw went through wood like a knife cutting through butter. His cuts where clean and straight. I was sweating to get the saw through. Half way through, I was getting fatigue in my arm and my cuts where ugly, uneven. It was frustrating.

I remember my father kept saying: "Just let the saw do the work and it helps if you go back and forth in a straight line." When cutting a plank, his words still resonate in my head.

"Let the saw do the work", really means "don't put too much pressure on the saw". It will make for cleaner cuts and will prevent fatigue in your arm. Similarly, "Going back and forth in a straight line" helps the saw cut easier though the wood, prevent fatigue, and makes for cleaner cuts as well. The things he said were not practices, but rather principles and heuristics that make the process smoother and the end result better.

Like cutting a plank, Test Driven Development (TDD) is deceptively simple to explain but hard to do well. Over the years of training and mentoring people we have collected quite a few heuristics and principles that help us to write, *and explain how to write*, tests that are clean, maintainable, and helpful.

We have summarized a few of these heuristics below, in the hope that they will help you cut cleaner tests.

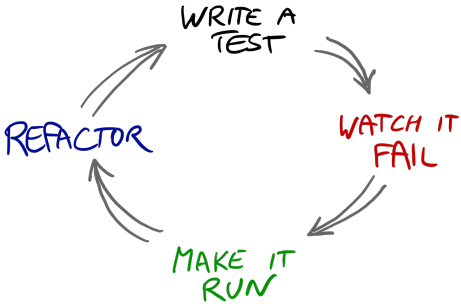
No silver bullet, the Test Driven Development version

There are only four steps in Test Driven Development (TDD):

1. First, we *write a test* - isolate the problem, make it concrete by expressing it in code. Provide details of the interface you're designing and the behaviour you're expecting.
2. Run the test and *watch it fail*. As we did not write any code yet, we expect missing definitions & missing behaviour. There should be only a few failures. A Red Bar is progress!

3. *Make it pass* - step by step we add code, run the test, and see less (or different) failures. We strive for simplicity: we remove one failure at a time and add just enough code to make it pass.

4. *Refactor* - once we have a green bar, we can safely clean up code, remove duplication, make the code express its intent better. This is what's called Refactoring - small, well defined improvements that will keep the bar green; small improvements that you do continuously.



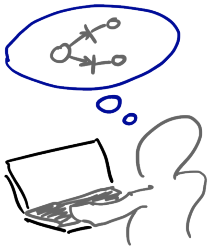
These steps look simple, but they are not easy. There are no tools that magically make TDD easier. Even tools that do make some aspects easier can have side effects that make your life harder. Refactoring tools are great, because they let you make big, breaking changes more easily. Refactoring tools are bad, because they let you make big, breaking changes more easily... Sometimes your design is lacking a component or a clear separation of concern and your tools help you proceed with this flawed design, instead of forcing you to tackle the flaws right away.

TDD requires discipline and deliberate practice.

Heuristics

Think about design in the test

When you write a test, you're actually writing an example of how the object-under-test will be used. Writing a test before the production code means you're specifying the API of the object-under-test and making this concrete in code: both syntax (method names, parameters) and semantics (what is the behaviour, what do the methods do).



Writing a test is an act of (micro) design. It will give you feedback on your design decisions, feedback about names, parameters, results, errors, call semantics, etc. Take your time to think your test through and to observe the test code once written.

Wishful thinking

Write the test based on how you wish the object under test could be used. Then make it work. Prevent letting you be guided by constraints and implementation details only... nothing is holding you back! Just write it as you'd wish the production code to be...

By writing test scenarios like this, we can already get feedback about our design ideas without having to implement them first. This enables us to steer the design just in time, based on concrete examples in code. It is a bit like sketching an interaction diagram on the whiteboard, but you are expressing it precisely in code.

The practice of wishful thinking has been around for a while, see:
Abelson & Sussman, Structure and Interpretation of Computer Programs

Test name describes the action and the expected result

We use test names or test descriptions to tell what the test is about: what is the action or event, what is the expected result?

If we name our tests like this, a test case will read like a specification of the object under test (in particular in a test report or when you look at your tests collapsed in your IDE). A good test name will also make the test more readable.

Trying to catch the test's intention in the name also helps to think of what exactly we're trying to do in the test. If it is hard to find a good name, we probably don't understand what we are trying to capture in the test.

We are not afraid of long names, but we want to prevent long, vague sentences. Names containing 'and' are a hint that there might be two (or more!) tests in there.

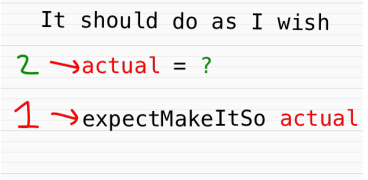
This heuristic helps thinking Given-When-Then: the test name gives a summary of the When and the Then.

```
// A nondescriptive name:
public void testDeliver() { ...

// We prefer:
public void deliversCanWhenPaidEnough() { ...
```

For more inspiration, take a look at RSpec - rspec.info

Start with an expectation

Write the last part of your test first, start with the assert or expectation. This may feel strange, as you may be inclined to write your test from top to bottom, from set-up, through invocation of production code, to expectation. This may feel the wrong way around.



It is ok to have it feel wrong and awkward. Starting with an expectation, you inject a little deliberate practice in your day to day work.

The test will focus on a desired outcome, which is often easier to express than the setup code needed. You will also notice duplications in set up code more clearly. Starting with the expectation is harder than just copy-pasting the previous test and changing some words. It forces you to think of desired behaviour and interfaces right from the beginning.

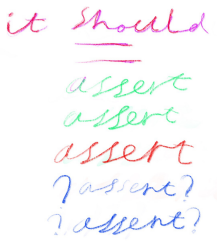
You may believe that this does not apply to legacy code. Quite the contrary: it is even harder, but will help you identify better, less cluttered, and more focused interfaces for existing code. The effort may be big, but so is the feeling of increased understanding and relief after you've done it.

See also: Wishful thinking, One assert per test, Given / When / Then

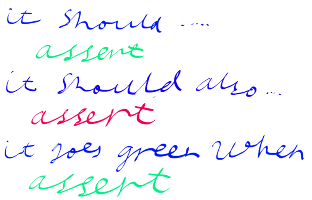
- TDD as if you Meant it
  cumulative-hypotheses.org/2011/08/30/tdd-as-if-you-meant-it

One (conceptual) assert per test

We do not like long test scenarios with loads of different asserts. A test case having many expectations is difficult to understand when it fails.



A test should have one (and only one) reason to fail. For each mistake, only one test should fail.



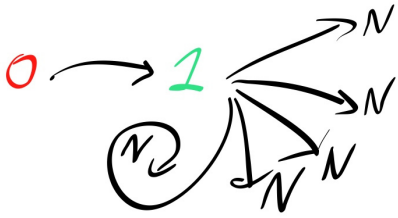
Test one thing per test:

```
public void savesOrderAndNotifiesOwnerIfPaid() {
  // this test asserts different behaviours:
  verify(repository).save(order);
  verify(notifier).notify("owner@x.com", orderNumber)
}

// Preferably, we split them up:
public void savesOrderIfPaid() {
  ....
  verify(repository).save(order);
}
public void notifiesOwnerIfPaid() {
```

```
  ....
  verify(notifier).notify("owner@x.com", orderNumber)
}
```

0, 1, N

Where do I start when writing a test? Start with a negative case for the behaviour, then add one positive, one more, and maybe that is enough. You are likely to find more cases after you start, getting over that 'blank piece of paper' feeling is often the biggest hurdle.



Starting with a negative case prevents you from only writing tests for happy paths. It is also often the smallest step you can take. Completing a step feels good and prepares you to take the next. Before you know it you may have a whole suite of these things.

An example:

> If we would test-drive a vending machine, we would start with the `deliversNothingWhenEmpty` test, then do `deliversColaWhenChoosingCola` - a single choice.
>
> Finally, we do `deliversCanOfChoice`, which allows us to generalize.

Act stupid in implementation

Do the simplest thing that could possibly work in production code, so that the test will pass. Cheating and faking are allowed.

Stupid means very simple, not complicated. We don't write conditionals to cheat; when it becomes complicated or difficult to cheat, we stop cheating and go for the simplest good solution.

An example: if we need to add a comparison to make a test pass, we prefer to start with $==$, not with $>=$, because there is probably no test yet that requires the greater than.

```
public void deliversItemWhenPaid() {
  ...
  machine.pay(1);
  assertThat(machine.deliver(), is(expectedItem));
}
class VendingMachine {
  public Optional<Can> deliver(Choice choice) {
    ...
    if (amountPaid == price) {
      // not >=, unless we have a test that demands it!
    }
  }
}
```

This feels awkward, because we write almost nothing. We 'know' what the implementation should be. But quite often we don't. So we go for the simplest thing and we end up with a much simpler and more elegant solution than we initially imagined.

Faking & Cheating

One way of *acting stupid in implementation* is to let the test pass by faking it, for example by returning the expected value as a constant. This is allowed, we even recommend it! If you can get away with faking, it means you need another test case to force yourself to write a more generic implementation.

Faking might feel weird at the beginning, but it helps to proceed towards your goal in baby steps and have continuous progress.

When we fake, we do it with simple code - no ifs, no boolean flags. We've learned that if we try to fake with flags or ifs, we'll code ourselves into a corner.

```
public void deliversColaWhenChoosingCola() {
  assertThat(machine.deliver(COLA_BUTTON), is(Can.of(COLA)))
}

class VendingMachine {
  // Make it work by faking - returning a constant value:
  public Optional<Can> deliver(Choice choice) {
    return Can.of(COLA);
  }
}
```

Triangulation

Triangulation is a TDD technique to generalize production code:

We start with one example (one test); we make it pass by faking it. We add a second example. We now have two data points, we can generalize the production code to suit the generic case.

```
// start with 1 example:
public void deliversColaWhenChoosingCola() {
  assertThat(machine.deliver(COLA_BUTTON), is(Can.of(COLA)))
}
class VendingMachine {
  public Optional<Can> deliver(Choice choice) {
    return Can.of(COLA);    // fake it!
}}

// add a second example:
public void deliversColaWhenChoosingCola() {
  assertThat(machine.deliver(COLA_BUTTON), is(Can.of(COLA)))
}
```

```
public void deliversFantaWhenChoosingFanta() {
  assertThat(machine.deliver(FANTA_BUTTON), is(Can.of(FANTA)))
}
class VendingMachine {
  Map<Choice, Can> choices = ...
  public Optional<Can> deliver(Choice choice) {
    return choices.get(choice);   // generalize it
}}
```

Test cases follow common setup

Do you need a different setup for some of the tests? A different setup is a different situation, you're testing a different aspect of the object's behaviour. We put that in a separate Test Case that describes that behaviour.

This will prevent our test being cluttered with all kind of different setups and situations. The test becomes cleaner and easier to understand.

```
class MachineTest
  def test1
    # sets up filled machine
  def test2
    # sets up filled machine
  def test3
    # sets up empty machine

# Split the tests:
class FilledMachineTest
  def test1
  def test2

class EmptyMachineTest
  def test3
```

Listen to the production code

When going through the TDD cycle in baby steps, each time you refine the production code a bit. Pay close attention to the production code as it evolves. What code smells start appearing? Is there a hint of duplication? Listen carefully to what the production code is trying to tell you, use the refactoring step to continuously improve the design of your code.

So we start outside in, apply Wishful Thinking, we Cheat, and we are also programming the inside. Let the production code tell you whether more tests are needed.

See also: Freeman & Pryce, Growing OO Software, Guided by Tests (2009)

Take as much care for test code as you do for production code

The tests are an important part of your software product. They will not only guide you when growing the production code, but they will also act as documentation (examples of how the production code behaves in different situations) and as a safety net (a feedback system that provides information about the impact of changes).

Tests need to be clean, readable, fast, explicit, intention revealing! So in the refactor step of the TDD cycle, we take care of our test code as well.

Tests are independent - always start from a clean slate

Making tests dependent on each other creates confusing test cases. When tests are dependent, it also becomes harder to get meaningful feedback. So ensure that each test has its own clean setup.

What if for instance a second test scenario continues where the first one finishes? It might look efficient to let the second test run after the first... until you get confused by the error messages when they fail.

A test is responsible for setting up the state it needs. Do not rely on proper teardown performed by other tests. Setting up a clean situation for each test should not be a problem if you have encapsulated all external dependencies, like databases/files/network access.

Sometimes we run into unintentional dependencies between tests. If a test depends on some global state or singleton, the left over state of another test might cause a test to fail, sometimes, on some machines...

Avoid global state, singletons and other global variables. The knowledge that an object is a singleton should be isolated in a small corner in your code. Most object don't need to know this.

If you still have to handle global state, make sure you manage it diligently. Make sure each test starts with a clean setup and clean, known global state, and clean up everything afterwards.

Given-When-Then or Arrange-Act-Assert

Given-When-Then and Arrange-Act-Assert are two similar ways of structuring your test cases to make them easier to understand at a glance. This test structure also helps to keep your test focused on a single concern.

> *Given* a state or situation (which you set up in the test)
>
> *When* I do something or an event happens
>
> *Then* I expect an outcome (assert) or interaction between collaborators (mock verify)

Given-When-Then originates from Behaviour Driven Development. Thinking Given-When-Then helps to think in terms of behaviour rather than internal state. Arrange-Act-Assert is a similar pattern from the eXtreme Programming community:

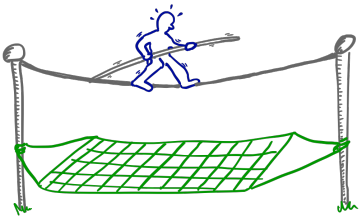
xp123.com/articles/3a-arrange-act-assert

An example:

```
# Given a filled machine

machine = VendingMachine.new
machine.fill(...);

# When I choose Cola
canOfChoice = machine.deliver(Choice.COLA)

# Then I expect a can of Cola
canOfChoice.should == Can.CanOfCola
```

Keep the bar green to keep the code clean

We strive to keep all our (finished) tests green all the time. The only failing test is the one we're currently working on. Keeping all tests green gives us the freedom to refactor, the tests will be our safety net.



"Keep the bar green to keep the code clean" quote from Gerard Meszaros, XUnit Test Patterns: Refactoring Test Code (2008)

No more than one failing test at a time

When you have only one failing test, there is no doubt about what to work on next. When, say, 20% of your tests are failing, you have to triage, which takes mental effort (decision fatigue). Or worse, when sometimes 20% and sometimes 25% of your tests are failing (flickering tests), you don't know what state you are in at all, and you cannot work on the code base with any degree of confidence.

Fix one failing test at a time

When you are changing an API or widely used interface, it can be tempting to do the change in one go and break a lot of tests at once, thinking you can just fix them. You risk ending up chasing failing tests around for hours, with lack of good feedback why they fail. And you risk breaking something along the way, because your safety net has holes!

The real art of TDD is being able to take small steps in any circumstance. If your next step breaks code in many places this may mean:

- your design has too many dependencies on one piece
- your tests have too much duplication
- tests share setup code where they should not

So how do you reduce step size?

Refactor first before adding the next test. Change the code while keeping the behaviour. Move the code in the direction you need, so that the intended change just fits in. Some usfeul techniques are:

- Test cases follow common setup
- Create 'example objects', test fixtures, or builders you can reuse across tests
- Extract a domain specific language (DSL) that encapsulates steps in your tests, to isolate the effect of changes

Another way to change the signature of a widely used method, is by adding default parameters, or a method or constructor overload in the production code, so that existing tests and code keep working.

The tests get more specific, the code gets more generic

> "As a test suite grows, it becomes ever more specific. i.e. it becomes an ever more detailed specification of behavior. Good software developers meet this increase in specification by increasing the generality of their code. To say this differently: Programmers make specific cases work by writing code that makes the general case work."

> "As a rule, the production code is getting more and more general if you can think of tests that you have not written; but that the production code will pass anyway. If the changes you make to the production code, pursuant to a test, make that test pass, but would not make other unwritten tests pass, then you are likely making the production code too specific."

Source: Robert C. Martin
blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html

What is a unit test?

A unit test does not:

- talk to the database
- communicate across the network
- touch the file system
- prevent other unit tests from running at the same time
- require you to do special things to your environment to run it, like editing config files

Such a test might still be a useful test, but it is not a *unit* test.

Source: Michael Feathers: A Set of Unit Testing Rules
www.artima.com/weblogs/viewpost.jsp?thread=126923

In addition to the above: if your unit test has a teardown, it is probably not a unit test. The one exception to this rule might be languages like C that do not manage memory for you, and you have to free memory after running the test.

Principles

Baby steps

We take tiny steps, a single test at a time, one refactoring at a time, ... Baby steps are not just for beginners; it is a skill, a discipline, something you can and should learn through practise.

Baby steps don't slow us down. On the contrary, we go fast by taking small, conscious steps, we keep on moving towards our goals every minute. Baby steps have saved us many times, especially under pressure - they're also a great way to keep your head cool.

You aren't gonna need it (YAGNI)

Whenever you start putting in a bit of complexity in the code, think twice. Is this really needed for the problem at hand? Does the test really require this? Or are you writing speculative code?

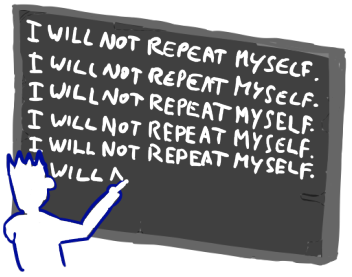
If you only write what is needed now, in the simplest way possible, your code will be much more malleable - much easier to understand and change in the future. You are not going to need it!

If you really think the extra behaviour should be in there, put it on your todo list, and write another test.

Source: Kent Beck, eXtreme Programming Explained, 1st edition (1999)

Don't repeat yourself

Duplication is a smell, it is your code trying to tell you there's a concept or thought in there that wants to be expressed, an intent that wants to be revealed.



Whenever we spot duplication, we remove it, by extracting methods, a common setup method, etc.

In unit tests, we do tolerate a bit of duplication in the tests in favour of clearness and readability.

Source: Andy Hunt & Dave Thomas, The Pragmatic Programmer (1999)

Simple Design Rules

A simple design is a design that:

1. passes all tests
2. reveals intention
3. has no duplication
4. has the least amount of classes and methods
5. has an abstraction level appropriate for the team working on the code

These rules are in priority order: so passing all tests has priority over revealing intention.

Source: martinfowler.com/bliki/BeckDesignRules.html

Simple Design Rules, for tests

Simple design rules for tests are similar but a bit different:

1. The test expresses the intent of the programmer.
2. The test passes.
3. The test has no duplication.
4. The test has a minimum of classes and methods.

When you're writing a test, you first focus on expressing the intent of the test. It does not pass yet. Once the test is written and it passes, you refactor the tests like you refactor your production code. Expressing intent takes priority over removing duplication, but lots of duplication in tests is a serious smell you need to attend to.

Intention comes first, letting the test pass test second, because you start with a failing test expressing your intent.

"You aren't gonna need it" (YAGNI) also applies to tests, but slightly differently: when you design a method or function interface and you desire a parameter, you put it in the test, even when you won't use it in the implementation just yet. Continue writing tests until your implementation is actually using the parameter.

Source: Robert C. Martin, blog.8thlight.com/uncle-bob/2013/09/23/Test-first.html

See also: Cheating

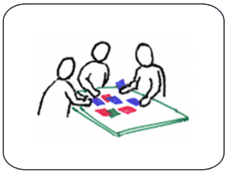
References

Some books and articles that inspired us, and that may inspire you in writing better tests and code:

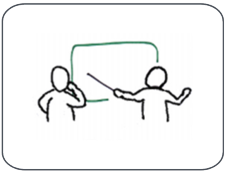
- Fred Brooks, No Silver Bullet - Essence and Accident in Software Engineering faculty.salisbury.edu/~xswang/Research/Papers/SERelated/no-silver-bullet.pdf
- Kent Beck, Test Driven Development By Example (2000)
- Kent Beck, eXtreme Programming Explained, 1st edition (1999)
- Jim Shore, The Art of Agile Development (2008)
- Lasse Koskela, Test Driven (2008)

We are QWAN – Quality Without a Name

We are Quality Without A Name, an initiative of pragmatic practitioners. We train and mentor teams, coach individuals and carry out bespoke development projects. We specialize in agile and lean software development.



Consultancy and Mentoring



Training

Development

By working with us, our customers have reduced their time to market, while improving software quality. This may seem like a paradox, but we've found that by improving quality we can keep costs down and at the same time deliver faster. Whenever customers get software that 'just works' they become happier, complain less and tell their friends. Combined with a shorter time to market this means a better return on your investments. You can use that money to reward investors, employees or taxpayers and find even better ways to develop software, so that the virtuous cycle continues.

How we can uncover better ways depends on your context. Invite us for a conversation and together we can find out what we can do for you.

{ willem | rob | marc } @ qwan.eu

Get your Code Smells & Refactorings Card decks from:
www.qwan.eu/shop


DEVELOPMENT
PRACTICES
AGILE
SOFTWARE
COURSE
PROGRAMMING
SCRUM
TESTING
QUALITY
PRINCIPLES
PROJECT
DAY
FOCUSES
CSM
DOOR
INCREMENTALLY
INTEGRATION
EXPERIENCE
MINI
BASICS
PRIDE
FEEDBACK
DESIGN
END
TEST
PAIR
DELIVER
EXERCISES
USING
RELENTLESS
RETROSPECTIVE
UNDERSTAND
SKILLS
TEST
DEVELOPMENT
USER
CERTIFIED
DRIVEN
TOGETHER
AUTOMATED
ONE
ARCHITECTS
VALUES
DAY
TECHNICAL
PRESENTATIONS
DONE
EXTREME
REFACTORING
DEMO
HANDS
AGILE
GET
CONTINUOUS
INCREMENTAL
CSPO
LEARN
DEVELOPER
FUN

www.qwan.eu