

Learning Objectives

Learners will be able to...

- **Define the different types of bugs**
- **Identify test-design techniques**
- **Define boundary values**
- **Remember the stages of the testing pyramid**
- **Describe the role of testing frameworks**

info

Make Sure You Know

Beginner Javascript

Limitations

In this assignment we only work with manual testing

What is Software Testing?

Defining Software Testing

Software Development is more than the act of writing code. You need to be sure everything is working as expected before release. If not, you may release something that will not be usable at all. Your software can have flaws and glitches. After all, nobody trusts a company that puts out unusable software. Releasing an unstable product could encourage customers to switch to a competitor's product. Also it is much cheaper to fix bugs and problems before the release. That is why we need quality assurance.

Quality Assurance is the process of comparing expected properties and the current product's properties. Expected products properties are also called specifications. Specifications are formalized user (or customer) expectations. Users can introduce errors when formalizing expectations, but we will not cover this topic.

Software Testing is a part of the quality assurance process applied to software development. Specifically, it is the process of comparing software programs with their specifications. The purpose is to check whether the software satisfies the specific requirements, needs, and expectations of the customer. The job of testing is to verify that the system is usable by the customer.

Software Testing is more complex than you might think. Within this course, you are going to cover various methods for ensuring your software is tested thoroughly before it is released.

Defining Bugs

What is a Bug in Software Testing?

In software testing, a software bug can be an issue, error, fault, or failure that causes the software to operate in an unexpected manner. Bugs often occur when developers make any mistakes or errors while developing the product.

The reasons for the occurrence are different: errors in the source code, the program interface, or incorrect operation of the compiler. They are discovered at different stages of the software development process — debugging, beta testing, or even after the release of the product.

These errors manifest themselves in different ways:

- * An error message has appeared, but the program continues to run.
- * The app freezes or crashes without any warning.
- * One of the events occurs while sending a report to the developer.
- * The hardest thing to work with are computer games, which often use the term “crash” to describe an error. It means a critical problem when starting or using the program. When talking about bugs, they often mean graphics failures. For example, if the player “falls into the textures.”

Bug Classification

The point of view of users often does not align with the opinion of programmers. Users think there was just a failure, “the application stopped working.” The coder, on the other hand, will have a headache when determining the source of the problem. After all, an error in the program probably manifests itself only on specific hardware or when combined with other software (often with antiviruses).

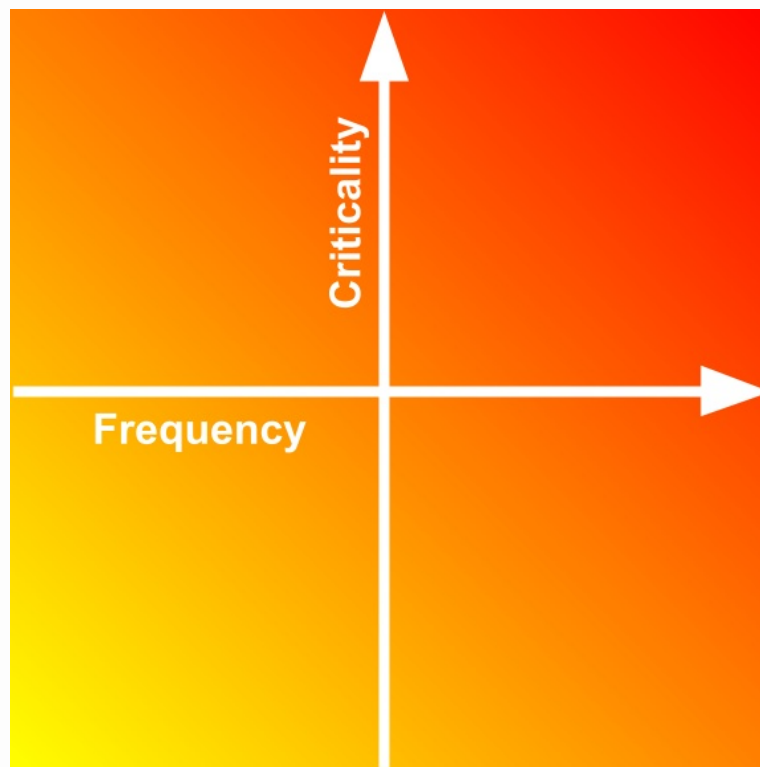
To help differentiate the different types of errors, bugs are divided into categories depending on their **criticality**:

- minor
- major
- showstopper

Showstoppers indicate a critical software or hardware problem, due to which the software loses its functionality by almost 100%. For example, it is not possible to log in using a login-password or the “Next” button has stopped working. Therefore, such errors are given the highest priority.

There is also a division of errors according to their **frequency**. The easiest bugs to fix are those that occur with the greatest frequency, as you tend to see them regardless of platform, computer hardware, or user action. Bugs become more difficult when they occur less frequently. The cause of these issues may well be buggy RAM or driver errors as opposed to problems with the code itself.

There is a variant when the problem occurs only on the machine of a particular client. Here, you have to either assign an individual to work on a solitary bug, or convince the client to change the computer. It is inefficient to devote so many developer hours on a bug that affects a single person.



heat map that shows bugs with the greatest frequency and criticality are the most severe (top-right corner). The severity decreases with these two attributes (bottom-left corner).

Types of Testing

There are several types of testing. While this course will not cover all of these types of testing in depth, you should be familiar with them as general principles.

Accessibility Testing

Accessibility testing is the practice of ensuring your mobile and web apps are working and usable for users without and with disabilities such as vision impairment, hearing disabilities, and other physical or cognitive conditions.

Acceptance Testing

Acceptance testing ensures that the end-user (customer) can achieve the goals set in the business requirements, which determines whether the software is acceptable for delivery or not. It is also known as user acceptance testing (UAT).

Black Box Testing

Black box testing involves testing against a system where the code and paths are invisible.

End to End Testing

End to end testing is a technique that tests the application's workflow from beginning to end to make sure everything functions as expected.

Functional Testing

Functional testing checks an application, website, or system to ensure it's doing exactly what it's supposed to be doing.

Interactive Testing

Also known as manual testing, interactive testing enables testers to create and facilitate manual tests for those who do not use automation and collect results from external tests.

Integration Testing

Integration testing ensures that an entire, integrated system meets a set of requirements. It is performed in an integrated hardware and software environment to ensure that the entire system functions properly.

Load Testing

This type of non-functional software testing process determines how the software application behaves while being accessed by multiple users simultaneously.

Non-Functional Testing

Non-functional testing verifies the readiness of a system according to nonfunctional parameters (performance, accessibility, UX, etc.) which are never addressed by functional testing.

Performance Testing

Performance testing examines the speed, stability, reliability, scalability, and resource usage of a software application under a specified workload.

Regression Testing

Software regression testing is performed to determine if code modifications break an application or consume resources.

Sanity Testing

Performed after bug fixes, sanity testing determines that the bugs are fixed and no further issues are introduced to these changes.

Security Testing

Security testing unveils the vulnerabilities of the system to ensure that the software system and application are free from any threats or risks. These tests aim to find any potential flaws and weaknesses in the software system that could lead to a loss of data, revenue, or reputation.

Single User Performance Testing

Single user performance testing checks that the application under test performs fine according to the specified threshold without any system load. This benchmark can be then used to define a realistic threshold when the system is under load.

Smoke Testing

This type of software testing validates the stability of a software application. It is performed on the initial software build to ensure that the critical functions of the program are working.

Stress Testing

Stress testing is a software testing activity that tests beyond normal operational capacity to test the results.

Unit Testing

Unit testing is the process of checking small pieces of code to ensure that the individual parts of a program work properly on their own, speeding up testing strategies and reducing wasted tests.

White Box Testing

White box testing involves testing the product's underlying structure, architecture, and code to validate input-output flow and enhance design, usability, and security.

Verification vs. Validation

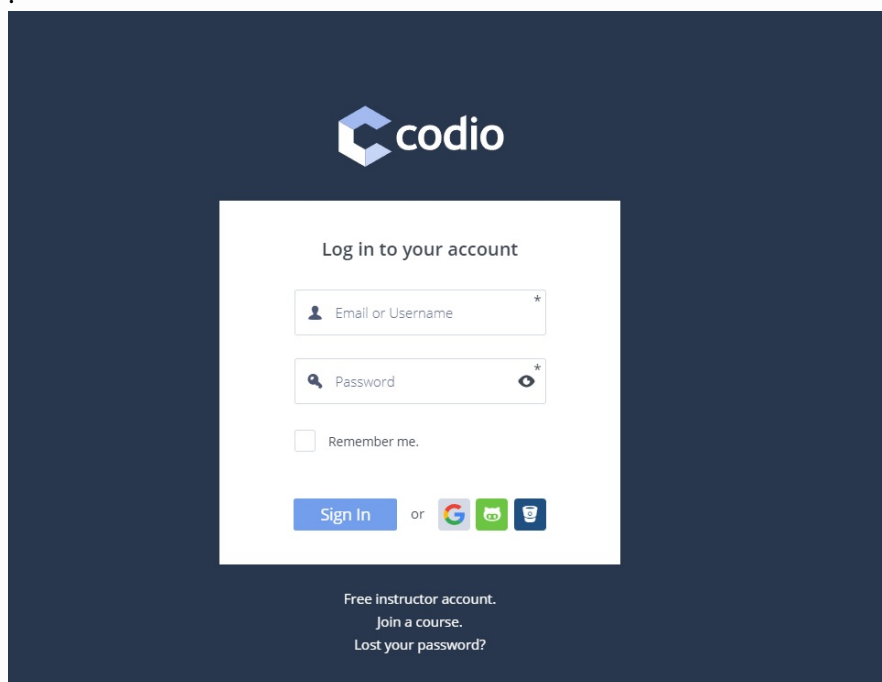
Verification and validation are two concepts that are closely related to testing and quality assurance processes. Unfortunately, they are often confused, although the differences between them are quite significant.

Verification

Verification is the process of evaluating a system to see if the results of the current development phase meet the conditions formulated at the beginning.

Verification uses methods such as reviews, walkthroughs, inspections, debugging, etc. It also uses techniques such as black box testing, white box testing and non-functional testing.

Let's imagine that we have a website. To enter the page of the website, the user must register or log in under his account.

A screenshot of the Codio login page. The page has a dark blue background. At the top center is the Codio logo, which consists of a blue cube icon followed by the word "codio" in white. Below the logo is a white rectangular box containing the login form. The form has the title "Log in to your account" at the top. It features two input fields: the first is labeled "Email or Username" with a person icon and an asterisk; the second is labeled "Password" with a magnifying glass icon, an eye icon for toggling visibility, and an asterisk. Below these fields is a checkbox labeled "Remember me.". At the bottom of the form is a blue "Sign In" button, followed by the word "or" and three social media icons (Google, GitHub, and another). Below the form, there are three links: "Free instructor account.", "Join a course.", and "Lost your password?".

Verification involves checking the fields. All fields must be valid and meet the requirements of the specification. Their number, display and features are determined by the designers who create the layouts. The necessary data is entered into the terms of reference, and in the absence of such, it is necessary to have access to the created layouts. When performing

verification, it is necessary to understand that all fields are initially working, and data can be entered into them according to the displayed symbols and names.

Validation

Validation is the determination of the compliance of the developed software with the expectations and needs of the user and their requirements for the system. In the example above, validation means checking the input data in the information fields, as well as their compliance with the approved specification.

Another example of validation is during the development of the Airbus A310. Software developers wanted the flaps to drop to the braking position when the landing gear touched the ground. They programmed the system so that flaps started to brake when the landing gear started to spin. However, testing the plane on a wet surface brought forth a bug. When the landing gear hit the slick runway, it did not spin. The flaps never moved into the braking position, and the plane rolled off the runway. From the verification point of view, the program worked. According to validation, however, it did not work. Changes were later made so that the flaps moved when tire pressure changed.

Test Design Techniques

Beginners in software testing start to realize the kinds of tests they need to write to ensure code quality. More experienced developers know that these techniques have formal names and rules.

Here are a few brief descriptions of the most common test design techniques:

Equivalent Partitioning As an example, if you have a range of valid values from 1 to 10, you must choose one correct value within the interval, say 5, and one incorrect value outside the interval, 0.

Boundary Value Analysis Similar to equivalent partitioning, this technique tests both the upper and lower boundaries. Using a valid range of numbers from 1 to 10, we test our code using 0 and 11. Boundary value analysis can be applied to fields, records, files, or any kind of constrained entity (the first two items are most important).

Cause / Effect This is, as a rule, the input of combinations of conditions (causes) that receive a response (effect) from the system. For example, you are testing the ability to add a customer using a particular view. To do this, you will need to enter several fields, such as “Name”, “Address”, “Phone Number” and then, click the “Add” button - this is the cause. After pressing the “Add” button, the system adds the client to the database and displays his number on the screen - this is the effect.

Error Guessing This is when the test analyst uses his knowledge of the system and the ability to interpret the specification in order to “predict” under what input conditions the system may give an error. For example, the spec says “the user must enter a code”. The test analyst will think: “What if I don’t enter the code?”, “What if I enter the wrong code?”, and so on. This is error prediction.

Exhaustive Testing This is an extreme case. You have to test all possible combinations of input values. In principle, this should find all problems. In practice, however, the use of this method is not possible due to the huge number of input values.

Boundary Value Analysis

As the boundary value analysis will be used in next assignments we will dig deeper.

Boundary value analysis is a software testing technique in which tests are designed to include representatives of boundary values in a range. The idea comes from the boundary. Given that we have a set of test vectors to test the system, a topology can be defined on that set. Those inputs which belong to the same equivalence class as defined by the equivalence partitioning theory would constitute the basis. Since the basis sets are neighbors, there would exist a boundary between them. The test vectors on either side of the boundary are called boundary values. In practice this would require that the test vectors can be ordered, and that the individual parameters follow some kind of order (either partial order or total order).



The image depicts a list of numbers from -1 to 14. The numbers -1, 0, 13, and 14 are highlighted in red. The numbers 1 to 12 are highlighted in green. There are boxes around 0, 1, 2 and 11, 12, 13. The labels for each box are “values to test”.

The purpose of this technique is to find errors associated with boundary values. At each end of the range, three values should be checked:

- boundary value
- value before boundary
- value after the boundary

When to Implement Boundary Value Analysis

Boundary value analysis works best when the following conditions are met:

- There is a large number of test cases that are available for testing purposes and for checking them individually.
- The analysis of test data is done at the boundaries of partitioned data after equivalence class partitioning happens and analysis is done.

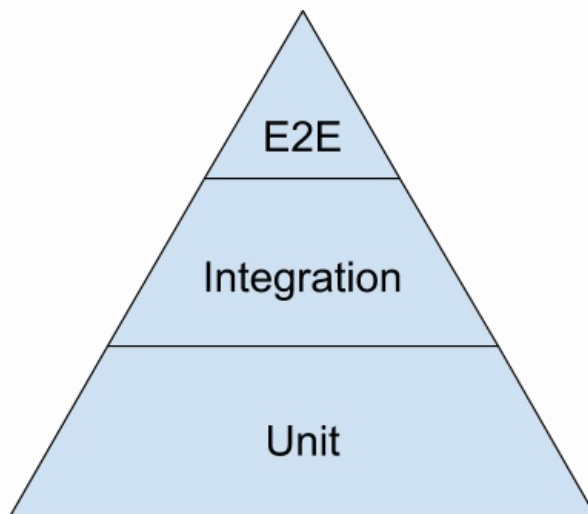
This testing technique is actually known as black-box testing that focuses on valid and invalid test case scenarios. This helps in finding the boundary values at the extreme ends without hampering any effective test data.

This is also used for testing where lots of calculations are required for any kind of variable inputs and for using in varieties of applications.

The testing mechanism also helps in detecting errors or faults at the boundaries of the partition. This is important since most errors occur at the boundaries.

Testing Pyramid

A **testing pyramid**, also often referred to as testing levels, is a grouping of tests by the level of detail and their purpose. This abstraction was invented by Mike Cohn and described in the book “Scrum: Agile Software Development” (Succeeding With Agile: Software Development Using Scrum).



This depicts a pyramid with three levels. The top level is E2E testing, the middle layer is integration testing, and the bottom layer is component testing.

Software development is some kind of a movement along the pyramid from the bottom up. It is important to note:

The test (manual, at high levels, or autotest, at low levels) must be at the same level as the object being tested. For example, a unit test (testing functions, classes, objects, etc.) should be at the unit level. It is wrong if the acceptance level runs a test that will check the minimum unit of code.

Tests at a level above do not test the logic of tests at a level/levels below.

As you move up the pyramid:

- tests become more difficult and expensive to implement;
- tests become more important for business and more critical for users;
- it takes more and more time to pass test cases.

Unit Level

The tests are more commonly referred to as unit tests. At this level, the atomic parts of the code are tested. These can be classes, functions, or class methods.

Example: Your company is developing a Calculator app that can add and subtract. Each operation is one function. Testing each function that does not depend on others is a unit test.

Unit tests find bugs at fundamental levels and are easier to develop and maintain. An important advantage of unit tests is that they are fast and allow you to quickly regress when you change the code (make sure that the new code does not break the old parts of the code).

You should always automate these tests as there are always more unit tests than tests from other levels. These tests run the fastest and require the least amount of resources. Almost always, component tests do not depend on other modules (that's why they are unit tests) and the UI of the system.

In 99% of cases, the development of unit tests is carried out by the developer. When an error is found at this level, no bug reports are generated. The developer finds a bug, fixes it, launches it, and checks it. Abstractly speaking, this is development through testing. The process continues until all tests pass successfully.

At the unit level, the developer (or the autotester) uses the white box method. They know what the minimum unit of code takes and gives, and how it works.

Integration Level

These tests check the relationship of the component, which was checked at the unit level, with another component(s). In addition, integration tests verify the integration of the component with the system (checking work with the OS, services, databases, hardware, etc.). Integration tests are often referred to in English articles as service tests or API tests.

In the case of integration tests, it is rarely necessary to have a UI to test it. Software or system components interact with the unit under test using interfaces. This is where testing comes into play. These are checks of the API, the operation of services (checking logs on the server, records in the database), etc.

Separately, I note that in integration testing, both functional (checking according to the technical specification) and non-functional checks (load on a bunch of components) are performed. This level uses either a gray box or a black box.

In integration testing, there are 3 main ways to test (imagine that each module can consist of even smaller parts):

- From the bottom up (Bottom Up Integration): all the small parts of the module are assembled into one module and tested. Next, the following small modules are assembled into one large one and tested with the previous one, etc.
- From top to bottom (Top Down Integration): first, check the operation of large modules. Any required data from levels below is simulated. Large modules are then broken down and tested as smaller entities.
- Big Bang (“Big Bang” Integration): developers collect all implemented modules of all levels, integrate them into the system, and test. If something does not work or has not been completed, then they fix or refine it.

System level

We previously talked about testing at the system level. It is important to note that the system level will check the interaction of the software under test with the system according to functional and non-functional requirements. Therefore tests should be conducted in an approximate environment of the end user.

Test cases at this level are prepared:

- By requirements.
- Possible ways of using the software.

At the system level, defects such as incorrect use of system resources, unintended combinations of user-level data, incompatibility with the environment, unintended use cases, missing or incorrect functionality, inconvenience of use, etc. are detected.

This level uses a black box. This level allows you to verify the requirements (check the compliance of the software with the prescribed requirements).

E2E Testing

E2E testing is also written as end-2-end or end-to-end, and is more commonly referred to as acceptance testing. At this level, requirements are validated (checking the operation of the software as a whole, not only according to the prescribed requirements, which were checked at the system level).

Requirements are checked on a set of acceptance tests. They are developed based on the requirements and possible uses of the software. Acceptance tests are carried out when the product has reached the required quality level and the software customer is familiar with the acceptance plan (they can describe the set of scenarios and tests, the date of the test, etc.).

Acceptance is carried out either by internal testing (not necessarily testers) or external testing (the customer himself and not necessarily the tester). It is important to remember that E2E tests are more difficult to automate,

take longer, cost more, are more difficult to maintain, and are difficult to regress. So there should be fewer tests like this.

Manual Tests

Up until now, we have talked about the theory behind testing. Let's take a look at writing a simple test. The `convertMonth` function takes an integer value between 1 and 12 and returns the corresponding month. If the value falls outside this range, the function returns a string informing the user of the appropriate value.

```
function convertMonth(n) {  
  if (!Number.isInteger(n)) return "Please use an integer  
between 1 and 12"  
  else if (n == 1) return "January"  
  else if (n == 2) return "February"  
  else if (n == 3) return "March"  
  else if (n == 4) return "April"  
  else if (n == 5) return "May"  
  else if (n == 6) return "June"  
  else if (n == 7) return "July"  
  else if (n == 8) return "August"  
  else if (n == 9) return "September"  
  else if (n == 10) return "October"  
  else if (n == 11) return "November"  
  else if (n == 12) return "December"  
  else return "Please use an integer between 1 and 12"  
}
```

Our test uses the boundary value analysis technique to determine if the boundaries between 0-1-2 and 11-12-13 work as expected. We are going to create a function to test each boundary. Start by creating the function `lowerBoundaryTest`. Create the `expectedIncorrect` variable that has the expected string for when a number falls outside the accepted range. We are going to check that `checkMonth(1)` returns January, `checkMonth(2)` returns February, and `checkMonth(0)` returns the expected error message. If these three things are true, then the test passes.

```
function lowerBoundaryTest() {
  let expectedIncorrect = "Please use an integer between 1 and 12";
  let verifyBoundary = convertMonth(1) === "January";
  let verifyInside = convertMonth(2) === "February";
  let verifyOutside = convertMonth(0) === expectedIncorrect;

  if (verifyBoundary && verifyInside && verifyOutside) {
    return "lower boundary test passed";
  }
  return "lower boundary test failed";
}
```

Similarly, we are going to create the upperBoundaryTest function. We want to verify that checkMonth(12) returns December, checkMonth(11) returns November, and checkMonth(13) returns the expected error message.

```
function upperBoundaryTest() {
  let expectedIncorrect = "Please use an integer between 1 and 12";
  let verifyBoundary = convertMonth(12) === "December";
  let verifyInside = convertMonth(11) === "November";
  let verifyOutside = convertMonth(13) === expectedIncorrect;

  if (verifyBoundary && verifyInside && verifyOutside) {
    return "upper boundary test passed";
  }
  return "upper boundary test failed";
}
```

Call each function and log the results of each function call.

```
console.log(lowerBoundaryTest());
console.log(upperBoundaryTest());
```

Click on the TRY IT button below to run the tests. Both of them should pass.

This may not seem like a lot of code to write for a test. However, we are only checking a simple function. Production code bases are quite large, and the amount of code to write for tests quickly adds up. In addition, there are several types of tests that can be run.

This is why testing frameworks were invented. They abstract away many of the common tasks. This allows you to write your tests in a much more efficient manner.

Frameworks and Their Types

Testing Frameworks support automation testing as a technical implementation guideline. For example, consider a scenario where a testing team includes members who are based on different automation testing code. And, they are not able to grasp the common pieces of code and scripts updated by a team member in a project.

The automation framework not only offers the benefit of reusing the code in various scenarios, but it also helps the team to write down the test script in a standard format. Hence, the test automation framework handles all the issues. Besides, there are many other benefits of using automation framework testing as listed below:

- Maintain a well-defined strategy across the test suites
- Enhanced speed at which testing progresses
- Maintaining the test code will be easy
- The URL or application can be tested accurately
- Continuous testing of coding and delivery will be achieved
- Test automation framework is helpful when you need to execute the same test scripts multiple times with different builds to examine the application and validate output. It is better to avoid automated testing for functionality, which you used only once since building an automation script itself is time-consuming.

Each automation framework has its own architecture, advantages, and disadvantages. Some of these frameworks are:

- Linear Scripting Framework
- Modular Driven Framework
- Behavior Driven Framework
- Data Driven Framework
- Keyword Driven Framework
- Test Driven Development Framework
- Hybrid Testing Framework

Linear Scripting Framework:

This framework is based on the concept of record and playback mode that is always achieved in a linear manner. It is more commonly named as record and playback model.

Typically, in this scripting driven framework, the creation and execution of test scripts are done individually and this framework is an effective way to get started for enterprises.

The automation scripting is done in an incremental manner where every new interaction will be added to the automation tests.

Modular Testing Framework:

Abstraction is the concept on which this framework is built. Based on the modules, independent test scripts are developed to test the software. Specifically, an abstraction layer is built for the components to be hidden from the application under test.

This sort of abstraction concept ensures that changes made to the other part of the application do not affect the underlying components.

Data Driven Testing Framework:

In this testing framework, a separate file in a tabular format is used to store both the input and the expected output results. In this framework, a single driver script can execute all the test cases with multiple sets of data.

This driver script contains navigation that spreads through the program which covers both reading of data files and logging of test status information.

Keyword Driven Testing Framework:

Keyword Driven Testing framework is an application independent framework and uses data tables and keywords to explain the actions to be performed on the application under test. This type of framework is often used for web-based applications, as it can be stated as an extension of data driven testing framework.

Hybrid Testing Framework:

A hybrid testing framework is the combination of modular, data-driven and keyword test automation frameworks. As this is a hybrid framework, it has been based on the combination of many types of end-to-end testing approaches.

Test Driven Development Framework (TDD):

Test driven development is a technique of using automated unit tests to drive the design of software and separates it from any dependencies. Earlier, with traditional testing, a successful test could find one or more defects. Using TDD increases the speed of tests and improves the confidence that the system meets the requirements and is working properly when compared to traditional testing.

Behavior Driven Development Framework (BDD):

This has been derived from the TDD approach. In this method tests are more focused and are based on the system behavior. Testers can create test cases in simple English language. This simple English language helps even non-technical people to easily analyze and understand the tests.