Test-Driven Development(TDD)

"Test-driven development" refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring). Even though test-driven development seems to be additional effort at first, it yields many benefits in the long term. Among them are agility, speed, and safety. The combination with Continuous Integration and Deployment results in a solid software development process proven by many modern software projects. In test-driven development, software is developed iteratively. For each feature you begin with a test that represents the most important functionality of this feature. This test should fail.

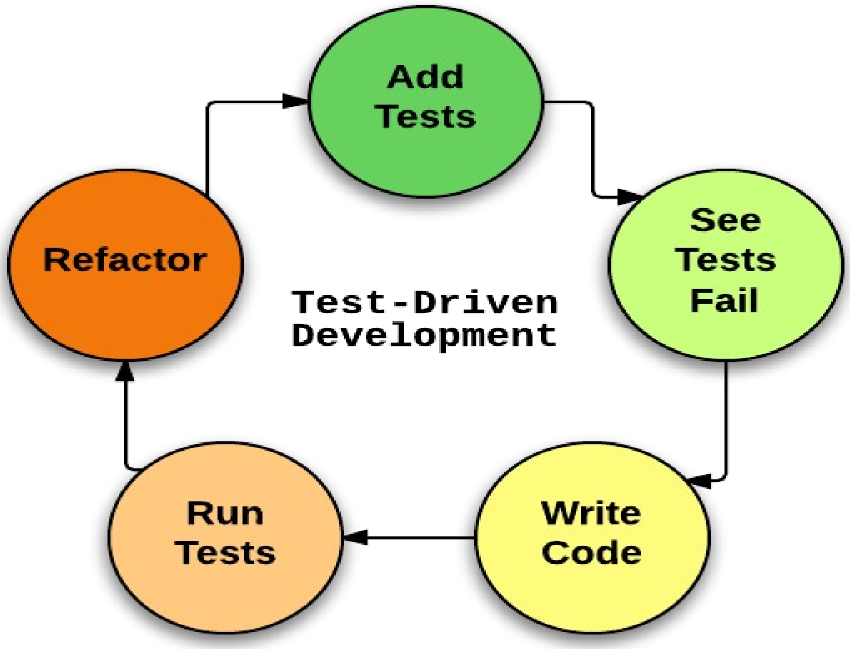


Fig:Test-driven development flowchart.

If we were to unpack the definition of TDD a bit more, we'd see that it is usually broken up into five different stages:

- First the developer writes some tests.
- The developer then runs those tests and (obviously) they fail because none of those features are actually implemented.
- Next the developer actually implements those tests in code.
- If the developer writes his code well, then the in next stage he will see his tests pass.
- The developer can then refactor his code, add comments, clean it up, as he wishes because the developer knows that if the new code breaks something, then the tests will alert him by failing.

The cycle can just continue as long as the developer has more features to add

Common Pitfalls

Typical individual mistakes include:

- forgetting to run tests frequently
- writing too many tests at once
- writing tests that are too large or coarse-grained
- writing overly trivial tests, for instance omitting assertions
- writing tests for trivial code, for instance accessors

Typical team pitfalls include:

- partial adoption - only a few developers on the team use TDD
- poor maintenance of the test suite - most commonly leading to a test suite with a prohibitively long running time
- abandoned test suite (i.e. seldom or never run) - sometimes as a result of poor maintenance, sometimes as a result of team turnover

Expected Benefits

- many teams report significant reductions in defect rates, at the cost of a moderate increase in initial development effort
- the same teams tend to report that these overheads are more than offset by a reduction in effort in projects' final phases

Although empirical research has so far failed to confirm this, veteran practitioners report that TDD leads to improved design qualities in the code, and more generally a higher degree of "internal" or technical quality, for instance improving the metrics of cohesion and coupling

Test-driven development keeps you agile. You will be slower at first but you avoid being stuck in the most critical situation: Imagine you’ve built this awesome new app in record time and you are getting your first customers. Congratulations! You start implementing this new feature your customers have asked for but suddenly your customers report that they can’t make payments anymore. As you try to fix this bug, you receive messages from angry customers telling you that the login is broken. You’re stuck. Instead of moving forward you’re working 24/7 trying to reconstruct the basic functionalities of your application. Test-driven development makes you sleep safe. You can be sure that everything you implemented still works. You can develop at a constant speed even years later when your app has grown massively and you have forgotten all these little implementation details. Your tests remember them and make sure they still work. Think of your tests as your little friends that watch over your app so you don’t have to.

Test-Driven Development and continuous integration

Over time you will gather an impressive test suite that covers all features of your application. The only problem is: The more tests you write, the longer they take to run. If you run all the tests several times a day this will slow you down more and more. A good solution is to run only the tests that correspond to your current feature and let the whole test suite run on a Continuous Integration system. This way you stay fast while still ensuring that your entire application keeps working. This way every new feature is deployed automatically if all your tests pass. You just need to take care of developing software anymore.

Behavior-Driven Development(BDD)

TDD focuses on the developer’s opinion on how parts of the software should work. BDD focuses on the users’ opinion on how they want your application to behave. BDD (Behavior Driven Development) is a synthesis and refinement of practices stemming from TDD .

Common Pitfalls

Although Dan North, who first formulated the BDD approach, claims that it was designed to address recurring issues in the teaching of TDD, it is clear that BDD requires familiarity with a greater range of concepts than TDD does, and it seems difficult to recommend a novice programmer should first learn BDD without prior exposure to TDD concepts. The use of BDD requires no particular tools or programming languages, and is primarily a conceptual approach; to make it a purely technical practice or one that hinges on specific tooling would be to miss the point altogether.

Expected Benefits

Teams already using TDD or ATDD may want to consider BDD for several reasons:

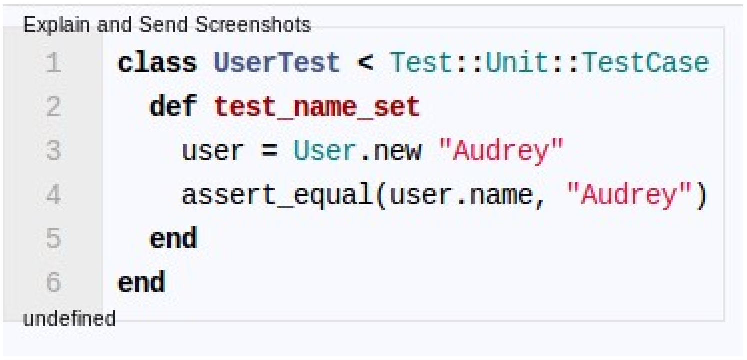
- BDD offers more precise guidance on organizing the conversation between developers, testers and domain experts
- notations originating in the BDD approach, in particular the given-when-then canvas, are closer to everyday language and have a shallower learning curve compared to those of tools such as Fit/FitNesse
- tools targeting a BDD approach generally afford the automatic generation of technical and end user documentation from BDD "specifications"

Test-driven development is rather a paradigm than a process. It describes the cycle of writing a test first, and application code afterwards – followed by an optional refactoring. But it doesn’t make any statements about:

- Where do I begin to develop?
- What exactly should I test?
- How should tests be structured and named?
- The name test-driven development also caused confusion. How can you test something that’s not there yet?

Instead of writing tests you should think of specifying behavior. Behavior is how the user wants the application to behave. When your development is Behavior-driven, you always start with the piece of functionality that’s most important to your user. We should consider this phase as taking the developer hat off and putting the user hat on. Once you’ve specified the user needs, you put the developer hat back on and implement your specification.

For now, let’s assume we know what’s most important to our users. So we know where to get started, but how do we specify Behavior? In traditional unit testing frameworks like Test::Unit a test would look like this:

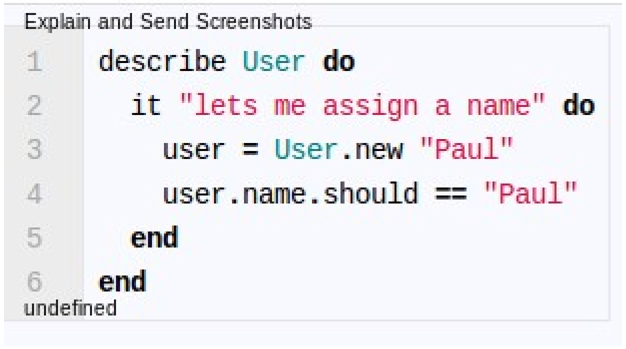


```
class UserTest < Test::Unit::TestCase
  def test_name_set
    user = User.new "Audrey"
    assert_equal(user.name, "Audrey")
  end
end
```

This certainly works, but there are some flaws:

- There’s the word test in the class name and the method name, but we’d like to specify requirements instead.
- The syntax is understandable but still appears artificial.
- And most importantly: The test name doesn’t state what it really tests.

In Behavior-driven development you specify Behavior in whole sentences. Not just the naming but also the code syntax should read naturally:


Explain and Send Screenshots

```
describe User do
  it "lets me assign a name" do
    user = User.new "Paul"
    user.name.should == "Paul"
  end
end
```

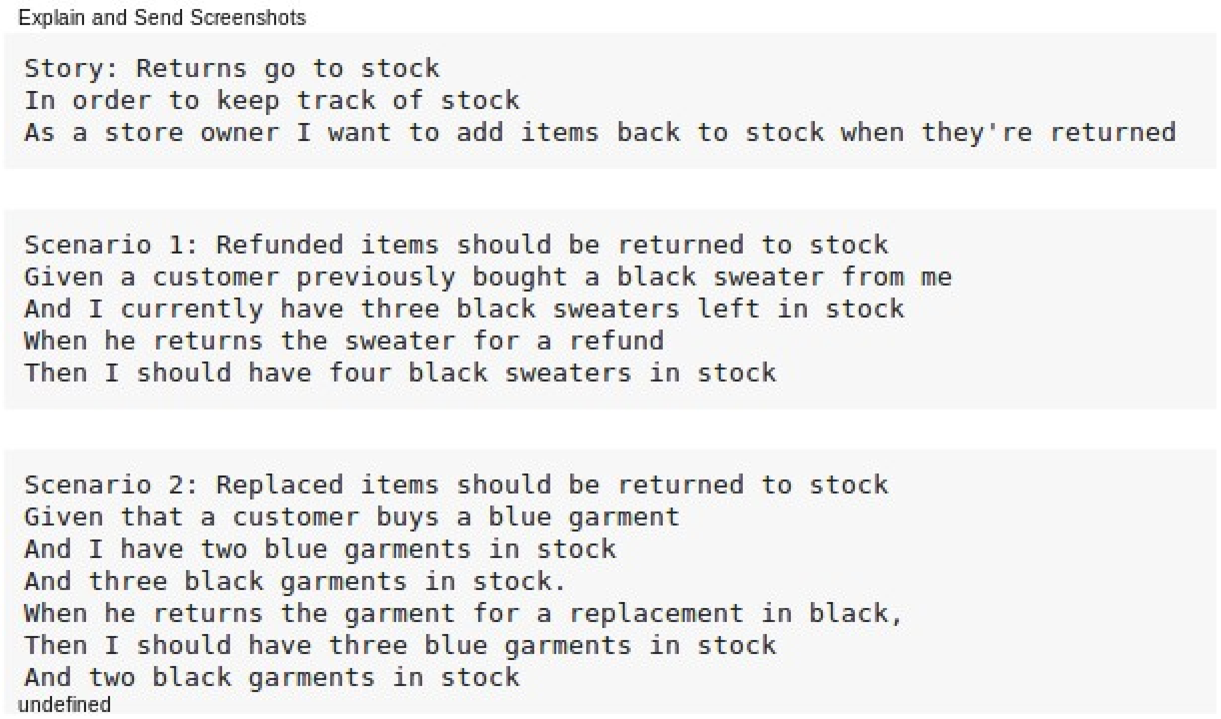
undefined

Now we know that a user lets you assign a name. But what’s the real value of this syntax?

- Focus: You test exactly what the sentence says. No more, no less. This will let you write fine-grained and expressive specifications.
- Documentation: Just by reading the sentence, your team-mates will understand what this Behavior is about. They don’t need to read a single line of code.
- Regression: When you run all these specifications later on, they become regression tests. When a regression test fails, you immediately see what Behavior of your application is broken.

But while this syntax is useful for specifying fine-grained Behavior of your application’s components, it still doesn’t say anything about the intentions of your users.

A template is suggested that lets you describe features in natural language:



Explain and Send Screenshots

```
Story: Returns go to stock
In order to keep track of stock
As a store owner I want to add items back to stock when they're returned
```

```
Scenario 1: Refunded items should be returned to stock
Given a customer previously bought a black sweater from me
And I currently have three black sweaters left in stock
When he returns the sweater for a refund
Then I should have four black sweaters in stock
```

```
Scenario 2: Replaced items should be returned to stock
Given that a customer buys a blue garment
And I have two blue garments in stock
And three black garments in stock.
When he returns the garment for a replacement in black,
Then I should have three blue garments in stock
And two black garments in stock
```

undefined

Each story has a title and a short description of what the story is about. The format of this description is always the same:

- In order to get some benefit
- As the user you are developing for
- I want what this feature does

This description is always followed by a list of scenarios containing ***Given*** steps (what has happened before), ***When*** steps (what actions the user performs), and ***Then*** steps (the desired outcome for the user). Here, we also apply the principle: Start with the Behavior that's most important to the user.

- Find out the feature that's most important.
- Within the feature always select the most important scenario.

This way you always stay aligned with your user's needs and focus on the stuff that matters.

Behavior-driven development focuses on the purpose of your work to people who will use it. This way you will create better software and successfully address your customers' needs. The technical solution arises from this process almost by itself.

References:

- http://blog.codeship.io/2013/04/16/tests-make-software.html
- http://guide.agilealliance.org/guide/tdd.html
- http://guide.agilealliance.org/guide/bdd.html
- http://joshldavis.com/2013/05/27/difference-between-tdd-and-bdd/
- http://blog.mattwynne.net/2012/11/20/tdd-vs-bdd/