**Data Driven Testing In Agile Industry**

A product can speak for itself if it's engineered to support its customers' needs with the highest quality. Quality therefore is not the responsibility of one but of many!  
  
However, in an organization this responsibility is rarely a function of one team, rather should be a collective and collaborative exercise. Most organizations find this hard unless there is effective communication across, which is hard to come by. This problem directly or indirectly reflects the way a product works. Agile solves this!

Agile is a methodology that supports an efficient software through iterative development. The philosophy allows frequent inspection on figuring out what needs to change, and upholds the willingness to adapt to that change quickly! It is a process that encourages collaboration between teams to layout requirements and solutions though a set of engineering best practices.

Agile, adopts the policy of embracing change promising frequent delivery with high-quality software, improved team performances and technical excellence. Agile also takes into consideration the business approach to properly align development with customer needs and company goals. These policies are not just confined to software development rather is a manifesto on the way our industry works as of date. Needless to say, this is accompanied with a lot of changes and challenges. One of the major challenges we still face as of today, is that of maintaining quality with such frequent delivery.

Quality in a service or product is not about what you put into it. It is what your client or customer gets out of it. - Peter Drucker

Now that we have established how important ‘Quality’ is, we need to explore solutions that give us that level of quality. ‘Testing’ therefore, was and will remain one of the most important standalone branches in the software development process that guarantees quality. This is one field that takes as much heat as the development itself as and when deadlines get narrower.

Testing is of many types and is broadly dependent on the software product or application type being built. However, one aspect remains the same - testing often and testing better. While this can be done in many automated ways, testing applications through data has remained the most widely accepted choice.

The information throughout will help you seek knowledge on what we call Data-Driven Testing (DDT).

**What Is Data-Driven Testing?**

Imagine a scenario, where a tester has to test a workflow in an application which has multiple entry points for inputs. The test engineer is likely to input a permutation of numbers and execute the application only then to realise that it might fail for a different permutation of numbers.  
  
To keep doing this over months manually and to maintain records on it is not just cumbersome and confusing, it’s sheer pain! For all that the software industry stands for, we have got to have better solutions for such problems. Don’t you agree?

If we could have all the usable test data stored or documented at a single place (a storage facility), it would save us a lot of time that would otherwise be spent in creating different test cases using different types of test data. And if we were to find a way to build an automated process that uses data in this file to run multiple times without any manual effort, we have the perfect solution

This is exactly what Data-Driven testing tries to achieve.

The concept is centered around separating the test data from the test script and this cannot get simpler than it already is! First, a tester would ideally document all input values in a storage file (could be a .csv, .xls, .xml and so on) that covers best case, worst case, positive and negative test scenarios; and second, we would have a test script developed that uses this data as variables (by substituting values in the scripted place holders) while it runs iteratively until for all data sets.

Data-Driven testing, therefore conceptualizes an automated framework where testing is triggered on a set of data stored in a storage facility (files, databases). This framework resolves the lengthy and time-consuming process of creating individual tests for each data set.

DDT is a methodology where a sequence of test steps (structured in test scripts as reusable test logic) are automated to run repeatedly for different permutations of data (picked up from data sources) in order to compare actual and expected results for validations.

**In this process, four important operations are observed:**

1. Collecting different sets of test data in a storage facility like a file or a database
2. Creation of scripts that is capable of reading this data, passing it through required layers of the script; which then automatically triggers simulation of other action items
3. Storing the retrieved results (from point 2) and then retrospecting the results (should it throw errors) in terms of what was expected and what was actually obtained
4. Continued testing with the next set of input data

**Data-Driven Testing: Use Case Example**

To explain what we just spoke about, consider the following examples.

**Example 1: Login Panel**

A basic Login Panel (post a sign up) consists of an email (could be business email for business applications) and a password. Allowing only a registered user is critical here, as the login feature is built for security reasons whose workability is to block unauthorized users from accessing data within the platform/product. Some of the best cases one can think about authenticating the email and password could be –

* Checking if the email id has the symbol @ or not
* Checking if the entered password meets the password entered at the time of registration/Sign-up process
* Checking if both input field types have matched data or not

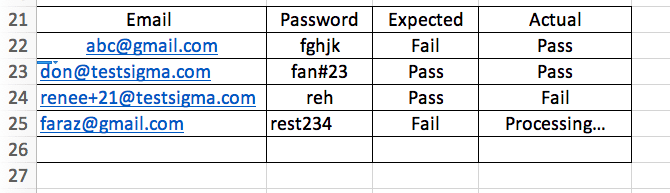
And to check these individual cases one would have to input different variables every time a script is run to test the authentication/login process. This is where a data-driven testing framework can be useful!

Writing these different variables in a storage file and calling these values individually in a functional test script is the most efficient way of approaching this problem.

A test script could look something like this -

pip install selenium  
from selenium import webdriver  
driver = webdriver.Chrome()  
def site\_login():  
driver.get (“URL”)  
driver.find\_element\_by\_id(“ID”).send\_keys(“username”)  
driver.find\_element\_by\_id (“ID”).send\_keys(“password”)  
driver.find\_element\_by\_id(“submit”).click()  
#if you are using FB then,  
def fb\_login():  
driver.find\_element\_by\_id(“email”).send\_keys(‘fakeemail@crossbrowsertesting.com’)  
driver.find\_element\_by\_id(“pass”).send\_keys(“fakepassword1”)  
driver.find\_element\_by\_id(“loginbutton”).click()  
from selenium.webdriver.support import expected\_conditions as EC  
from selenium.webdriver.common.by import By  
from selenium.webdriver.support.ui import WebDriverWait WebDriverWait(driver, until(EC.title\_contains("home"))

A test file could look something like this -



Please note that in this automated framework, test scripts that test the functional workflow are kept separate from the test values.

**Example 2: Testing an agreed upon format in a CSV upload**

In many instances, we use the ‘upload a file’ feature in an application. One of the most important requirements in an ‘upload a file’ feature is to test for the format/template of the file that is being uploaded. Assuming that we use a .csv file to store data, then the test should ideally help validate the data-type in every cell of the csv file.

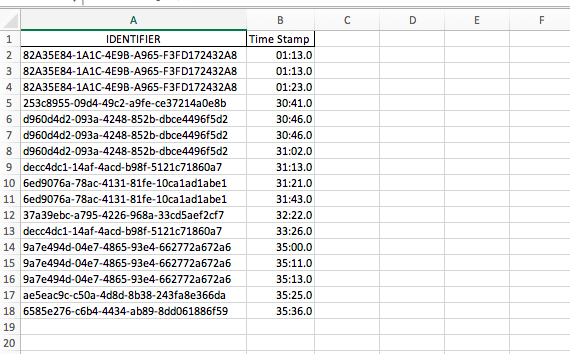
The data in the file is expected to be picked up by an application that processes it for a specific use case (such as, to power a graph!). Here, testing the content-type in every cell according to a certain type takes precedence.

Most importantly, the test should be around checking the format of supported and unsupported files, thereby reporting them in case they don’t meet requirements. Supported files should be enabled and uploaded without any data loss. For instance, if the upload file is text, an image upload should not be allowed. Or if the upload file should have just two columns with a specific header, then it should not allow any other file format.

Let’s say the requirement suggests we upload a file with two columns, that have a single header each. The rest of the content in the file should not have any special characters in between a cell content.

Should the file format be correct, the application is triggered to read the right data in every cell to be processed further. This test has to run for all cells, across all rows and columns. This is where DDT can help!

A sample template could look something like -



All of these mentioned above can be tested using a data-driven automated framework.

Creating a test script that creates a placeholder to accommodate the data (content that’s normally present in a file upload) from the test file, and then having different values called in this test script will execute this test framework seamlessly!

This being explained, let’s try to understand the applications coverage that they make happen.**your tea**

**How Data-Driven Testing Works**

Data driven testing is implemented by following the below steps :

* Data driven testing fetches the input data from the data sources like xls, csv, xml files.
* It enters the input data into the web application by means of automated test scripts.
* It compares the actual results with expected output.
* Same tests are reused with another variation in data.

**How To Implement Data-Driven Testing**

Implementing data driven test automation is slightly different from generic test automation. In a generic test you include a sample data in the test script itself whereas in data driven testing you connect your test to an external data source. Data sources can be of various types like excel files, csv files, xml and even fully featured databases like MySQL.

There are three steps by which the data driven testing is implemented :

* Choosing the data source
* Connecting the data source
* Assessing the result

Top of Form

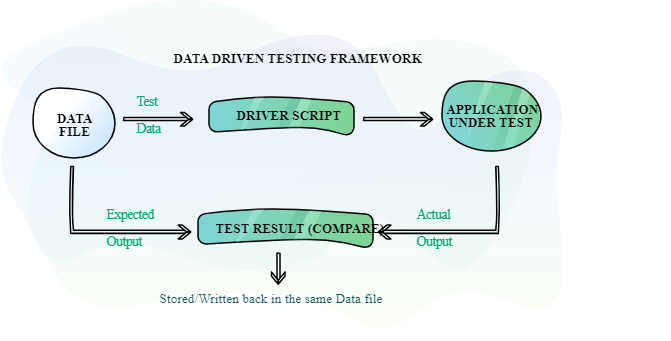
Bottom of Form

**Architecture Of DDT In An Automated Framework**

DDT framework is an automation testing framework that allows you to use a single test script to validate a test case against different types of test data. The test data corresponding to positive as well as negative testing is stored in a file, and the test script uses all these values as inputs to execute the tests. Therefore, the framework provides re-usable logic that improves test coverage.

Since the agenda is based on test data in an automated framework, we would ideally focus on helping you understand how to input data and what to expect as the output data through a type of automated framework explained above; most importantly, how do we go about structuring this data.

Take a look at the flow-diagram below. This replicates a typical Data-Driven Testing framework.



**Data File And The Test Data**

A Data File is the first point of input for any type of DDT framework. A typical Data File is loaded with test data that is inclusive of scenarios such as positive test cases, negative test cases, exception throwing and exception handling test cases, min-max limit cases and not to forget the permutations of the data that contributes for appropriate data coverage.

Now that we have knowledge on the test data and the Data File, it is rendered as ‘ready’ only if one can parse this data through the Driver Script depending on the requirement of the application that is being tested. These data sets can be stored in data storage files such as an excel, xml, json, yml, or even a HDP database.

Sometimes, it’s wise to include Hash (key, value pairs), Arrays, Data Tables to structure big data that is to be tested. Testsigma enables all these capabilities in a single module.

The test file that contains the data does not just contain input values, but may also contain output or expected results that relates to successful running of the application under test. The idea of having the output results in the same file is to make sure that the results obtained on testing an application are in par with the expected ones.

This is technically termed as analyzing Actual Results vs Expected Results.

**Driver Script**

If you are using a codeless data driven testing solution like Testsigma, you don’t need to manage this. This piece is like any other script whose usage matches that of an executable file. Driver Script, as the name indicates, is a piece of code that replicates the test case of an application.It reads data from the data files such that they are used in the test scripts to execute corresponding tests on the application. The Driver script then outputs results on execution.

In other words, its structure contains placeholders for variables (test data) picked up from the Test File. The output that the script generates is compared with the ‘expected results.

A driver script is often a simple piece of code written efficiently. A driver script in a DDT usually contains the application code and the implementation code that works together.

Overall, DDT revolves around how efficiently an application can be tested. Technically, it’s about how the test data works with the test script to bring out expected results.

**Data-Driven Scripts are similar to application specific scripts (pyspark script, JavaScript) programmed to accommodate variable datasets.**

* **Automated Scripting with dynamic variables**Test scripts are often hard coded because they are created under the impression that they are to run ‘once’ for a certain set of data. In DDT, everything is tested dynamically with different variables of data sets. Therefore, we need modified scripts that aren’t just hardcoded (with static data) but also are capable of handling dynamic data and its behavior when the application runs. It is upto the capability of the automation tester to design scripts with the right balance of both.
* **Duplication of the Test Design**This is often tricky, but doable! The idea is to have the same workflow followed in manual testing duplicated or reproduced in the automation workflow. This will allow the same test design that a manual tester would follow, be automated by an automation process. Comparing the two, manual testing cushions the way an application works by manually triggering the next process in real-time; whereas, automation workflow tests this transition through code which should work seamlessly without any intervention.

This brings us to the last part of the architectural design that is to compare results.

**Expected Vs Actual Results**

DDT framework allows testers and stakeholders to understand the performance of the existing design in the application.

In other words, when a system gives out an output, it becomes necessary that the output needs validation. This validation is achieved by comparing the actual results with that of the expected ones. If there are any differences, the root cause is found and evaluated to be corrected. Further, these corrections should not affect the expected workflow of the product. For this, a valid feedback loop is enabled in the organization’s process which directs these corrections to the corresponding development teams.

Please note that, in this process, there could be new test cases that are to be added in order to validate an instance more thoroughly. In such cases, the Data File and the Driver Script are modified to fit requirements.

With an efficient architecture as discussed above, there are many advantages DDT framework enables. This is discussed in the next section.

**Advantages Of Data-Driven Testing**

Data-Driven testing offers many advantages. Some of them are:

* We all know how important [Regression testing](https://testsigma.com/regression-testing) is.  
  Regression testing revolves around having a bunch of test cases re-executed to ensure that the additions made to the software for the latest deployment did not affect the overall functionality of the software. Data-Driven testing (DDT) makes this process happen faster adding to its efficiency. Since DDT makes use of multiple sets of data values for testing, regression tests can run from end-to-end workflow for multiple data sets.
* DDT enables clarity. It allows clear logical separation of the test cases or test scripts from the test data that is being used. In other words, they don’t have to re-alter their test cases over and over again for different sets of test values. This will enable actions of the application (test data) and functions to be dealt with (test scripts) separately as two different entities that can be reused in different test cases and scenarios.
* Change in either the test script or the test data will not affect the other. Main reason being that they are kept and maintained separately. If a tester wants additional test data to be added, he can do so without disturbing a test case. Likewise, if a programmer wants to change the code in the test script, he can do so without worrying about the test data.

**Disadvantages Of DDT**

DDT has no jarring problems or cons as such. The cons listed below are more of limitations rather than disadvantages.

* In a cycle where you are testing data continuously, the ‘right data set’ is hard to come by! Data validations are time consuming processes and the quality of such tests are dependent on the automation skills of the implementing teams.
* Although DDT maintains separate test scripts and test data documents, the test code that’s written to read this data is slightly complicated. The programmer needs to keep in mind that the test module should test every data row in the data set before it could kill the job. This is not the case for tools that automate.
* For a tester, debugging errors even in a DDT environment may be slightly tough due to the lack of exposure he/she has on a programming language. He/she would generally not be able to identify logical errors while a script runs and throws an exception. Sometimes there might be a need to entirely learn a new language from the scratch! However, automated testing tools do not require much programming skills. For instance, Testsigma supports NLP which is easier to use, compared to traditional programming languages.
* Increased documentation – Since DDT approaches testing in a modular way, there would be an increasing need to have all of this documented to make it easier for all team members/new joinees to know the structure/workflow. Such documentation would be around script management, its infrastructure, results obtained at different levels of testing and so on.

**Best Practices Of Data-Driven Testing**

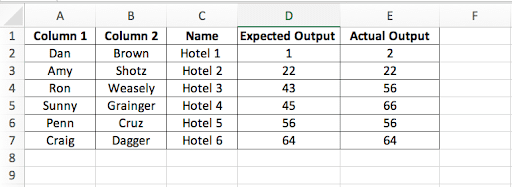
To make Data-Driven Testing as efficient as it can get, here is a checklist of best practices you might want to take a note of:

* **Testing with Positive and Negative Data**Testing positives is a mandatory rule everyone follows, but testing the negatives are also equally as important. A system’s performance is gauged on its ability to handle exceptions. These exceptions can occur because of a worst-case scenario that was reproduced in the system at some point. An efficient system should be designed to handle these exceptions well. In other words, these exceptions are a bunch of negative test cases. Therefore, the test cases that one writes or lays down should cover both, positives and negatives that a system should be capable of handling.
* **Driving Dynamic Assertions**Driving dynamic assertions, that augment the pre-test values/scenarios into the latest ones are extremely necessary. Verifications get critical during code revisions and new releases. At this juncture, it’s important to have automated scripts that can augment these dynamic assertions; i.e., include what was previously tested into the current test lines.
* **Checkpoints where manual effort can be neutralized**Avoiding unnecessary manual interventions for the purpose of triggering a workflow to continue is a must. When we have a workflow with multiple navigational or re-directional paths, it’s best to write a script that can accommodate all of this; importantly because a manual trigger is never an efficient way to test a navigational flow. Therefore, it is always a best practice to have the test-flow navigation coded inside the test script itself.
* **Perspective of the test cases**Perspective should also be considered! This is rather an insightful testing practice than a logical one. If you merely are interested in checking the workflow, you run basic tests to avoid a break or an exception that is anticipated somewhere in the process. But having the same tests extended for additional capabilities, such as security and performance, will provide fool-proof coverage into the existing network of the design. For instance, you can test performance of the same workflow by introducing data that meets max limits of a product – by observing the latency of the load, get-pull from APIs and so on.

**How Does Data-Driven Testing Work?**

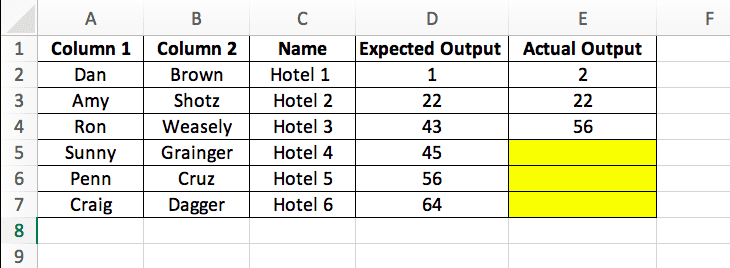
Complimentary to the architecture as explained in the previous section, DDT can comfortably be tagged as a test automation framework in an agile environment. DDT accommodates both positive and negative test cases in a single flow of test.

* Test data is stored in a columnar distribution, a table or a structure that mimics a spreadsheet format. This storage canvas is a test table/data file. This file contains all inputs to different test scenarios. It also contains values tagged as ‘expected output’ in a separate column.  
    
  Assume that we have the following data table recorded that only has positive test cases recorded



* A Script is written to read the data from the data file, such that the test input is picked up from every cell of the data file, and substituted as the variable in the run-time flow. This is called a **Driver Script**. Now that there are tools that automate this process, a Driver Script does not need scripting as such. Based on the tool’s specifications, all you’ll have to do sometimes is connect data sets to the test cases.  
    
  For example, Testsigma and Katalon build tools that enable this. Testsigma’s intuitive UI enables you to create Test Data Profiles outside your scripts and manage large data sets(positive and negative) without much effort.
* The driver script is written in a way where it’s enabled to ‘read’ a file, pick up a variable from the data file in a predefined format (for instance, read column A, cell 2A; column B, cell 2B; Name C, cell 3C). Perform a logic or run this through an application and ‘write’ the output in column E.  
    
  For example, in the above data file, assuming that the script is drafted fairly straightforward, will open the file > read ‘Dan’ > read ‘Brown’ > read ‘Hotel 1’ > Run through code at runtime > outputs a value ‘2’ which is fed in column E

If the script has run 3 times then the table would look like this:



The highlighted cells are blank as the assumption here is that the computation was incomplete.

Post this, once all the tests run, the outputs recorded in Columns D and E are compared. If they are in good agreement, the design is passed to production, else feedback that lists problems with an RCA (Root Cause Analysis) must be shared across.

**Example Of DDT With Its Implementation Structure -**

Let us see an example for Data Driven Testing.

Consider the Login Page of a Hotel Reservation website. A pseudo workflow could be something as follows:

1. A test data file is created as TestData\_HotelReservation.csv (Comma Separated Values)
2. This file contains inputs given to the driver script and expected results in a table
3. The driver script for above data file will be,  
   data = open(‘TestData\_HotelReservation.csv’).read()  
   lines = data.splitlines()
4. Steps performed for above driver scripts are as follows:
   * Read Value1
   * Read Value2
   * Read Operator
5. Calculate the result using an operator on Value1 and value2
6. Finally, compare the expected result with the actual result

**Data Driven Testing With Testsigma**

Testsigma enables data driven testing for below storage types:

* Data tables created in Testsigma
* Excel file
* json file

**The excel and json files can be easily imported into Testsigma. A test case in Testsigma can then be configured to read data from these data files by toggle of a button.**

**Types Of Data Driven Testing**

As you already know, DDT uses classes of iterative data to cover portions of application. The method is staight-forward where the test scripts (programmed in a scripting language) are executed for test data stored in a test file.

At the core of this is the way the scripting language supports the data file. This is where DDT varies slightly; i.e., based on the types of data files it uses.

Some of them are as follows -

* Comma-separated values (CSV) files
* Excel sheets
* Database tables
* Script arrays
* Table variables

**Importance Of Data Driven Testing**

Once you get a hang of how a product application should work, you will understand the nuances of handling things in a way that will cover all scenarios, both -positive and negative.

A test engineer at this point will understand the kind of variables to add that will make the data-file holistic. This being present, a script representing a test, runs in loops for all test data in the file. The outputs are compared with verifiable data making the process a success in an automated environment.

Some of the outstanding attributes of this offering are as follows :

* Duplication of test scripts, variables, data inputs are reduced. No one wants to bear the brunt of having to deal with clutter that repeats. DDT with Testsigma enables this.
* There is streamlining of documentation with DDT. This creates code sanity, which is among the best practices as listed in [Edureka](https://www.edureka.co/blog/what-is-sanity-testing/" \t "_blank).
* DDT offers great flexibility in maintaining an application as there is no hard coding of values in the test scripts. Also, because the test data and test scripts are separate. New test data can be introduced as and when needed without any modifications in the test script.
* The Data File and the test scripts are maintained separately. In this case, if there is an error while running a test through the application, the test engineer can pinpoint the fault in the system. In other words, debugging becomes easy and fast
* Therefore, if the application under test is such that it needs to be tested against a large number of data regularly, DDT is an obvious choice.

If a codeless testing tool is used for this automation framework, the coding hassles can be forgotten. When such a tool also provides easy integration with data files, the automated testing becomes as easy as a manual test creation task. Testsigma is one such tool that offers these features and is recommended for implementation of data driven testing .

**Create-A-Data-Driven-Automation-Framework**

Let’s consider a rather interesting concept that leverages DDT to function better - [**Test Automation Framework**](https://testsigma.com/blog/when-to-choose-test-automation-framework-over-manual-testing/).

A framework basically is a set of guidelines that are to be followed in order to glean beneficial results.

In an automation environment, a set of guidelines such as best coding practices, effective test-data handling capabilities, object and class repository treatment and so on - when followed, will lead to better results, less maintenance, increased re-usability. This is the optimum version of supporting an application with DDT.

There are different types of automated frameworks -

1. Linear Scripting
2. The Test Library Architecture Framework.
3. The Data-Driven testing Framework.
4. The Keyword-Driven or Table-Driven Testing Framework.
5. The Hybrid Test Automation Framework

**Keyword Driven Test Automation**

Keyword Driven Test Automation is often regarded as Table Driven Test Automation or Action Word Based Testing.

The framework revolves around dividing the Test Case into four processes -

* **Test Step -**small description of the test step describing the action that would be performed on the Test Object
* **Object of the Test Step -**name of the object/element of a web page on an excel sheet.
* **Action on the Test Object -**is an action item that is going to be performed on the Object. Examples include, Click, Open Browser, Read/Write etc.
* **Data for Test Object -**any value to substitute an Object that performs an action. This could be Username value, Username field and so on.

You can maintain all these categorizations on an excel sheet.

Let’s consider an example of an Online Shopping Store

What you will need in terms of process :

1. To enable Opening a Browser
2. Navigate to URL from a registered domain
3. Access My Account button to change countries
4. Enter Username and Password of standard specifics
5. Enable LogIn and LogOut buttons
6. Redirection to Homepage if loading a child page takes more than 5 seconds.

**Requirements in terms of resources:**

* **Excel Sheet:**This spreadsheet should contain most of the data for Keyword Driven Test which would be used in Test Cases, Test Objects and Actions.
* **Data Sheet:**This Excel file should store the data value needed by the object to perform an action item on it.
* **Object Repository:**The Property file should store the html elements of the web application. This property file will be linked with the Objects used in the test.
* **Keyword Function Library:**In a keyword Driven Framework, function file is critical. It’s maintained to call a function that mimics working of an action.
* **Execution Engine:**Is a test script that contains all the code to drive the test from Excel sheet, linking factors from Function Lib and Properties file.

Some features of Keyword Driven Test Automation are -

* **Reusable Code :**In an application, independent modules that are built accept application specific data. These modules and their corresponding test files can be reused for similar application modules that would be developed in future.
* **All in One Record :**All records are documented in a master copy which is easy to maintain and refer to.
* **Error Correction and Synchronization :**Every module tested would spot errors that need corrections. Further, these corrections should not affect the expected workflow of the product (process is referred to as a feedback loop). Therefore, error correction and synchronization should align well with the feedback loop.

**The Difference Between Keyword Driven Testing And Data Driven Testing**

Automated testing aims at covering large test scenarios. Automated Testing framework supports both - Keyword Driven Testing and Data Driven Testing, although they solve the same objective of optimizing an application.

**Data Driven Testing** refers to a storage file from which variables are copied onto a script that runs automatically several times on different cases; thereafter, storing the actual output in the same file base to compare it with the expected output. The number of data rows in the storage file therefore, is directly proportional to the test cases executed at run time. Ex: Booking and Reservation Functions

On the other hand, **Keyword Driven Testing** enables you to use a keyword to represent an action item. A sequence of keywords, therefore, drives a script. Further, you could use the same set of keywords to build a variety of test scripts. Ex: Real-time machines that operate on time-bound basis

At this point we have a clear understanding on what Data Driven Testing and Keyword Driven Testing actually mean in an automated environment. Let’s take a look at the ‘data’ part of it for a bit. Questions like what is data? - Its origin and so on. If you are already familiar with this piece, skip to the bottom of the page for conclusions and takeaways.

For a detailed comparison between data-driven testing and keyword-driven testing read here.

**What Is Test Data?**

Data that is considered exclusively for usage in tests is test data. It powers a script with inputs and produces outputs by measuring actual results from an application that is being developed.

They are selected to power best case scenarios, worst case scenarios, exceptions and anything in-between. The variable or containers (place holders in the script where these variables are replaced) could be numbers or names or a combination of both. They also can be correlated.

These data sets can be created by the tester, a program, a function, or could be a computerised generation. They are inhabited with characteristics such as re-utility, redundancy and so on.

**What Is Test Data Generation? How Can We Generate It?**

Test data generation is a process of creating a set of test data for testing the adequacy of new or revised software applications. Test data can either be the actual data taken from previous operations or a set of artificial data designed for testing purposes.

Test data is created depending on the testing environment before the test execution begins.

Test data can be generated through the following ways :

* Copy of data from production to testing environment.
* Copy of data from client machines.
* Manual test data generation process.
* Automated test data generation tools, e.g. : SQL data generator, DTM data generator.
* Back-end data injection : Back-end Data Injection provides swift data injection in the system. It enables creation of backdated entries, which is a huge drawback in both, manual as well as automated test data generation techniques.
* Third party tools : e.g. Test sigma test data generation technique.

If test data is generated during the test execution phase, it is quite possible that you may exceed your testing deadline. In order to simplify the test management process, sample data should be generated before you begin test execution.

**Some tips for good data-driven tests:**

* Try to form tests that create their own scenario test data.
* Ensure tests don't make any permanent changes to the database.
* Include tests that cover positive as well as negative test scenarios.
* Create automated test scripts that can easily be modified for various data scenarios.
* Place test data in one storage format for all test cases, be it an excel spreadsheet or a table.
* Refer the test result report generated for any errors in the test execution that arose due to data issues.

**How Does Test Data Generation Work?**

Now that we have an understanding of what test data is, we very well understand and are in terms of the fact that test data is a critical piece of programming or testing an application.

They cannot be random all the time, and therefore, there are places to be looked at to scrape data. Data can be from past operations collected and archived. You can borrow archived data anytime you want. Other ways to gather data are -

* Through brainstorming and manually collecting data
* Mass copy from production environment to staging environment
* Usage of automated test data generation tools
* Using legacy client systems to duplicate or fork data

That clarified now, an obvious question remains, what if one is not technically sound? How does one validate such test data and therefore test cases? This is where BDT comes into play. In other words, Behavior Driven Testing supports management staff with limited technical knowledge.

**What Is Behavior Driven Testing?**

Testing need not be a technical function all the time. In complex systems, there are applications that require logical test data such as dates in terms of duration, discounts and other mathematical calculations, temperature measurements and so on.

Such kinds of data are most often required in real-time operational systems, and areas related to logistics and inventory management. Please note that although we are talking about Software systems/applications, they ultimately serve an industry purpose as exemplified.

The skill set here is inclined more towards logical, rational and research based expertise, rather than tech skills. These kinds of tasks are often palmed off to management personnels or product owners. Needless to say, the test strategy also changes with the changing audience. This is where Behavior Driven Testing comes into play!

Behavior Driven Testing is most often focused on the behavior of users for an application rather than the technical functions of a software. This test takes into consideration business objectives.

The test cases are written in a more natural language which is verbose and easily understood by anyone. This encourages communication across teams with ease.

BDT is designed to enable an agreed upon expectation although there could be different perspectives of Stakeholders and Delivery Teams. It is initiated as a business goal, and translates into features and user stories. These are then approved by the non-tech stakeholders before hitting development.

The objective is to make a business readable and domain specific language that enables you to describe a system’s behavior without having to be involved in how the behavior is to be implemented.