S.NO.	Behavior Driven Development	Test Driven Development
01	Behavior Driven Development is a development technique which focuses more on a software application's behavior.	Test Driven Development is a development technique which focuses more on the implementation of a feature of a software application/product.
02.	In BDD the participants are Developers, Customer, QAs.	In TDD the participants are developers.
03.	Mainly it creates an executable specification that fails because the respective feature doesn't exist, then writing the simplest code that can make the specification pass and as a result we get the required behavior implemented in the system.	Mainly it refers to write a test case that fails because the specified functionality doesn't exist and after that update the code that can make the test case pass and as a result we get the feature implemented in the system.
04.	Its main focus is on system requirements.	Its main focus is on unit test.
05.	In BDD the starting point is a scenario.	In TDD the starting point is a test case.
06.	It is a team methodology.	It is a development practice.
07.	Here language used to write behavior/scenarios is simple English language.	Here language is used is similar to the one used for feature development like programming language.
08.	In BDD collaboration is required between all the stakeholders.	In TDD collaboration is required only between the developers.
09.	It is a good approach for project development which are driven by user actions.	It is a good approach for projects which involve API and third-party tools.
10.	Some of the tools used are Cucumber, Dave, JBehave, Spec Flow, Concordian, BeanSpec etc.	Some of the tools used are JBehave, JDave, Cucumber, Spec Flow, BeanSpec, FitNesse etc.

Advantages of TDD :

- **You only write code that's needed –**
  Following the principles, you've got to prevent writing production code when all of your tests pass. If your project needs another feature, you would like a test to drive the implementation of the feature. The code you write is the simplest code possible. So, all the code ending up within the product is really needed to implement the features.
- **More modular design –**
  In TDD, you consider one microfeature at a time. And as you write the test first, the code automatically becomes easy to check. Code that's easy to check features a clear interface. This leads to a modular design for your application.
- **Easier to maintain –**
  Because the different parts of your application are decoupled from one another and have clear interfaces, the code becomes easier to take care of, you'll exchange the implementation of a microfeature with a far better implementation without affecting another module. you'll even keep the tests and rewrite the entire application. When all the tests pass, you're done.
- **Easier to refactor –**
  Every feature is thoroughly tested. you do not get to be afraid to form drastic changes because if all the tests still pass, everything is ok. Now, is extremely important because you, as a developer, improve your skills each and each day. If you open the project after six months of performing on something else, most likely, you will have many ideas on the way to improve the code. But your memory about all the various parts and the way they fit together isn't fresh anymore. So, making changes is often dangerous. With an entire test suite, you'll easily improve the code without the fear of breaking your application.
- **High test coverage –**
  There's a test for each feature. This leads to a high test coverage It develops gain confidence in your code.
- **Tests document the code –**
  The test code shows you ways your code is supposed to be used. As such, it documents your code. The test code is a sample code that shows what the code does and the way the interface has got to be used.
- **Less debugging –**
  How often have you ever wasted each day to seek out a nasty bug? How often have you copied a mistake message from Xcode and looked for it on the web.

Disadvantages of TDD :

- **No silver bullet –**
  Tests help to seek out bugs, but they can not find bugs that you simply introduce within the test code and in implementation code. If you haven't understood the matter you would like to unravel, writing tests most likely doesn't help.
- **slow process –**
  If you begin TDD, you'll get the sensation that you simply need an extended duration of your time for straightforward implementations. you would like to believe the interfaces, write the test code, and run the tests before you'll finally start writing the code.
- **All the members of a team got to do it –**
  As TDD influences the planning of code, it's recommended that either all the

members of a team use TDD or nobody in the least. additionally, to the present, it's sometimes difficult to justify TDD to the management because they often have the sensation that the implementation of latest features takes longer if developers write code that will not find themselves within the product half the time. It helps if the entire team agrees on the importance of unit tests.

- **Tests got to be maintained when requirements change –**
  Probably, the strongest argument against TDD is that the tests need to be maintained because the code has got to. Whenever requirements change, you would like to vary the code and tests. But you're working with TDD. this suggests that you simply got to change the tests first then make the tests pass. So, actually, this disadvantage is that the same as before when writing code that apparently takes an extended time.y takes a long time.

BDD

- BDD stands for Behavior Driven Development. BDD is an extension to TDD where instead of writing the test cases, we start by writing a behavior. Later, we develop the code which is required for our application to perform the behavior.
- The scenario defined in the BDD approach makes it easy for the developers, testers and business users to collaborate.
- BDD is considered a best practice when it comes to automated testing as it focuses on the behavior of the application and not on thinking about the implementation of the code.
- The behavior of the application is the center of focus in BDD and it forces the developers and testers to walk-in the customer's shoes.

Process Of BDD

- The process involved in BDD methodology also consists of 6 steps and is very similar to that of TDD.
- **1) Write the behavior of the application:** The behavior of an application is written in simple English like language by the product owner or the business analysts or QAs.
- **2) Write the automated scripts:** This simple English like language is then converted into programming tests.
- **3) Implement the functional code:** The functional code underlying the behavior is then implemented.
- **4) Check if the behavior is successful:** Run the behavior and see if it is successful. If successful, move to the next behavior otherwise fix the errors in the functional code to achieve the application behavior.
- **5) Refactor or organize code:** Refactor or organize your code to make it more readable and re-usable.
- **6) Repeat the steps 1-5 for new behavior:** Repeat the steps to implement more behaviors in your application.

- # Example Of Behavior Implementation In BDD
- Let’s assume that we have a requirement to develop a login functionality for an application which has username and password fields and a submit button.
- **Step1:** Write the behavior of the application for entering the username and password.
- **Scenario:** Login check
- **Given** I am on the login page
- **When** I enter "username" username
- **And** I enter "Password" password
- **And** I click on the "Login" button
- **Then** I am able to login successfully.
- **Step2:** Write the automated test script for this behavior as shown below.

```
@RunWith(Cucumber.class)
public class MyStepDefinitions {

@Steps
LoginPage loginPage;

@Steps
HomePage hp;

    @Given("^I am on the login page $")
    public void i_am_on_the_login_page(){
        loginPage.gotoLoginPage();
    }

    @When("^I enter \"([^\"]*)\" username$")
    public void i_enter_something_username(String username) {
      loginPage.enterUserName(username);
    }

    @When("^I enter \"([^\"]*)\" password$")
    public void i_enter_something_password(String password) {
       loginPage.enterPassword(password);
    }

    @When("^I click on the \"([^\"]*)\" button$")
    public void i_click_on_the_submit_button(String strArg1) {
    hp =   loginPage.submit();
    }

    @Then("^I am able to login successfully\.$")
    public void i_am_able_to_login_successfully() {
      Assert.assertNotNull(hp);
    }
}
```

- **Step3:** Implement the functional code (This is similar to the functional code in TDD example step 3).

```
public class LoginPage{
```

```
String username = "";
String password = "";
//store username
public void enterUserName(String username){
this.username = username;
}
//store password
public void enterPassword(String password){
this.password = password;
}
//match username and passowrd in db and return home page
public HomePage submit(){
if(username.existsInDB()){
String dbPassword = getPasswordFromDB(username);
if(dbPassword.equals(password){
Return new HomePage();
}
}
}
```

- **Step4:** Run this behavior and see if it is successful. If it is successful, then go to step 5 otherwise debug the functional implementation and then run it again.
- **Step5:** Refactoring the implementation is an optional step and in this case, we can refactor the code in the submit method to print the error messages as shown in step 5 for the TDD example.

```
//match username and passowrd in db and return home page
public HomePage submit(){
if(username.existsInDB()){
String dbPassword = getPasswordFromDB(username);
if(dbPassword.equals(password){
Return new HomePage();
}
else{
System.out.println("Please provide correct password");
return;
}
}
else{
System.out.println("Please provide correct username");
}
```

- **Step6:** Write a different behavior and follow steps 1 to 5 for this new behavior.

Gherkin Keywords:
The primary keywords are:

- `Feature`
- `Rule` (as of Gherkin 6)
- `Example` (or `Scenario`)
- `Given`, `When`, `Then`, `And`, `But` for steps (or `*`)

- `Background`
- `Scenario Outline` (or `Scenario Template`)
- `Examples` (or `Scenarios`)

There are a few secondary keywords as well:

- `"""` (Doc Strings)
- `|` (Data Tables)
- `@` (Tags)
- `#` (Comments)

Features of Cucumber:

1)Feature: Search available trains

Scenario:

Given: The website https://www.irctc.co.in/nget/train-search is given.

When: Enter From Station

Then: Enter To station

And: Click Valid Date.

And: click on Valid class

And: Click on Quota.

And : Select Valid Concession

And: Click on Search Button

Then : It will Show all available trains with that Entered Inputs.

2)Feature: Book Ticket On IRCTC Website

Scenario: To book ticket

Given: Visit IRCTC website

When: Search the available Trains

Then : Enter Valid UserName And Password

Then :Enter Visible captcha

Then: Click on Booking with OTP

Then : Click on Submit

Then: Do payment.

Then: It should Show Ticket is booked.

3) Feature: Seat Availability

Scenario: To check seat is available or not

Given: Visit IRCTC Website

When: Search the available seats

Then: Enter Valid Train No.

And: Enter journey Date

And: Enter Source Station

And :Enter Destination station

And : Enter Class

And: Enter Quata

Then: It ask to Enter Valid ans

Then: It will show available seats with train name and number

When:

Then: