What Is Integration Testing?

The integration testing definition refers to assessing the communication between separate software modules. Typically, the project team has to unit test the system before moving on to integration testing. In the software development life cycle, integration testing is the second step.

The main aim of integration testing is to make sure the differences in logic patterns developers use when creating a module don't compromise the connectivity of the system. There are several approaches to integration testing:

- **Top-down** integration testing is the practice of prioritizing the validation of complex, layered modules over low-level ones. In case one of the modules is not ready for testing yet, QA teams use stubs.
- **Bottom-up** integration testing is the opposite method to top-down integration testing. It implies validating basic modules first and integrating the complex ones later. The reasoning behind the approach is that it takes less time to create a low-level module — that's why such components should be tested even when the more complex parts of the system are still in development.
- **Big bang**. If the testing team chooses this approach, it means that all modules will be tested simultaneously. QA specialists that execute tests by the big bang framework don't test modules individually and instead wait until they are fully complete. Such a testing strategy is not the most efficient due to the high odds of missing major defects during testing.

- **Mixed** (or sandwiched) integration testing is a synchronized adoption of top-down and bottom-up practices. By following this approach, the testing team does not necessarily have to wait until either high- or low-level modules are fully coded, testing whichever of the two is ready.

Integration Testing Objectives

By conducting integration testing, teams aim to ensure that the system has no connectivity or communication issues on the level of software modules. If undetected, integration failures are difficult and expensive to fix after the product's release as developers have to make in-depth system-level changes to remove these defects.

After the integration testing process is complete, the testing team can focus on validating end-user journeys and usability.

The main integration testing objectives go as follows:

- **Making sure that software modules work well when you integrate them together.** Integration testing ensures that the

connectivity between modules meets the requirements specified by the testing plan.

- **Find interface errors.** During integration testing, both functional and non-functional interface components are validated. After completing a series of integration tests, the testing team should have full confidence in the performance of the software's interface.
- **Ensure the synchronization between modules**. Integration tests help project teams ensure that software modules can function with no defects simultaneously and are fully synchronized with each other.
- **Fixes exception handling defects**. Exception handling mechanisms are crucial for high-assurance systems. Typically, such mechanisms are presented in most programming languages. However, QA teams need to ensure that the application is well protected against exception handling defects — integration testing helps pinpoint weak spots and red flags and mitigate the risks before the release of the final build.

Advantages of Integration Testing in Software Development

Integration testing in software testing is a must-have. It helps teams pinpoint weak spots and system defects at the early stages of development and promotes more confidence in the product.

Here are some integration testing advantages:

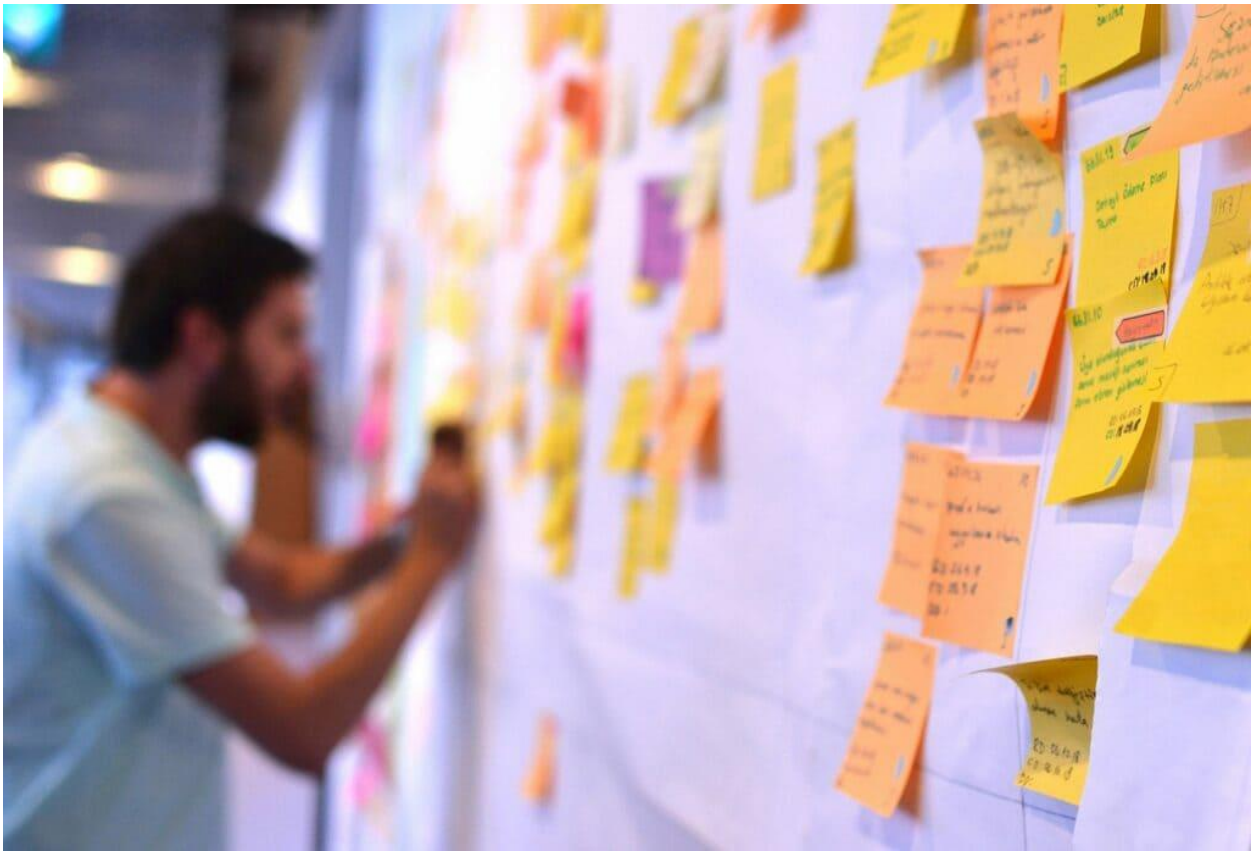
- **Relatively fast testing process**. Although it takes more time to run integration tests, as opposed to validating separate system units, the process improves the speed and facilitates end-to-end testing.
- **High code coverage.** Integration testing has a wide scope, allowing QA specialists to test the entire system. The odds of missing out on a critical connectivity defect after a series of integration tests are slim. Other than that, the process is easy to keep track of.
- **Efficient system-level issue detection**. Integration testing falls under the definition of system-level testing since a tester has to combine modules and validate their joint performance. Later, the team will get a better look at the system's overall performance by moving on to the next stage — system testing.
- **Detects bugs early as it's run at the early stages of development**. Adopting integration testing allows the project team to pinpoint security and connectivity issues in the early stages of development. As such, integration testing offers developers superior control over the product and promotes the awareness of system vulnerabilities.

Unit Testing Definition

If you compare unit vs integration testing, unit testing is the first testing activity in the software testing life cycle. It’s a common practice for project teams to not involve testers in these stages, asking developers to perform unit tests instead.

Unit testing does not require any specialized skills or well-trained workforce. Although there’s been much backlash regarding the high cost and inefficiency of unit tests, the approach has its own advantages.

Units are the objects of unit testing — these are the smallest components of a tested system. As a software project typically consists of multiple units, automating unit testing is a popular practice among QA teams.



The Objectives of Unit Test

In a nutshell, unit testing aims to separate system components and check their individual functionality. Other than the primary objective, the approach helps tech teams accomplish the following:

Find bugs early in the development cycle

Unit testing allows introducing bugs and system defects early on in the development process. This way, the development team can resolve issues before integrating the units together and impacting the whole system.

Use unit testing logs as documentation

The logs of unit testing will offer the project team a detailed description of the system on the micro-level. This testing method improves the interchangeability within the team since a newcomer developer can rely on logs provided by peers to be more familiar with the system. Unit testing provides a solid basic framework for understanding and handling APIs.

Improve the efficiency of code reuse

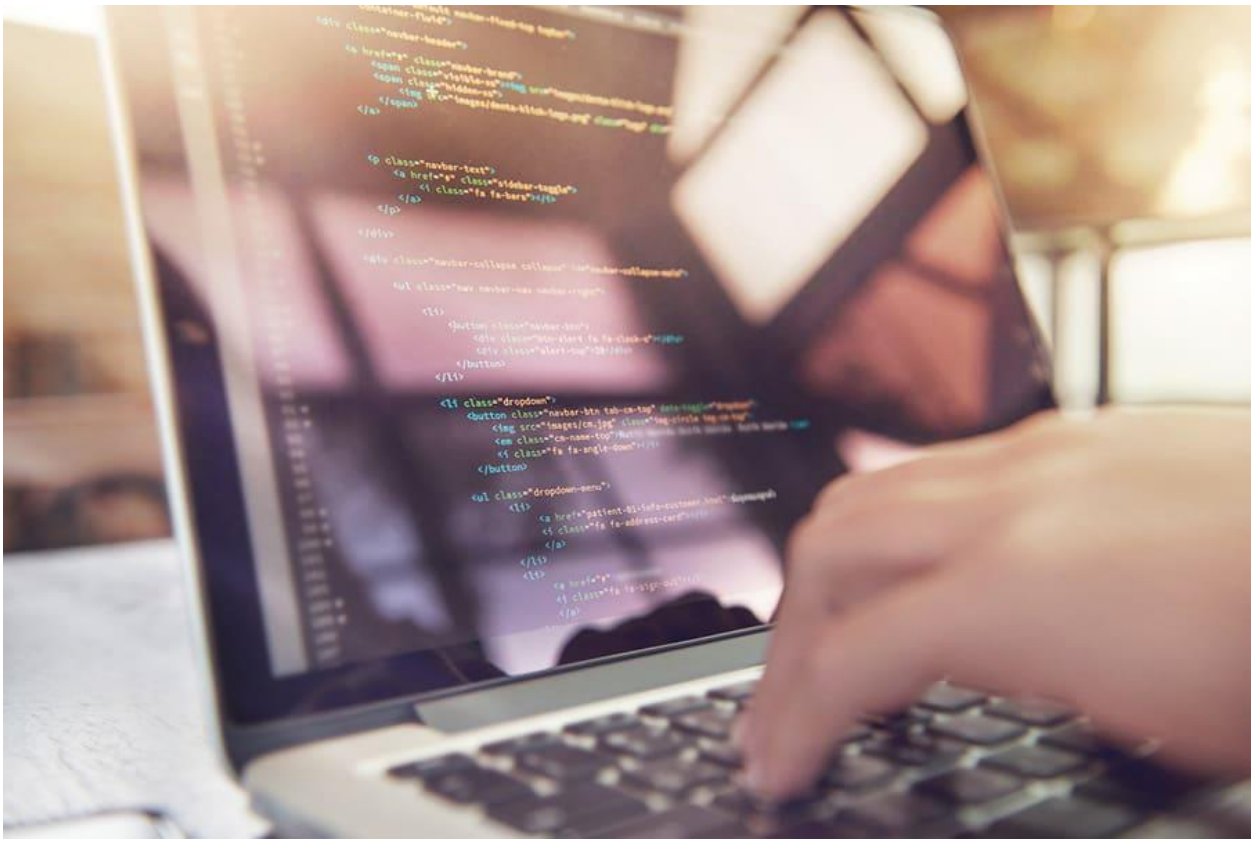
Unit tests improve code reusability since the reused units are well-tested. If the development team misses out on unit testing, the odds of reusing buggy code and spawning numerous system failures in the future increase dramatically.

By testing the units beforehand, developers can be confident that there are no bugs or compilation issues, and that the written code fulfills its function according to business specification requirements.

Validate the behavior of the system’s atomic behavioral unit

Missing out on unit testing will make other testing cycle stages considerably more challenging since the impact of system failures will have a higher magnitude and is likely to prevent the program from working altogether.

Testing every unit of a system individually is a way to ensure that code-level bugs will not complicate integration or system testing.



Advantages of Unit Tests in Software Development

Unit testing is considered, by many teams, an unneeded addition to a tester's busy working routine. Since it takes a while to unit-test the entire system, it's common for tech project managers to skip the stage altogether.

The truth is, unit testing should be integrated into the development routine since it's relatively cheap and easy to perform. The advantages of the approach are numerous — here are but a few:

- **It lowers maintenance costs**. Testing early and often is a tried-and-true way to reduce the number of testing expenses. In our experience, fixing a bug in the early stages of development is about 4-5 times cheaper than coming back to it after the product is released.
- **Reduces uncertainty in the behavior of units.** Unit testing in software testing helps validate the performance of the basic code, offers a detailed description of a unit's behavior in the shape of testing documentation and logs, and increases the confidence in the functionality of the backbone code among the tech team, as well as the acceptance of the system by the project stakeholders.
- **Helps detect changes that can break the design contract**. Other than helping maintain and change the code, unit tests help pinpoint the defects that break the design contracts. The testing method helps improve code design as a whole, encouraging developers to establish a uniform code interface and ensuring the test ability of every component.
- **Doesn't require a highly skilled team of testers and can be conducted by developers**. When conducting unit testing, developers don't have to manage multi-layered interfaces or write a complex test case. As a rule of thumb, most types of unit tests are executed in an automated testing environment and don't require superior concentration from the testing team.

Unit testing	**Integration testing**
Validates the system unit-by-unit	Assesses the system as a whole by integrating several modules simultaneously
Fully autonomous — each unit is treated as a separate system	Connected since the testing team predominantly pays attention to the relationship between tested components
First level of software testing	Follows unit testing in the SLDR
No reliance on dependencies	Strongly relies on dependencies, involves the use of databases
Tests are faster to perform	Tests are slower

Usually performed by developers	Requires an experienced testing team
Is performed at the coding level	Is performed at the communication level

Difference Between Unit Testing and Integration Testing

As we compared unit and integration testing, it's evident that both approaches have significant differences. To clarify the distinction between the two, take a look at the unit test vs integration test comparative table.

Summary

Unit testing and integration testing are both a part of the software testing life cycle. The two share a common objective — detecting software defects as early on as possible.

There's a big difference between unit testing and integration testing. While the former approaches the system as a series of modules and examines the interactions and proper connectivity between them, the latter tests the product unit-by-unit.

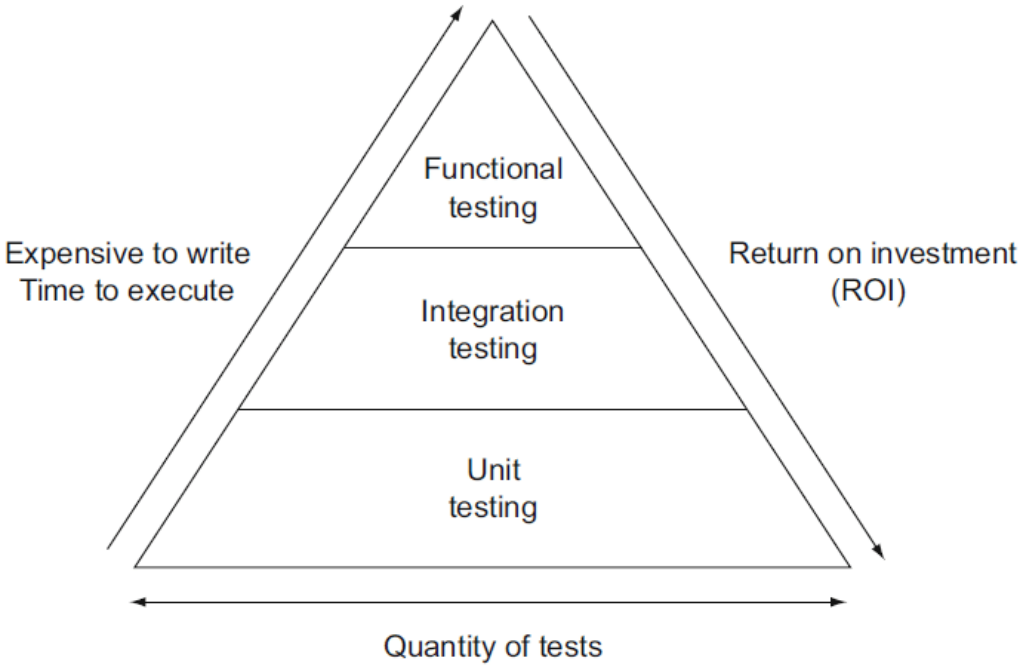
If you want to conduct both of these software testing types, reach out to Performance Lab. Our certified testers will bring forth the best testing practices to ensure your software is fully tested and the results are well-documented to avoid post-release bugs and defects. Take a look at the group of testing services our team offers. Leave us a message to discuss your idea and testing needs in detail — our account manager will reach out to you shortly.

Unit Testing Vs Integration Testing Vs Functional Testing

Unit testing means testing individual modules of an application in isolation (without any interaction with dependencies) to confirm that the code is doing things right.
Integration testing means checking if different modules are working fine when combined together as a group.
Functional testing means testing a slice of functionality in the system (may interact with dependencies) to confirm that the code is doing the right things.
Functional tests are related to integration tests; however, they signify to the tests that check the entire application's functionality with all the code running together, nearly a super integration test.

Unit testing considers checking a single component of the system whereas functionality testing considers checking the working of an application against the intended functionality described in the system requirement specification. On the other hand, integration testing considers checking integrated modules in the system.
And, most importantly, to optimize the return on investment (ROI), your code base should have as many unit tests as possible, fewer integration tests and the least number of functional tests.

This is illustrated best in the following test pyramid:



Unit tests are easier to write and quicker to execute. The time and effort to implement and maintain the tests increases from unit testing to functional testing as shown in the above pyramid.

Example:
Let us understand these three types of testing with an oversimplified example.

E.g. For a functional mobile phone, the main parts required are “battery” and “sim card”.
Unit testing Example – The battery is checked for its life, capacity and other parameters. Sim card is checked for its activation.
Integration Testing Example – Battery and sim card are integrated i.e. assembled in order to start the mobile phone.
Functional Testing Example – The functionality of a mobile phone is checked in terms of its features and battery usage as well as sim card facilities.

Now, let us now take a technical example of a login page:

Almost every web application requires its users/customers to log in. For that, every application has to have a “Login” page which has these elements:

- Account/Username
- Password
- Login/Sign in Button

For Unit Testing, the following may be the test cases:

- Field length – username and password fields.
- Input field values should be valid.
- The login button is enabled only after valid values (Format and lengthwise) are entered in both the fields.

For Integration Testing, the following may be the test cases:

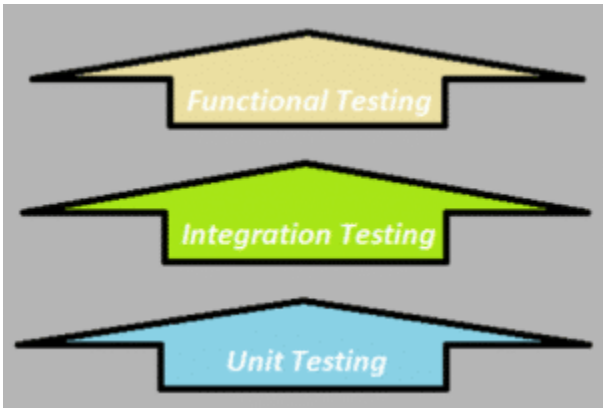
- The user sees the welcome message after entering valid values and pushing the login button.
- The user should be navigated to the welcome page or home page after valid entry and clicking the Login button.

Now, after unit and integration testing are done, let us see the additional **test cases that are considered for functional testing:**

1. The expected behavior is checked, i.e. is the user able to log in by clicking the login button after entering a valid username and password values.
2. Is there a welcome message that is to appear after a successful login?
3. Is there an error message that should appear on an invalid login?
4. Are there any stored site cookies for login fields?
5. Can an inactivated user log in?
6. Is there any ‘forgot password’ link for the users who have forgotten their passwords?

There are much more such cases which come to the mind of a functional tester while performing functional testing. But a developer cannot take up all cases while building Unit and Integration test cases.

Thus, there are a plenty of scenarios that are yet to be tested even after unit and integration testing.



It is now time to examine Unit, Integration and Functional testing one by one.

What is Unit Testing?

As the name suggests, this level involves testing a 'Unit'.

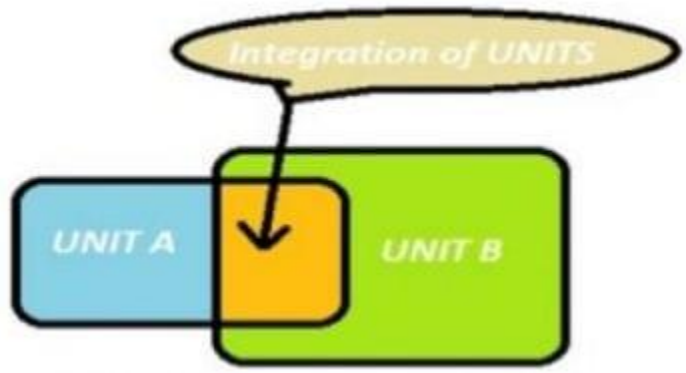
Here unit can be the smallest part of an application that is testable, be it the smallest individual function, method, etc. Software developers are the ones who write the unit test cases. The aim here is to match the requirements and the unit's expected behavior.

Below are a few important points about unit testing and its benefits:

- Unit testing is done before Integration testing by software developers using white box testing techniques.
- Unit testing does not only check the positive behavior i.e. the correct output in case of valid input, but also the failures that occur with invalid input.
- Finding issues/bugs at an early stage is very useful and it reduces the overall project costs. As Unit testing is done before integration of code, issues found at this stage can be resolved very easily and their impact is also very less.
- A unit test tests small pieces of code or individual functions so the issues/errors found in these test cases are independent and do not impact the other test cases.
- Another important advantage is that the unit test cases simplify and make testing of code easier. So, it becomes easier to resolve the issues at a later stage too as only the latest change in the code is to be tested.
- Unit test saves time and cost, and it is reusable and easy to maintain.

What is Integration Testing?

Integration testing is testing the integration of different part of the system together. Two different parts or modules of the system are first integrated and then integration testing is performed.

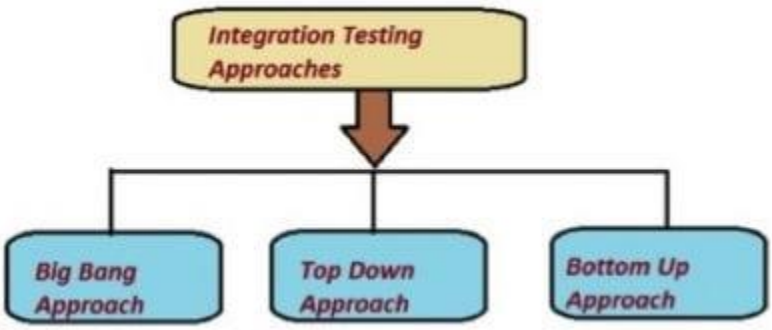


The aim of integration testing is to check the functionality, reliability, and performance of the system when integrated.

Integration testing is performed on the modules that are unit tested first and then integration testing defines whether the combination of the modules give the desired output or not.

Integration testing can either be done by independent testers or by developers too.

There are 3 different types of Integration testing approaches. Let us discuss each one of them briefly:

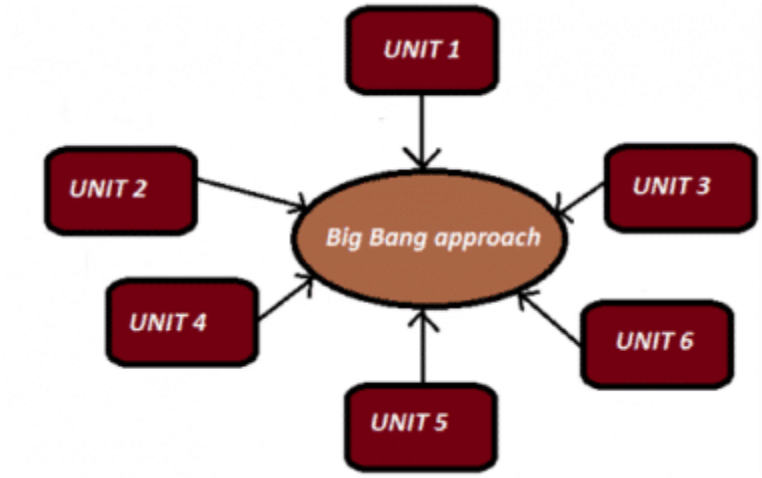


a) Big Bang Integration Approach

In this approach, all the modules or units are integrated and tested as a whole at one time. This is usually done when the entire system is ready for integration testing at a single point of time.

Please do not confuse this approach of integration testing with system testing, only the integration of modules or units is tested and not the whole system as it is done in system testing.

The big bang approach's major **advantage** is that everything integrated is tested at one time.
One major **disadvantage** is that it becomes difficult to identify the failures.
Example: In the figure below, Unit 1 to Unit 6 are integrated and tested using the Big bang approach.

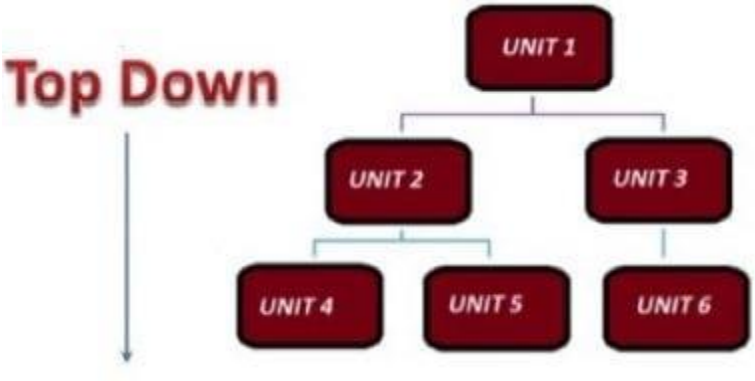


b) Top-Down Approach

Integration of the units/modules is tested from the top to bottom levels step by step.

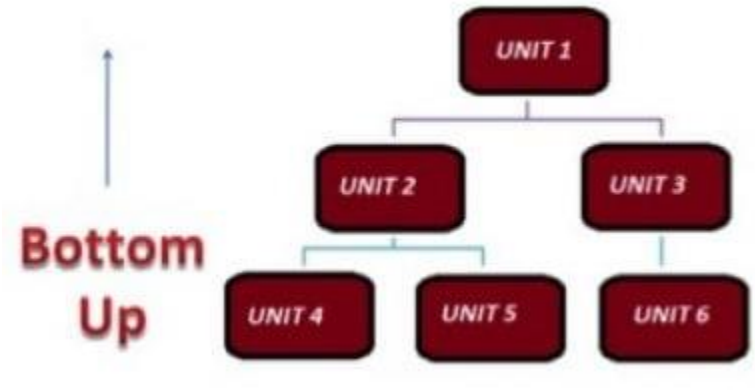
The first unit is tested individually by writing test STUBS. After this, the lower levels are integrated one by one until the last level is put together and tested.
The top-down approach is a very organic way of integrating as it is consistent with how things happen in the real environment.

The only **concern** with this approach is that the major functionality is tested at the end.



c) Bottom-Up Approach

Units/modules are tested from bottom to top level, step by step, until all levels of units/modules are integrated and tested as one unit. Stimulator programs called **DRIVERS** are used in this approach. It is easier to detect issues or errors at the lower levels.
The major **disadvantage** of this approach is that the higher-level issues can only be identified at the end when all the units have been integrated.



Unit Testing vs Integration Testing

Having had enough discussion about unit testing and integration testing, let us quickly go through the differences between the two in the following table:

Unit Testing	Integration Testing
Tests the single component of the whole system i.e. tests a unit in isolation.	Tests the system components working together i.e. test the collaboration of multiple units.
Faster to execute	Can run slow
No external dependency. Any external dependency is mocked or stubbed out.	Requires interaction with external dependencies (e.g. Database, hardware, etc.)
Simple	Complex
Conducted by developer	Conducted by tester
It is a type of white box testing	It is a type of black box testing
Carried out at the initial phase of testing and then can be performed anytime	Must be carried out after unit testing and before system testing
Cheap maintenance	Expensive maintenance
Begins from the module specification	Begins from the interface specification
Unit testing has a narrow scope as it just checks if each small piece of code is doing what it is intended to do.	It has a wider scope as it covers the whole application
The outcome of unit testing is detailed visibility of the code	The outcome of integration testing is the detailed visibility of the integration structure

Unit Testing	Integration Testing
Uncover the issues within the functionality of individual modules only. Does not exposes integration errors or system-wide issues.	Uncover the bugs arise when different modules interact with each other to form the overall system

Functional Testing

A black box testing technique, where the functionality of the application is tested to generate the desired output on providing a certain input is called 'Functional testing'.
In our software testing processes, we do this by writing test cases as per the requirements and scenarios. For any functionality, the number of test cases written can vary from one to many.

Test cases basically comprise of the following parts:

- Test Summary
- Prerequisites (if any)
- Test case input steps
- Test data (if any)
- Expected output
- Notes (if any)

"Requirement-Based" and "Business scenario-based" are the two forms of functional testing that are carried out.
In Requirement based testing, test cases are created as per the requirement and tested accordingly. In a Business scenario based functional testing, testing is performed by keeping in mind all the scenarios from a business perspective.

However, the major **disadvantage** of functional testing is the probable redundancy in testing and the possibility of missing some logical errors.

Exact Difference

Let's look at their differences.

Here are some of the major ones:

	Unit testing	Integration testing	Functional testing
Definition and purpose	Testing smallest units or modules individually.	Testing integration of two or more units/modules combined for performing tasks.	Testing the behavior of the application as per the requirement.
Complexity	Not at all complex as it includes the smallest codes.	Slightly more complex than unit tests.	More complex compared to unit and integration tests.
Testing techniques	White box testing technique.	White box and black box testing technique. Grey box testing	Black box testing technique.

	Unit testing	Integration testing	Functional testing
Major attention	Individual modules or units.	Integration of modules or units.	Entire application functionality.
Error/Issues covered	Unit tests find issues that can occur frequently in modules.	Integration tests find issues that can occur while integrating different modules.	Functional tests find issues that do not allow an application to perform its functionality. This includes some scenario-based issues too.
Issue escape	No chance of issue escape.	Less chance of issue escape.	More chances of issue escape as the list of tests to run is always infinite.