

What is Test Driven Development(TDD)?

Test-driven development (TDD) is a software development methodology in which tests are written for a piece of code before the code itself is written. The idea behind TDD is that by writing tests first, you can ensure that your code behaves as expected and meets the requirements.

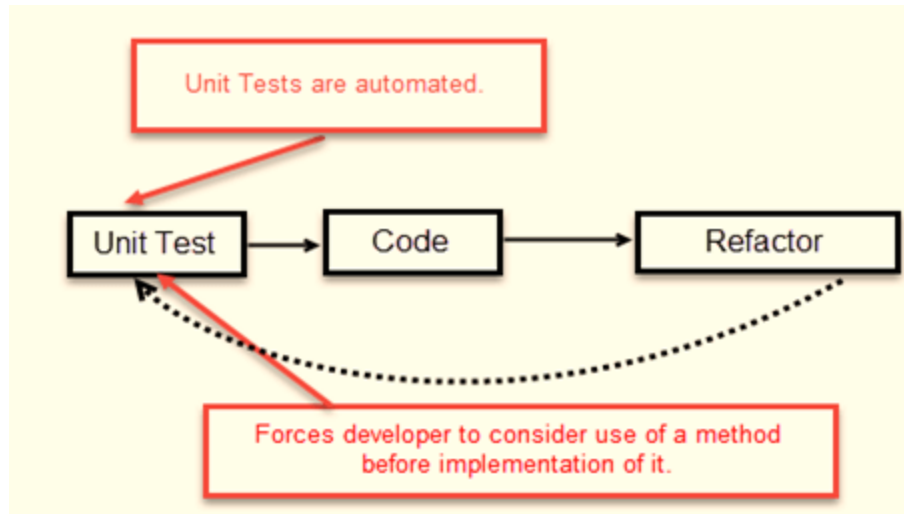
Here is an example of the TDD process:

1. Write a test that describes the desired behavior of the code.
2. Run the test and verify that it fails (since the code has not been written yet).
3. Write the minimum amount of code needed to make the test pass.
4. Run the test again and verify that it passes.
5. Refactor the code if necessary, making sure to run the tests after each refactoring step to ensure that the code still behaves as expected.

The TDD process is typically iterative, with multiple tests and code changes being made in a loop until the feature is complete. By following this process, you can ensure that your code is well-tested and behaves as expected. TDD can also help you to identify and fix issues early in the development process, which can save time and effort in the long run.

Test Driven Development (TDD) is software development approach in which test cases are developed to specify and validate what the code will do. In simple terms, test cases for each functionality are created and tested first and if the test fails then the new code is written in order to pass the test and making code simple and bug-free.

Test-Driven Development starts with designing and developing tests for every small functionality of an application. TDD framework instructs developers to write new code only if an automated test has failed. This avoids duplication of code. The TDD full form is Test-driven development.



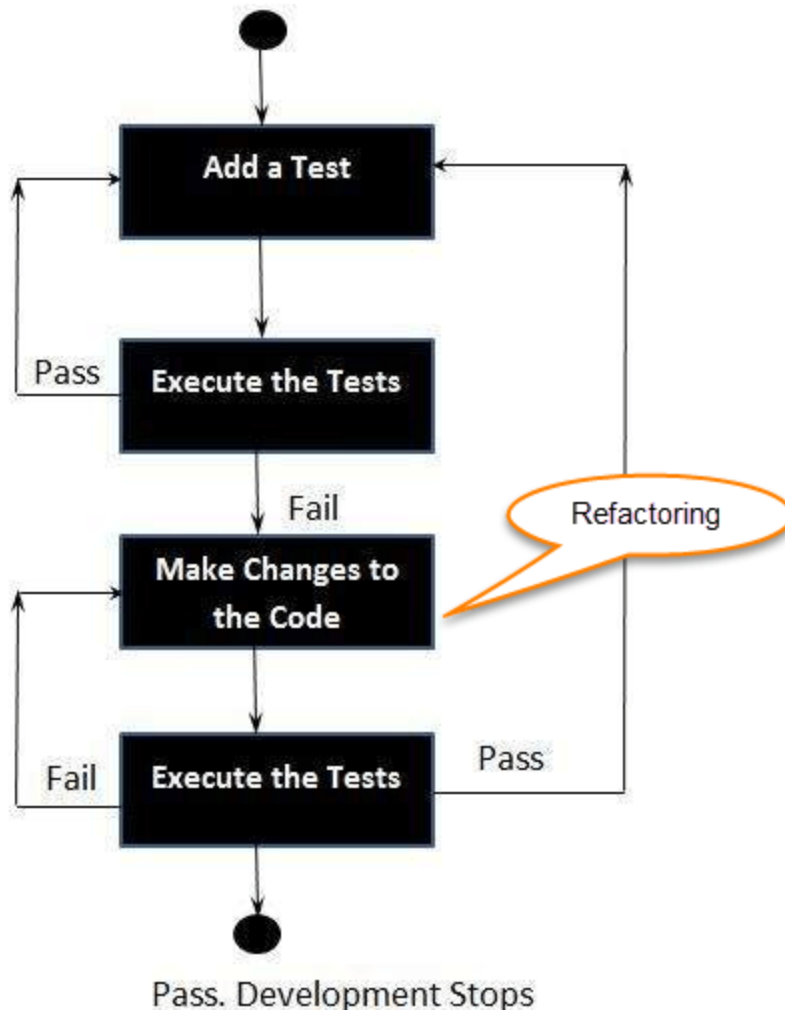
The simple concept of TDD is to write and correct the failed tests before writing new code (before development). This helps to avoid duplication of code as we write a small amount of code at a time in order to pass tests. (Tests are nothing but requirement conditions that we need to test to fulfill them).

Test-Driven development is a process of developing and running automated test before actual development of the application. Hence, TDD sometimes also called as **Test First Development**.

How to perform TDD Test

Following steps define how to perform TDD test,

1. Add a test.
2. Run all tests and see if any new test fails.
3. Write some code.
4. Run tests and Refactor code.
5. Repeat.



Five Steps of Test-Driven Development

TDD cycle defines

1. Write a test
2. Make it run.
3. Change the code to make it right i.e. Refactor.
4. Repeat process.

Some clarifications about TDD:

- TDD approach is neither about “Testing” nor about “Design”.
- TDD does not mean “write some of the tests, then build a system that passes the tests.
- TDD does not mean “do lots of Testing.”

TDD Vs. Traditional Testing

Below is the main difference between Test driven development and traditional testing:

TDD approach is primarily a specification technique. It ensures that your source code is thoroughly tested at confirmatory level.

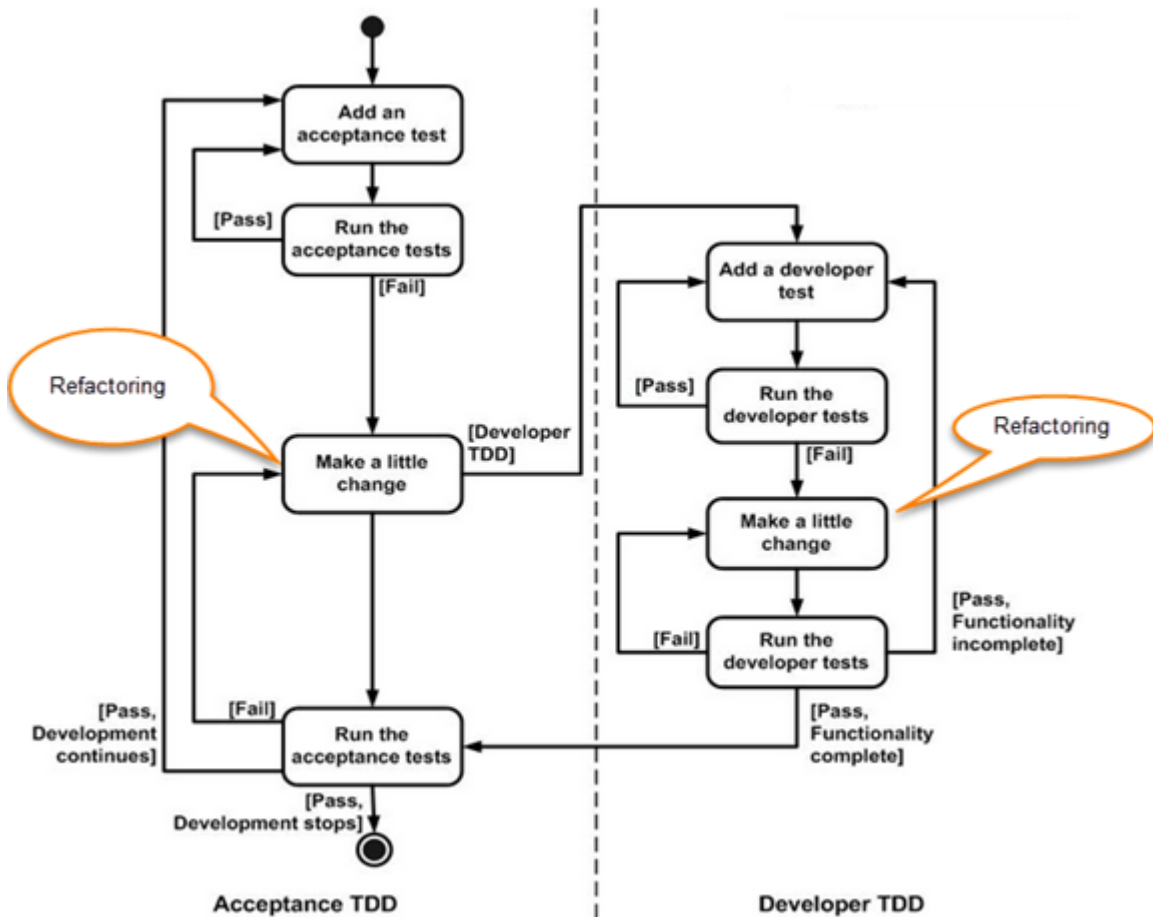
- With traditional testing, a successful test finds one or more defects. It is same as TDD. When a test fails, you have made progress because you know that you need to resolve the problem.
- TDD ensures that your system actually meets requirements defined for it. It helps to build your confidence about your system.
- In TDD more focus is on production code that verifies whether testing will work properly. In traditional testing, more focus is on test case design. Whether the test will show the proper/improper execution of the application in order to fulfill requirements.
- In TDD, you achieve 100% coverage test. Every single line of code is tested, unlike traditional testing.
- The combination of both traditional testing and TDD leads to the importance of testing the system rather than perfection of the system.
- In [Agile Modeling \(AM\)](#), you should “test with a purpose”. You should know why you are testing something and what level its need to be tested.

What is acceptance TDD and Developer TDD

There are two levels of TDD

1. **Acceptance TDD (ATDD):** With ATDD you write a single acceptance test. This test fulfills the requirement of the specification or satisfies the behavior of the system. After that write just enough production/functionality code to fulfill that acceptance test. Acceptance test focuses on the overall behavior of the system. ATDD also was known as **Behavioral Driven Development (BDD)**.
2. **Developer TDD:** With Developer TDD you write single developer test i.e. unit test and then just enough production code to fulfill that test. The unit test focuses on every small functionality of the system. Developer TDD is simply

called as **TDD**. The main goal of ATDD and TDD is to specify detailed, executable requirements for your solution on a just in time (JIT) basis. JIT means taking only those requirements in consideration that are needed in the system. So increase efficiency.

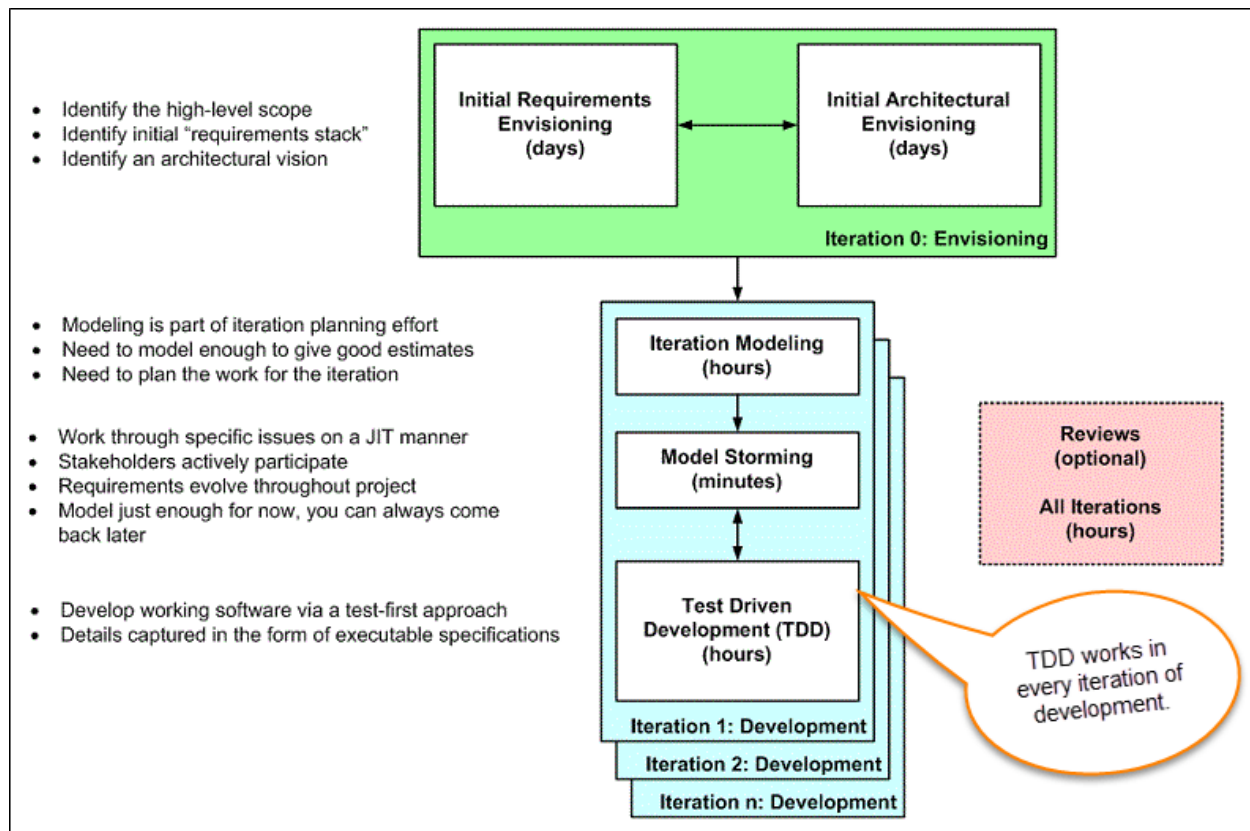


Scaling TDD via Agile Model Driven Development (AMDD)

TDD is very good at detailed specification and validation. It fails at thinking through bigger issues such as overall design, use of the system, or UI. AMDD addresses the Agile scaling issues that TDD does not.

Thus AMDD used for bigger issues.

The lifecycle of AMDD



In Model-driven Development (MDD), extensive models are created before the source code is written. Which in turn have an agile approach?

In above figure, each box represents a development activity.

Envisioning is one of the TDD process of predicting/imagining tests which will be performed during the first week of the project. The main goal of envisioning is to identify the scope of the system and architecture of the system. High-level requirements and architecture modeling is done for successful envisioning.

It is the process where not a detailed specification of software/system is done but exploring the requirements of software/system which defines the overall strategy of the project.

Iteration 0: Envisioning

There are two main sub-activities.

1. **Initial requirements envisioning.** It may take several days to identify high-level requirements and scope of the system. The main focus is to explore usage model, Initial domain model, and user interface model (UI).
2. **Initial Architectural envisioning.** It also takes several days to identify architecture of the system. It allows setting technical directions for the project. The main focus is to explore technology diagrams, User Interface (UI) flow, domain models, and Change cases.

Iteration modeling

Here team must plan the work that will be done for each iteration.

- Agile process is used for each iteration, i.e. during each iteration, new work item will be added with priority.
- First higher prioritized work will be taken into consideration. Work items added may be reprioritized or removed from items stack any time.
- The team discusses how they are going to implement each requirement. Modeling is used for this purpose.
- Modeling analysis and design is done for each requirement which is going to implement for that iteration.

Model storming

This is also known as Just in time Modeling.

- Here modeling session involves a team of 2/3 members who discuss issues on paper or whiteboard.
- One team member will ask another to model with them. This modeling session will take approximately 5 to 10 minutes. Where team members gather together to share whiteboard/paper.
- They explore issues until they don't find the main cause of the problem. Just in time, if one team member identifies the issue which he/she wants to resolve then he/she will take quick help of other team members.
- Other group members then explore the issue and then everyone continues on as before. It is also called as stand-up modeling or customer QA sessions.

Test Driven Development (TDD)

- It promotes confirmatory testing of your application code and detailed specification.
- Both acceptance test (detailed requirements) and developer tests (unit test) are inputs for TDD.
- TDD makes the code simpler and clear. It allows the developer to maintain less documentation.

Reviews

- This is optional. It includes code inspections and model reviews.
- This can be done for each iteration or for the whole project.
- This is a good option to give feedback for the project.

Test Driven Development (TDD) Vs. Agile Model Driven Development (AMDD)

TDD	AMDD
<ul style="list-style-type: none">• TDD shortens the programming feedback loop	<ul style="list-style-type: none">• AMDD shortens modeling feedback loop.
<ul style="list-style-type: none">• TDD is detailed specification	<ul style="list-style-type: none">• AMDD works for bigger issues
<ul style="list-style-type: none">• TDD promotes the development of high-quality code	<ul style="list-style-type: none">• AMDD promotes high-quality communication w and developers.
<ul style="list-style-type: none">• TDD speaks to programmers	<ul style="list-style-type: none">• AMDD talks to Business Analyst, stakeholders, a professionals.
<ul style="list-style-type: none">• TDD non-visually oriented	<ul style="list-style-type: none">• AMDD visually oriented
<ul style="list-style-type: none">• TDD has limited scope to software works	<ul style="list-style-type: none">• AMDD has a broad scope including stakeholders working towards a common understanding

- Both support evolutionary development
-

Now, let's learn Test Driven Development by example.

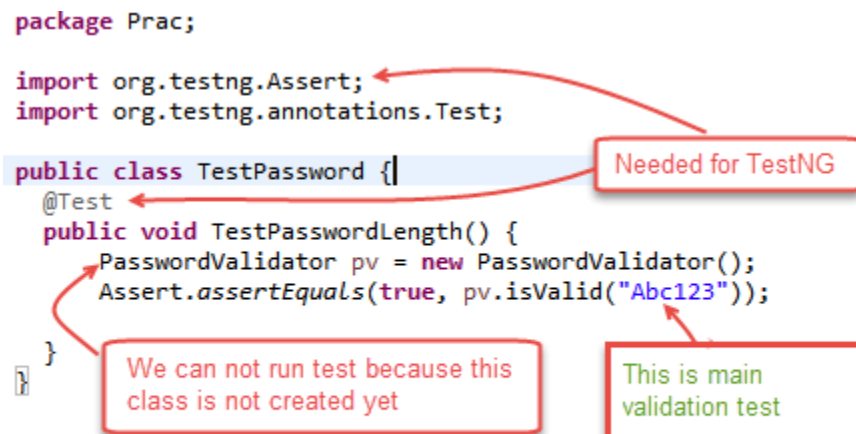
Example of TDD

Here in this Test Driven Development example, we will define a class password. For this class, we will try to satisfy following conditions.

A condition for Password acceptance:

- The password should be between 5 to 10 characters.

First in this TDD example, we write the code that fulfills all the above requirements.



Scenario 1: To run the test, we create class `PasswordValidator ()`;

```

package Prac;

public class PasswordValidator {
    public boolean isValid(String Password)
    {
        if (Password.length() >= 5 && Password.length() <= 10)
        {
            return true;
        }
        else
            return false;
    }
}

```

This is main condition checking length of password. If meets return true otherwise false.

We will run above class TestPassword ();

Output is PASSED as shown below;

Output:

```

<terminated> TestPassword [TestNG] C:\Program Files\Java\jre1.8.0_77\bin\javaw.exe (Jul 25, 2016, 2:10:22 PM)
[TestNG] Running:
  C:\Users\kanchan\AppData\Local\Temp\testng-eclipse--571370159\testng-customsuite.xml

PASSED: TestPasswordLength
=====
      Default test
      Tests run: 1, Failures: 0, Skips: 0
=====

=====
Default suite
Total tests run: 1, Failures: 0, Skips: 0
=====

[TestNG] Time taken by org.testng.reporters.EmailableReporter2@1b40d5f0: 202 ms
[TestNG] Time taken by org.testng.reporters.XMLReporter@28f67ac7: 63 ms
[TestNG] Time taken by org.testng.reporters.jq.Main@546a03af: 78 ms
[TestNG] Time taken by org.testng.reporters.JUnitReportReporter@5a01ccaa: 2 ms
[TestNG] Time taken by [FailedReporter passed=0 failed=0 skipped=0]: 1 ms
[TestNG] Time taken by org.testng.reporters.SuiteHTMLReporter@2b80d80f: 10 ms

```

Result of test as Passed

Scenario 2: Here we can see in method TestPasswordLength () there is no need of creating an instance of class PasswordValidator. Instance means creating an **object** of class to refer the members (variables/methods) of that class.

```

package Prac;

import org.testng.Assert;

public class TestPassword {
    @Test
    public void TestPasswordLength() {
        PasswordValidator pv = new PasswordValidator();
        Assert.assertEquals(true, pv.isValid("Abc123"));
    }
}

```

We will remove it.

We will remove class PasswordValidator pv = new PasswordValidator () from the code. We can call the **isValid ()** method directly by **PasswordValidator. isValid ("Abc123")**. (See image below)

So we Refactor (change code) as below:

```

package Prac;

import org.testng.Assert;
import org.testng.annotations.Test;

public class TestPassword {
    @Test
    public void TestPasswordLength() {
        Assert.assertEquals(true, PasswordValidator.isValid("Abc123"));
    }
}

```

Re factor code as there is no need of creating instance of class PasswordValidator().

Scenario 3: After refactoring the output shows failed status (see image below) this is because we have removed the instance. So there is no reference to **non – static** method **isValid ()**.

```
[TestNG] Running:
C:\Users\kanchan\AppData\Local\Temp\testng-eclipse--157192639\testng-customsuite.xml

FAILED: TestPasswordLength
java.lang.Error: Unresolved compilation problem:
  Cannot make a static reference to the non-static method isValid(String) from the type PasswordValidator

  at Prac.TestPassword.TestPasswordLength(TestPassword.java:10)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
  at java.lang.reflect.Method.invoke(Unknown Source)
  at org.testng.internal.MethodInvocationHelper.invokeMethod(MethodInvocationHelper.java:127)
  at org.testng.internal.Invoker.invokeMethod(Invoker.java:639)
  at org.testng.internal.Invoker.invokeTestMethod(Invoker.java:821)
```

If we removed instance creation statement compiler will give error. As we do not create instance it becomes non static method and there is no any reference to this method. Test results in Fail. To remove this error we have to make isValid() method of class PasswordValidator as static.

So we need to change this method by adding “static” word before Boolean as public static boolean isValid (String password). Refactoring Class PasswordValidator () to remove above error to pass the test.

```
package Prac;

public class PasswordValidator {
    public static boolean isValid(String Password)
    {
        if (Password.length() >= 5 && Password.length() <= 10)
        {
            return true;
        }
        else
            return false;
    }
}
```

Re factor : Added static word to pass test.

Output:

After making changes to class PassValidator () if we run the test then the output will be PASSED as shown below.

```
<terminated> TestPassword [TestNG] C:\Program Files\Java\jre1.8.0_77\bin\javaw.exe (Jul 25, 2016, 3:02:16 PM)
[TestNG] Running:
C:\Users\kanchan\AppData\Local\Temp\testng-eclipse--1385484104\testng-customsuite.xml
```

PASSED: TestPasswordLength

Test results passed as we changed code in class PasswordValidator().

```
=====
Default test
Tests run: 1, Failures: 0, Skips: 0
=====
```

```
=====
Default suite
Total tests run: 1, Failures: 0, Skips: 0
=====
```

```
[TestNG] Time taken by org.testng.reporters.EmailableReporter2@1b40d5f0: 19 ms
[TestNG] Time taken by org.testng.reporters.XMLReporter@28f67ac7: 10 ms
[TestNG] Time taken by org.testng.reporters.jq.Main@546a03af: 34 ms
```

Advantages of TDD

Following are the main advantages of Test Driven Development in Software Engineering:

- **Early bug notification.**
 - Developers test their code but in the database world, this often consists of manual tests or one-off scripts. Using TDD you build up, over time, a suite of automated tests that you and any other developer can rerun at will.
- **Better Designed, cleaner and more extensible code.**
 - It helps to understand how the code will be used and how it interacts with other modules.
 - It results in better design decision and more maintainable code.
 - TDD allows writing smaller code having single responsibility rather than monolithic procedures with multiple responsibilities. This makes the code simpler to understand.
 - TDD also forces to write only production code to pass tests based on user requirements.
- **Confidence to Refactor**
 - If you refactor code, there can be possibilities of breaks in the code. So having a set of automated tests you can fix those breaks before release. Proper warning will be given if breaks found when automated tests are used.
 - Using TDD, should results in faster, more extensible code with fewer bugs that can be updated with minimal risks.
- **Good for teamwork**
 - In the absence of any team member, other team members can easily pick up and work on the code. It also aids knowledge sharing, thereby making the team more effective overall.
- **Good for Developers**
 - Though developers have to spend more time in writing TDD test cases, it takes a lot less time for debugging and developing new features. You will write cleaner, less complicated code.

Summary:

- TDD stands for Test-driven development.
- TDD meaning: It is a process of modifying the code in order to pass a test designed previously.
- It more emphasis on production code rather than test case design.
- Test-driven development is a process of modifying the code in order to pass a test designed previously.
- In Software Engineering, It is sometimes known as “**Test First Development.**”
- TDD testing includes refactoring a code i.e. changing/adding some amount of code to the existing code without affecting the behavior of the code.
- TDD programming when used, the code becomes clearer and simple to understand.

<https://www.softwaretestinghelp.com/tdd-vs-bdd/>

What is BDD (Behavior Driven Development) Testing?

Behavior-driven development (BDD) is a software development methodology that focuses on defining the behavior of a system through examples. These examples, known as "scenarios," are used to drive the development of the system and ensure that it behaves as expected.

Here is an example of how BDD might be used in a software development project:

1. A product owner creates a list of high-level user stories that describe the desired behavior of the system.
2. The development team collaborates with the product owner to break down the user stories into smaller, more specific scenarios.
3. For each scenario, the team writes a detailed description of the input, action, and expected output. For example:
 - **Scenario:** User logs in with correct credentials
 - **Given:** The user has entered the correct username and password
 - **When:** The user clicks the "Log in" button
 - **Then:** The user is logged in and redirected to the dashboard
4. The development team writes code to implement the behavior described in the scenarios.
5. The team uses tools like Cucumber or JBehave to automate the execution of the scenarios and verify that the system behaves as expected.

6. If a scenario fails, the team investigates and fixes the issue.
7. Once all the scenarios pass, the team can consider the feature complete and move on to the next set of scenarios.

BDD helps teams focus on the desired behavior of the system and ensure that it is implemented correctly. It also helps to facilitate communication and collaboration between the development team and stakeholders, as the scenarios provide a clear, concise description of the system's behavior.

BDD (Behavior-driven development) Testing is a technique of agile software development and is as an extension of TDD, i.e., Test Driven Development. In BDD, test cases are written in a natural language that even non-programmers can read. In this BDD Testing tutorial, we are going to see BDD Testing of REST API with Behave and Python

- [What is BDD \(Behavior Driven Development\) Testing?](#)
- [How BDD Testing works?](#)
- [What is REST API Testing?](#)
- [What is Behave?](#)
- [Setting up BDD Testing Framework Behave on Windows](#)
- [Example POST scenario](#)
- [Steps implementation](#)
- [Running the tests](#)
- [Reports](#)

How BDD Testing works?

Consider you are assigned to create Funds Transfer module in a Net Banking application.

There are multiple ways to test it

1. Fund Transfer should take place if there is enough balance in source account

2. Fund Transfer should take place if the destination a/c details are correct
3. Fund Transfer should take place if transaction password / rsa code / security authentication for the transaction entered by user is correct
4. Fund Transfer should take place even if it's a Bank Holiday
5. Fund Transfer should take place on a future date as set by the account holder

The **Test Scenario** become more elaborate and complex as we consider additional features like transfer amount X for an interval Y days/months , stop schedule transfer when the total amount reaches Z , and so on

The general tendency of developers is to develop features and write test code later. As, evident in above case, **Test Case** development for this case is complex and developer will put off **Testing** till release , at which point he will do quick but ineffective testing.

To overcome this issue (Behavior Driven Development) BDD was conceived. It makes the entire testing process easy for a developer

In BDD, whatever you write must go into **Given-When-Then** steps. Lets consider the same example above in BDD

```
Given that a fund transfer module in net banking application has been developed
And I am accessing it with proper authentication
When I shall transfer with enough balance in my source account
Or I shall transfer on a Bank Holiday
Or I shall transfer on a future date
And destination a/c details are correct
And transaction password/rsa code / security authentication for the transaction is correct
And press or click send button
Then amount must be transferred
And the event will be logged in log file
```

Isn't it easy to write and read and understand? It covers all possible test cases for the fund transfer module and can be easily modified to accommodate more. Also, it more like writing documentation for the fund transfer module.

What is REST API Testing?

As REST has become quite a popular style for building APIs nowadays, it has become equally important to automate REST API test cases along with UI test

cases. So basically, these REST [API testing](#) involves testing of CRUD (Create-Read-Update-Delete) actions with methods POST, GET, PUT, and DELETE respectively.

What is Behave?

Behave is one of the popular Python BDD test frameworks.

Let's see how does Behave function:

Feature files are written by your Business Analyst / Sponsor / whoever with your behavior scenarios in it. It has a natural language format describing a feature or part of a feature with representative examples of expected outcomes

These Scenario steps are mapped with step implementations written in [Python](#).

And optionally, there are some environmental controls (code to run before and after steps, scenarios, features or the whole shooting match).

Let's get started with the setup of our automation test framework with Behave:

Setting up BDD Testing Framework Behave on Windows

Installation:

- Download and Install Python 3 from <https://www.python.org/>
- Execute the following command on command prompt to install behave
- `pip install behave`
- IDE: I have used PyCharm Community Edition <https://www.jetbrains.com/pycharm/download>

Project Setup:

- Create a New Project
- Create the following Directory Structure:

```
+---features/  
| +---steps/      # -- Steps directory  
| |   +--- *.py   # -- Step implementation or use step-library python files.  
| +--- *.feature  # -- Feature files.
```

Feature Files:

So let's build our feature file **Sample_REST_API_Testing.feature** having feature as Performing CRUD operations on 'posts' service.

In our example, I have used <http://jsonplaceholder.typicode.com/> posts sample REST Service.

Example POST scenario

Scenario: POST post example ->Here we are considering creating new post item using 'posts' service
Given: I set post posts API endpoint ->This is prerequisite for the test which is setting URL of posts service
When: I set HEADER param request content type as "application/json."
And set request body
And send POST HTTP request ->This is actual test step of sending a post request
Then: Then I receive valid HTTP response code 201
And Response body "POST" is non-empty-> This is verification of response body

Similarly, you can write the remaining Scenarios as follows:

1	Feature: Test CRUD methods in Sample REST API testing framework
2	
3	Background:
4	Given I set sample REST API url
5	
6	Scenario: POST post example
7	Given I Set POST posts api endpoint
8	When I Set HEADER param request content type as "application/json"
9	And Set request Body
10	And Send POST HTTP request
11	Then I receive valid HTTP response code 201
12	And Response BODY "POST" is non-empty
13	
14	
15	Scenario: GET posts example
16	Given I Set GET posts api endpoint "1"
17	When I Set HEADER param request content type as "application/json"
18	And Send GET HTTP request
19	Then I receive valid HTTP response code 200 for "GET"
20	And Response BODY "GET" is non-empty
21	
22	
23	Scenario: UPDATE posts example
24	Given I Set PUT posts api endpoint for "1"
25	When I Set Update request Body
26	And Send PUT HTTP request
27	Then I receive valid HTTP response code 200 for "PUT"
28	And Response BODY "PUT" is non-empty
29	
30	
31	Scenario: DELETE posts example
32	Given I Set DELETE posts api endpoint for "1"
33	When I Send DELETE HTTP request
34	Then I receive valid HTTP response code 200 for "DELETE"

Sample_REST_API_Testing.feature

Feature: Test CRUD methods in Sample REST API testing framework

Background:

 Given I set sample REST API url

Scenario: POST post example

 Given I Set POST posts api endpoint

 When I Set HEADER param request content type as "application/json."

 And Set request Body

 And Send a POST HTTP request

 Then I receive valid HTTP response code 201

 And Response BODY "POST" is non-empty.

Scenario: GET posts example

```
Given I Set GET posts api endpoint "1"
When I Set HEADER param request content type as "application/json."
    And Send GET HTTP request
Then I receive valid HTTP response code 200 for "GET."
    And Response BODY "GET" is non-empty
```

Scenario: UPDATE posts example

```
Given I Set PUT posts api endpoint for "1"
When I Set Update request Body
    And Send PUT HTTP request
Then I receive valid HTTP response code 200 for "PUT."
    And Response BODY "PUT" is non-empty
```

Scenario: DELETE posts example

```
Given I Set DELETE posts api endpoint for "1"
When I Send DELETE HTTP request
Then I receive valid HTTP response code 200 for "DELETE."
```

Steps Implementation

Now, for feature Steps used in the above scenarios, you can write implementations in Python files in the “steps” directory.

Behave framework identifies the Step function by decorators matching with feature file predicate. For Example, Given predicate in Feature file Scenario searches for step function having decorator “given.” Similar matching happens for When and Then. But in the case of ‘But,’ ‘And,’ Step function takes decorator same as it’s preceding step. For Example, If ‘And’ comes for Given, matching step function decorator is @given.

For Example, when step for POST can be implemented as follows:

```
@when (u'I Set HEADER param request content type as "{header_conent_type}"')
Mapping of When, here notice "application/json" is been passed from feature
file for "{header_conent_type}" . This is called as parameterization
```

```
def step_impl (context, header_conent_type):
This is step implementation method signature
```

```
request_headers['Content-Type'] = header_conent_type
Step implementation code, here you will be setting content type for request
header
```

Similarly, the implementation of other steps in the step python file will look like this:

```

1  from behave import given, when, then, step
2  import requests
3
4  api_endpoints = {}
5  request_headers = {}
6  response_codes = {}
7  response_texts = {}
8  request_bodies = {}
9  api_url = None
10
11  @given(u'I set sample REST API url')
12  def step_impl(context):
13      global api_url
14      api_url = 'http://jsonplaceholder.typicode.com'
15
16  # START POST Scenario
17  @given(u'I Set POST posts api endpoint')
18  def step_impl(context):
19      api_endpoints['POST_URL'] = api_url + '/posts'
20      print('url :'+api_endpoints['POST_URL'])
21
22  @when(u'I Set HEADER param request content type as "{header_content_type}"')
23  def step_impl(context, header_content_type):
24      request_headers['Content-Type'] = header_content_type
25
26  #You may also include "And" or "But" as a step - these are renamed by behave to take the name of their preceding step, so:
27  @when(u'Set request Body')
28  def step_impl(context):
29      request_bodies['POST'] = {"title": "foo", "body": "bar", "userId": "1"}
30
31  #You may also include "And" or "But" as a step - these are renamed by behave to take the name of their preceding step, so:
32  @when(u'Send POST HTTP request')
33  def step_impl(context):
34      # sending get request and saving response as response object
35      response = requests.post(url=api_endpoints['POST_URL'], json=request_bodies['POST'], headers=request_headers)
36      #response = requests.post(url=api_endpoints['POST_URL'], headers=request_headers) #https://jsonplaceholder.typicode.com/posts
37      # extracting response text
38      response_texts['POST'] = response.text
39      print("post response :"+response.text)
40      # extracting response status_code
41      statuscode = response.status_code
42      response_codes['POST'] = statuscode
43
44  @then(u'I receive valid HTTP response code 201')
45  def step_impl(context):
46      print('Post rep code :'+str(response_codes['POST']))
47      assert response_codes['POST'] is 201
48  # END POST Scenario
49
50  # START GET Scenario
51  @given(u'I Set GET posts api endpoint "{id}"')
52  def step_impl(context, id):
53      api_endpoints['GET_URL'] = api_url + '/posts/' + id
54      print('url :'+api_endpoints['GET_URL'])

```

Given step implementation

When step implementation

Then step implementation

sample_step_implementation.py

```

from behave import given, when, then, step
import requests

```

```

api_endpoints = {}
request_headers = {}
response_codes = {}
response_texts = {}
request_bodies = {}
api_url = None

```

```

@given(u'I set sample REST API url')

```

```

def step_impl(context):
    global api_url
    api_url = 'http://jsonplaceholder.typicode.com'

# START POST Scenario
@given(u'I Set POST posts api endpoint')
def step_impl(context):
    api_endpoints['POST_URL'] = api_url+'/posts'
    print('url :'+api_endpoints['POST_URL'])

@when(u'I Set HEADER param request content type as "{header_conent_type}"')
def step_impl(context, header_conent_type):
    request_headers['Content-Type'] = header_conent_type

#You may also include "And" or "But" as a step - these are renamed by behave
to take the name of their preceding step, so:
@when(u'Set request Body')
def step_impl(context):
    request_bodies['POST']={"title": "foo","body": "bar","userId": "1"}

#You may also include "And" or "But" as a step - these are renamed by behave
to take the name of their preceding step, so:
@when(u'Send POST HTTP request')
def step_impl(context):
    # sending get request and saving response as response object
    response = requests.post(url=api_endpoints['POST_URL'],
json=request_bodies['POST'], headers=request_headers)
    #response = requests.post(url=api_endpoints['POST_URL'],
headers=request_headers) #https://jsonplaceholder.typicode.com/posts
    # extracting response text
    response_texts['POST']=response.text
    print("post response :"+response.text)
    # extracting response status_code
    statuscode = response.status_code
    response_codes['POST'] = statuscode

@then(u'I receive valid HTTP response code 201')
def step_impl(context):
    print('Post rep code ;'+str(response_codes['POST']))
    assert response_codes['POST'] is 201
# END POST Scenario

# START GET Scenario
@given(u'I Set GET posts api endpoint "{id}"')
def step_impl(context,id):
    api_endpoints['GET_URL'] = api_url+'/posts/'+id
    print('url :'+api_endpoints['GET_URL'])

#You may also include "And" or "But" as a step - these are renamed by behave
to take the name of their preceding step, so:
@when(u'Send GET HTTP request')
def step_impl(context):
    # sending get request and saving response as response object
    response = requests.get(url=api_endpoints['GET_URL'],
headers=request_headers) #https://jsonplaceholder.typicode.com/posts
    # extracting response text
    response_texts['GET']=response.text

```

```

# extracting response status_code
statuscode = response.status_code
response_codes['GET'] = statuscode

@then(u'I receive valid HTTP response code 200 for "{request_name}"')
def step_impl(context,request_name):
    print('Get rep code for '+request_name+':'+
str(response_codes[request_name]))
    assert response_codes[request_name] is 200

@then(u'Response BODY "{request_name}" is non-empty')
def step_impl(context,request_name):
    print('request_name: '+request_name)
    print(response_texts)
    assert response_texts[request_name] is not None
# END GET Scenario

#START PUT/UPDATE
@given(u'I Set PUT posts api endpoint for "{id}"')
def step_impl(context,id):
    api_endpoints['PUT_URL'] = api_url + '/posts/'+id
    print('url :' + api_endpoints['PUT_URL'])

@when(u'I Set Update request Body')
def step_impl(context):
    request_bodies['PUT']={"title": "foo","body": "bar","userId": "1","id":
"1"}

@when(u'Send PUT HTTP request')
def step_impl(context):
    # sending get request and saving response as response object # response
= requests.post(url=api_endpoints['POST_URL'], headers=request_headers)
#https://jsonplaceholder.typicode.com/posts
    response = requests.put(url=api_endpoints['PUT_URL'],
json=request_bodies['PUT'], headers=request_headers)
    # extracting response text
    response_texts['PUT'] = response.text
    print("update response :" + response.text)
    # extracting response status_code
    statuscode = response.status_code
    response_codes['PUT'] = statuscode
#END PUT/UPDATE

#START DELETE
@given(u'I Set DELETE posts api endpoint for "{id}"')
def step_impl(context,id):
    api_endpoints['DELETE_URL'] = api_url + '/posts/'+id
    print('url :' + api_endpoints['DELETE_URL'])

@when(u'I Send DELETE HTTP request')
def step_impl(context):
    # sending get request and saving response as response object
    response = requests.delete(url=api_endpoints['DELETE_URL'])
    # response = requests.post(url=api_endpoints['POST_URL'],
headers=request_headers) #https://jsonplaceholder.typicode.com/posts
    # extracting response text
    response_texts['DELETE'] = response.text

```

```

print("DELETE response :" + response.text)
# extracting response status_code
statuscode = response.status_code
response_codes['DELETE'] = statuscode
#END DELETE

```

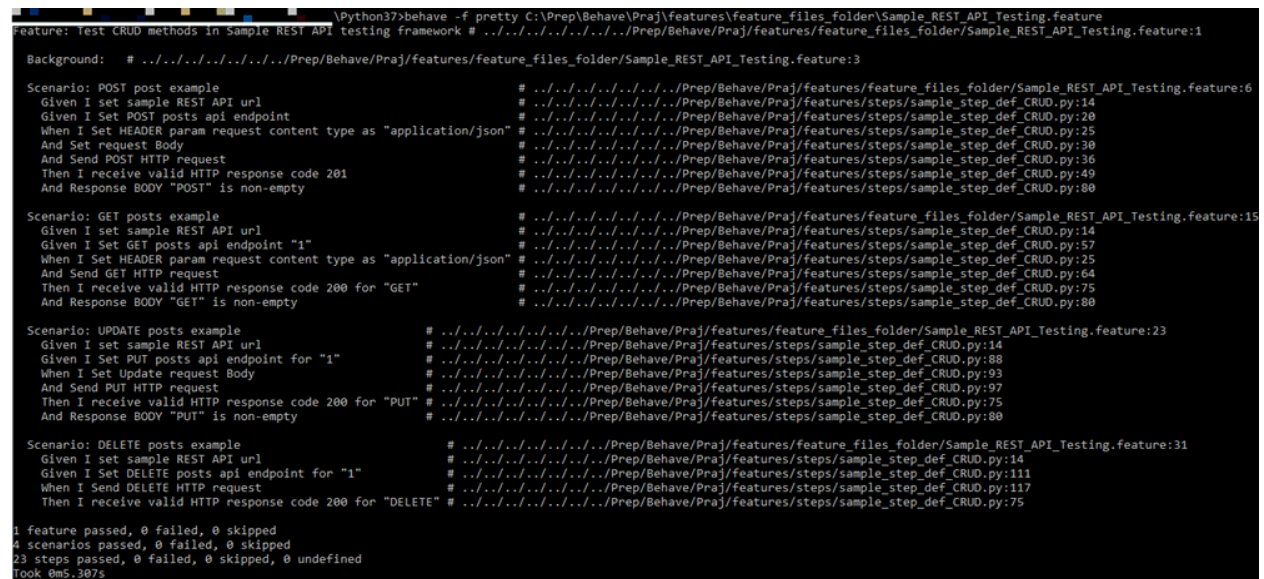
Running the Tests

Now, we are done with our test script development part, so let's run our tests:

Execute the following command on command prompt to run our feature file

C:\Programs\Python\Python37>**behave -f pretty C:\<your project path>\features\feature_files_folder\Sample_REST_API_Testing.feature**

This will display test execution results as follows:



```

Python37>behave -f pretty C:\Prep\Behave\PraJ\features\feature_files_folder\Sample_REST_API_Testing.feature
Feature: Test CRUD methods in Sample REST API testing framework # ...../Prep/Behave/PraJ/features/feature_files_folder/Sample_REST_API_Testing.feature:1
Background: # ...../Prep/Behave/PraJ/features/feature_files_folder/Sample_REST_API_Testing.feature:3
Scenario: POST post example # ...../Prep/Behave/PraJ/features/feature_files_folder/Sample_REST_API_Testing.feature:6
  Given I set sample REST API url # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:14
  Given I Set POST posts api endpoint # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:20
  When I Set HEADER param request content type as "application/json" # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:25
  And Set request Body # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:30
  And Send POST HTTP request # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:36
  Then I receive valid HTTP response code 201 # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:49
  And Response BODY "POST" is non-empty # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:80
Scenario: GET posts example # ...../Prep/Behave/PraJ/features/feature_files_folder/Sample_REST_API_Testing.feature:15
  Given I set sample REST API url # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:14
  Given I Set GET posts api endpoint "1" # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:57
  When I Set HEADER param request content type as "application/json" # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:25
  And Send GET HTTP request # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:64
  Then I receive valid HTTP response code 200 for "GET" # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:75
  And Response BODY "GET" is non-empty # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:80
Scenario: UPDATE posts example # ...../Prep/Behave/PraJ/features/feature_files_folder/Sample_REST_API_Testing.feature:23
  Given I set sample REST API url # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:14
  Given I Set PUT posts api endpoint for "1" # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:88
  When I Set Update request Body # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:93
  And Send PUT HTTP request # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:97
  Then I receive valid HTTP response code 200 for "PUT" # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:75
  And Response BODY "PUT" is non-empty # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:80
Scenario: DELETE posts example # ...../Prep/Behave/PraJ/features/feature_files_folder/Sample_REST_API_Testing.feature:31
  Given I set sample REST API url # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:14
  Given I Set DELETE posts api endpoint for "1" # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:111
  When I Send DELETE HTTP request # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:117
  Then I receive valid HTTP response code 200 for "DELETE" # ...../Prep/Behave/PraJ/features/steps/sample_step_def_CRUD.py:75
1 feature passed, 0 failed, 0 skipped
4 scenarios passed, 0 failed, 0 skipped
23 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m5.307s

```