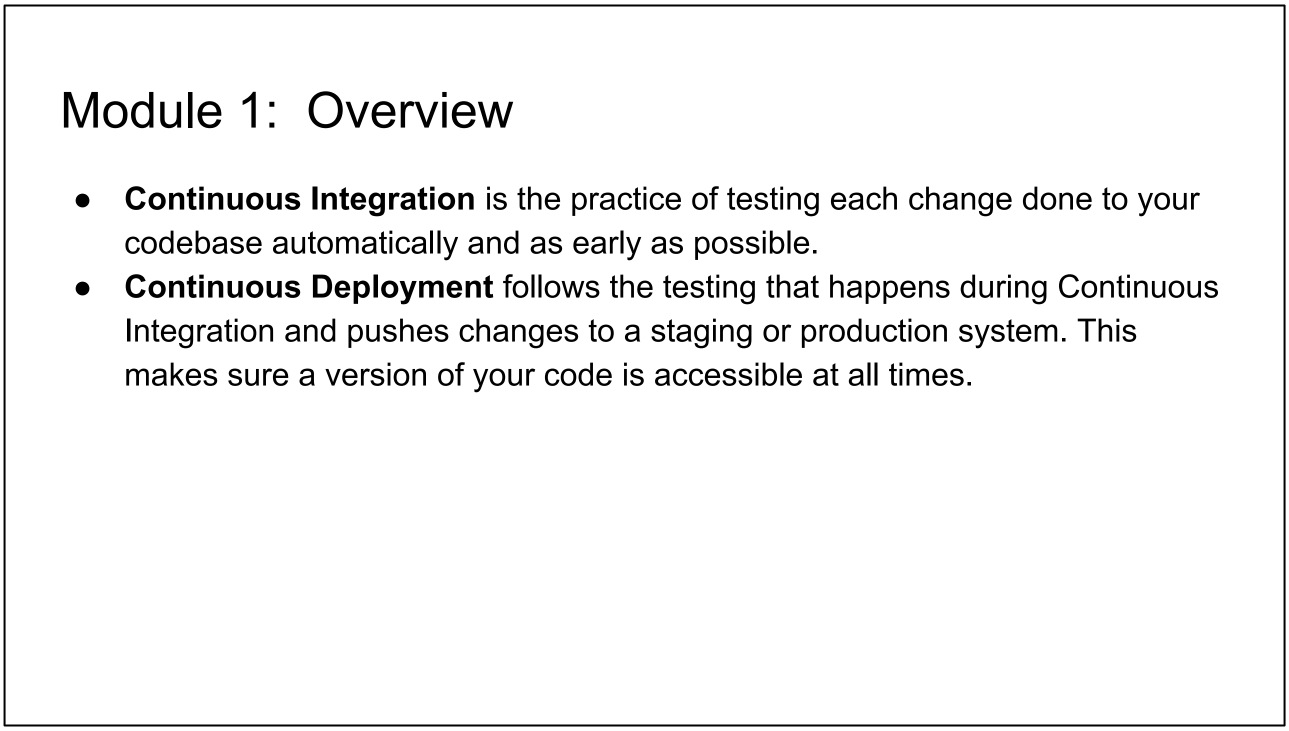
Continuous Integration, Testing, and Deployment Overview (CI/CD)

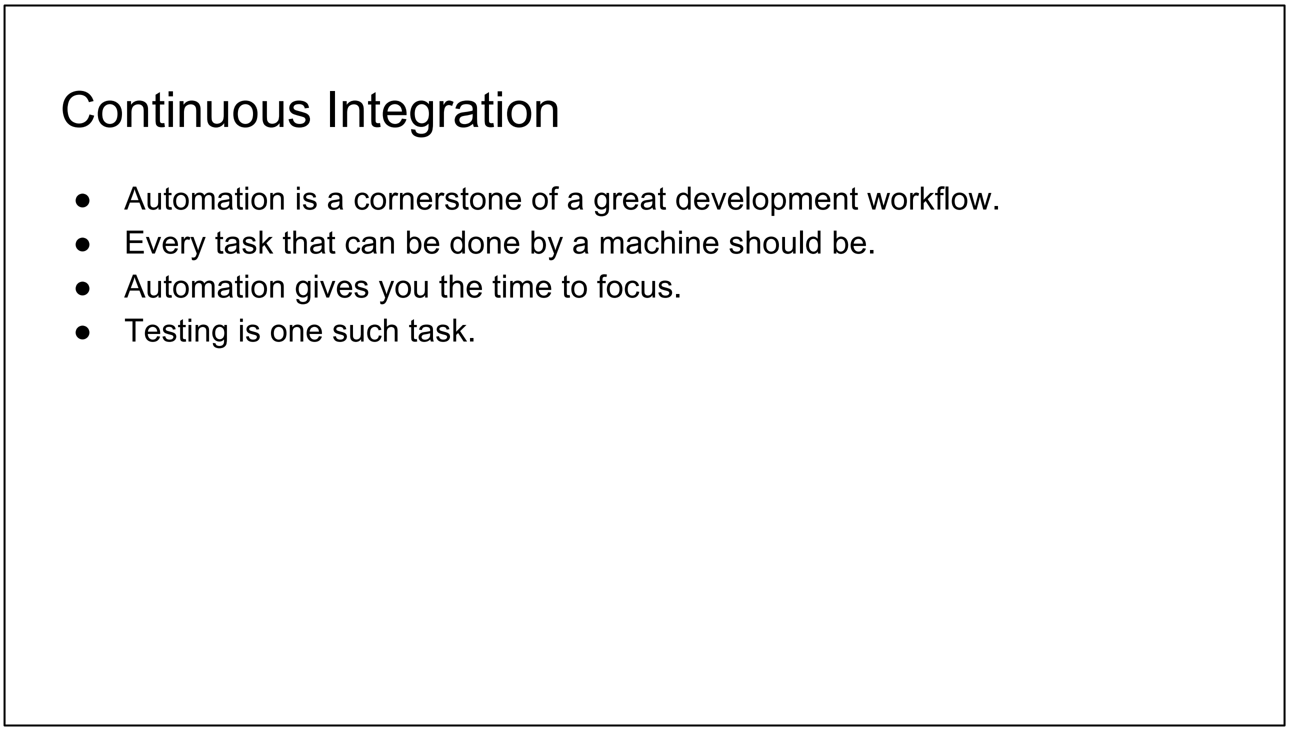


Module 1: Overview

- **Continuous Integration** is the practice of testing each change done to your codebase automatically and as early as possible.
- **Continuous Deployment** follows the testing that happens during Continuous Integration and pushes changes to a staging or production system. This makes sure a version of your code is accessible at all times.

First, we will take a close look at Continuous Integration and show you how to get started with testing. Then we will move on to Continuous Deployment which is the next logical step.

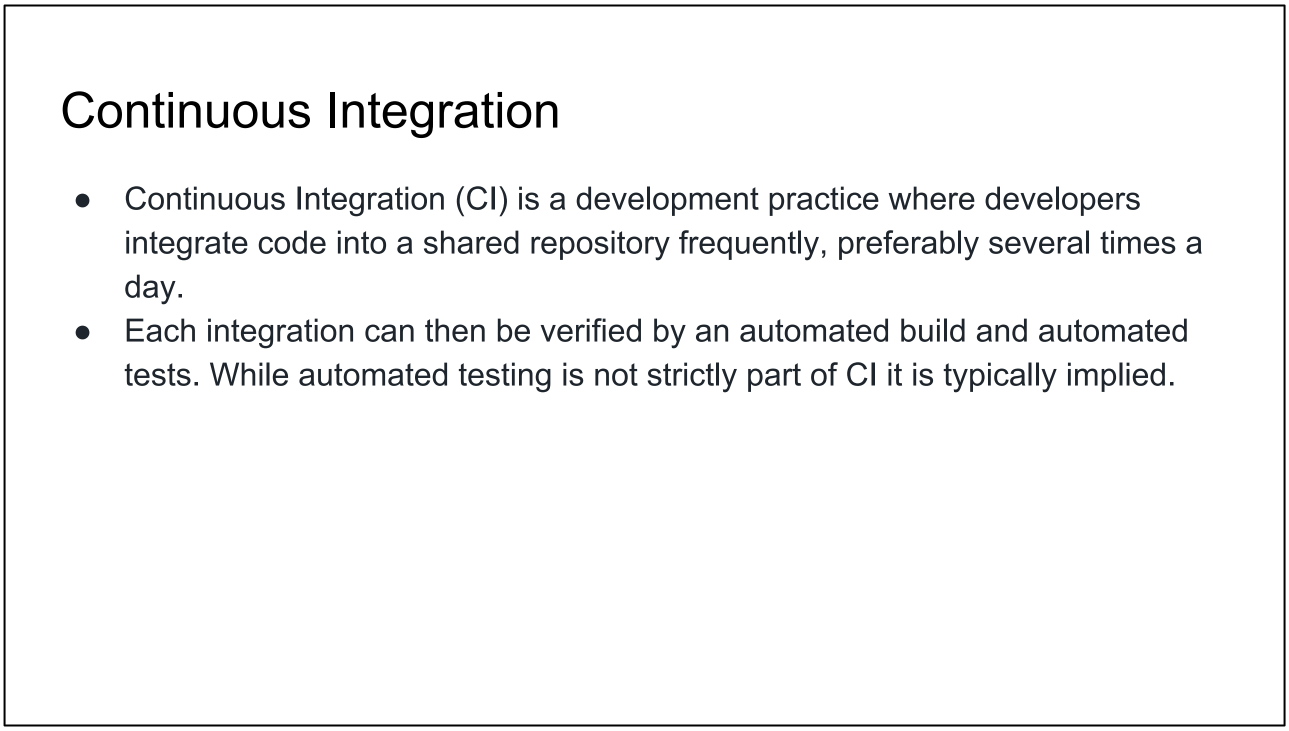
With both in place, your development team will be orders of magnitude more efficient.



Continuous Integration

- Automation is a cornerstone of a great development workflow.
- Every task that can be done by a machine should be.
- Automation gives you the time to focus.
- Testing is one such task.

Through testing, you can be sure that the most important steps your customers will take through your system are working, regardless of the changes you make. This gives you the *confidence* to experiment, implement new features, and ship updates quickly.



Continuous Integration

- Continuous Integration (CI) is a development practice where developers integrate code into a shared repository frequently, preferably several times a day.
- Each integration can then be verified by an automated build and automated tests. While automated testing is not strictly part of CI it is typically implied.

One of the key benefits of integrating regularly is that you can detect errors quickly and locate them more easily. As each change introduced is typically small, pinpointing the specific change that introduced a defect can be done quickly.

In recent years CI has become a best practice for software development and is guided by a set of key principles. Among them are revision control, build automation and automated testing.

Continuous Integration

Continuous Integration doesn’t get rid of bugs, but it does make them dramatically easier to find and remove.

Martin Fowler, Chief Scientist

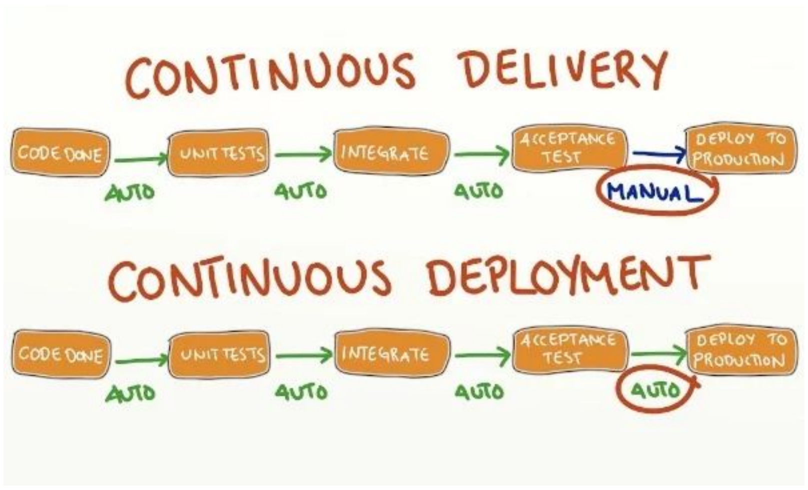
Continuous Deployment

Continuous Deployment is closely related to Continuous Integration and refers to keeping your application deployable at any point or even automatically releasing to a test or production environment if the latest version passes all automated tests.

Continuous Delivery

Continuous Delivery is the practice of keeping your codebase deployable at any point. Beyond making sure your application passes automated tests it has to have all the configuration necessary to push it into production. Many teams then do push changes that pass the automated tests into a test or production environment immediately to ensure a fast development loop.

Continuous Delivery vs Continuous Deployment



You should focus on setting up a simple **Continuous Integration** process as early as possible. But that's not where things should end. Even though Continuous Integration (CI) is important, it's only the first step in the process. You also want to set up Continuous Deployment (CD), the workflow that automates your software deployment and lets you focus on building your product.

As we pointed out before, Continuous Deployment is closely related to Continuous Integration and refers to keeping your application deployable at any point or even automatically releasing into production if the latest version passes all automated tests.

If you wish to release your product really fast, you should automate your entire workflow, not just the testing. Having a well designed and smoothly running Continuous Deployment (CD) solution will be the glue between the tools you use, especially between the SCM (Source Control Management) provider/server and the hosting environment you are using. This will also help you to onboard new people and grow your team as they can rely on a fully automated process from day one.

Test Driven Development (TDD)

- Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle.
- First the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.

The following sequence of steps is generally followed:

- Add a test
- Run all tests and see if the new one fails
- Write some code
- Run tests
- Refactor code
- Repeat

Benefits and Advantages of Continuous Integration and Deployment

- Continuous Integration has many benefits.
- A good CI setup speeds up your workflow and encourages the team to push every change without being afraid of breaking anything.
- There are more benefits to it than just working with a better software release process.
- Continuous Integration brings great business benefits as well.

If you test and deploy code more frequently, it will eventually reduce the risk level of the project you are working on as you can detect bugs and code defects earlier. This means they are easier to fix and you can fix them sooner which makes it cheaper to fix them.

When you have a CI process in place that is hooked into a Continuous Delivery workflow it’s easy to share your code regularly. This code sharing helps to achieve more visibility and collaboration between team members. Eventually this increases communication speed and efficiency within your organization as everybody is on the same page, always.

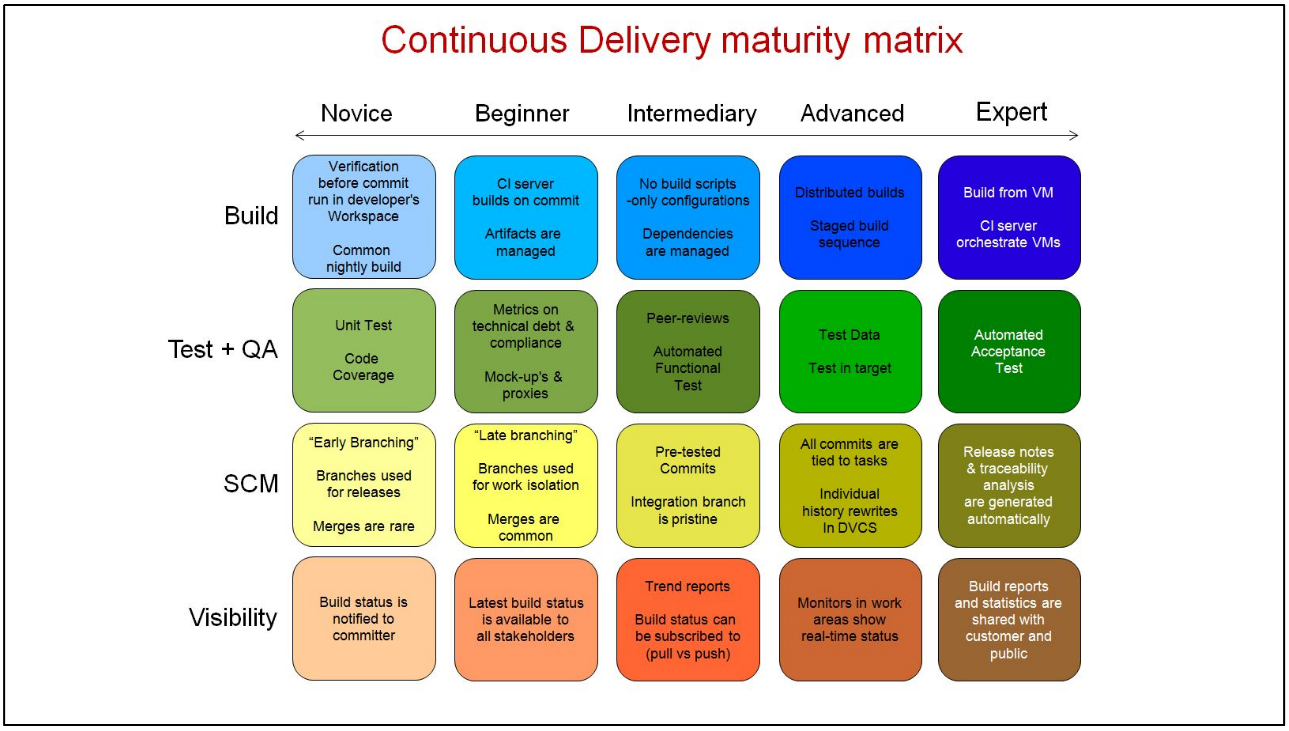
As you release code often, the gap between the application in production and the one the developer is working on will be much smaller. Your thinking about how to develop features most probably will change. As every small change will be tested automatically and the whole team can know about these changes you will want to work on small, incremental changes when developing new features. This results in less assumptions as you can build features quicker and test and deploy them automatically for your users to see as soon as possible, thus gaining valuable feedback from them faster.

Principles of Continuous Integration

- Maintain a code repository
- Automate your build
- Make your build self-testing
- Daily commits to the baseline by everyone on the team
- Every commit (to the baseline) should be built

Principles of Continuous Integration con’t

- Keep your builds fast
- Clone the production environment and test there
- Make it easy to get the latest deliverables
- Everyone on the team can see the results of your latest build
- Automate build deployment



Continuous Delivery maturity matrix

	Novice	Beginner	Intermediary	Advanced	Expert
Build	Verification before commit run in developer's Workspace Common nightly build	CI server builds on commit Artifacts are managed	No build scripts -only configurations Dependencies are managed	Distributed builds Staged build sequence	Build from VM CI server orchestrate VMs
Test + QA	Unit Test Code Coverage	Metrics on technical debt & compliance Mock-up's & proxies	Peer-reviews Automated Functional Test	Test Data Test in target	Automated Acceptance Test
SCM	"Early Branching" Branches used for releases Merges are rare	"Late branching" Branches used for work isolation Merges are common	Pre-tested Commits Integration branch is pristine	All commits are tied to tasks Individual history rewrites In DVCS	Release notes & traceability analysis are generated automatically
Visibility	Build status is notified to committer	Latest build status is available to all stakeholders	Trend reports Build status can be subscribed to (pull vs push)	Monitors in work areas show real-time status	Build reports and statistics are shared with customer and public

In the **Continuous Delivery Maturity Checklist** you can actually check the practices you currently perform to see how mature you are in each area of Continuous Delivery. The higher you score on the test, the closer you are to achieving CD Maturity. The checklist is not only good to follow when you code, but it can also help you identify weaknesses and areas to improve in your company's CD process. The main areas of the CD process include:

- Source Control
- Build Process
- Testing & Q&A
- Deployment
- Visibility

Module 2: Git Essentials Review and Branching

Git Tools

Module : Functional Testing Overview

Functional Testing

- Functional testing is a form of automated testing that deals with how applications functions, or, in other words, its relation to the users and especially to the rest of the system.
- Traditionally, functional testing is implemented by a team of testers, independent of the developers.

While unit testing tests both what an application segment does and how it does it, **functional testing tests only *what* the application does**. A functional test is not concerned with an application's internal details.

Another way of stating this is that functional testing is "the customer test". But even this is misleading. Developers need a benchmark during all development stages — a developer-independent benchmark — to tell them what they have and have not achieved. Functional testing begins as soon as there is a function to test and continues through the application's completion and first customer contact.

The expression, "the customer test", can be misleading in another way. Some people think of functional testing as mimicking a user and checking for an expected output. But the real customer is not just someone feeding commands into the application. They are running the application on a system, simultaneously with other applications, with constant fluctuations in user load. The application, of course, should be crash-resistant in the face of these conditions.

More and more, the user interface of our applications is supplied by pre-tested components, which produce few surprises once they are integrated correctly. On the other hand, we are constantly asked to write custom applications for systems of ever-increasing complexity. Therefore, the central functions tested by a functional test are increasingly system-related rather than user-related.

Functional Testing

- Functional testing is concerned only with the functional requirements of a system or subsystem and covers how well (if at all) the system executes its functions.
- These include any user commands, data manipulation, searches and business processes, user screens, and integrations.
- Functional testing is done using the functional specifications provided by the client or by using the design specifications like use cases provided by the design team.

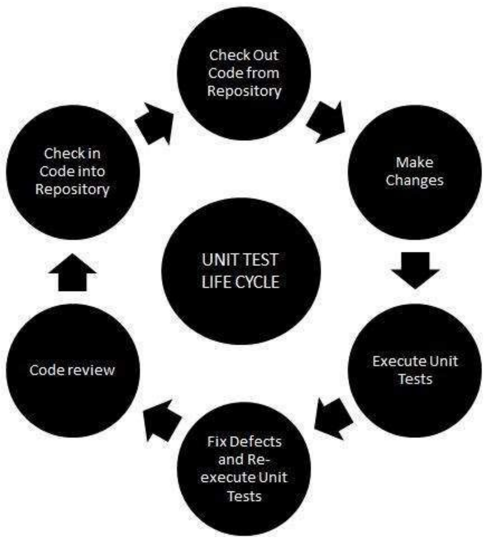
Types of Functional Testing

- Black Box Testing
- White Box Testing
- Unit Testing
- Integration Testing
- Smoke Testing
- User Acceptance Testing
- Localization Testing
- Interface Testing
- Usability Testing
- System Testing
- Regression Testing
- Globalization Testing

Unit Testing

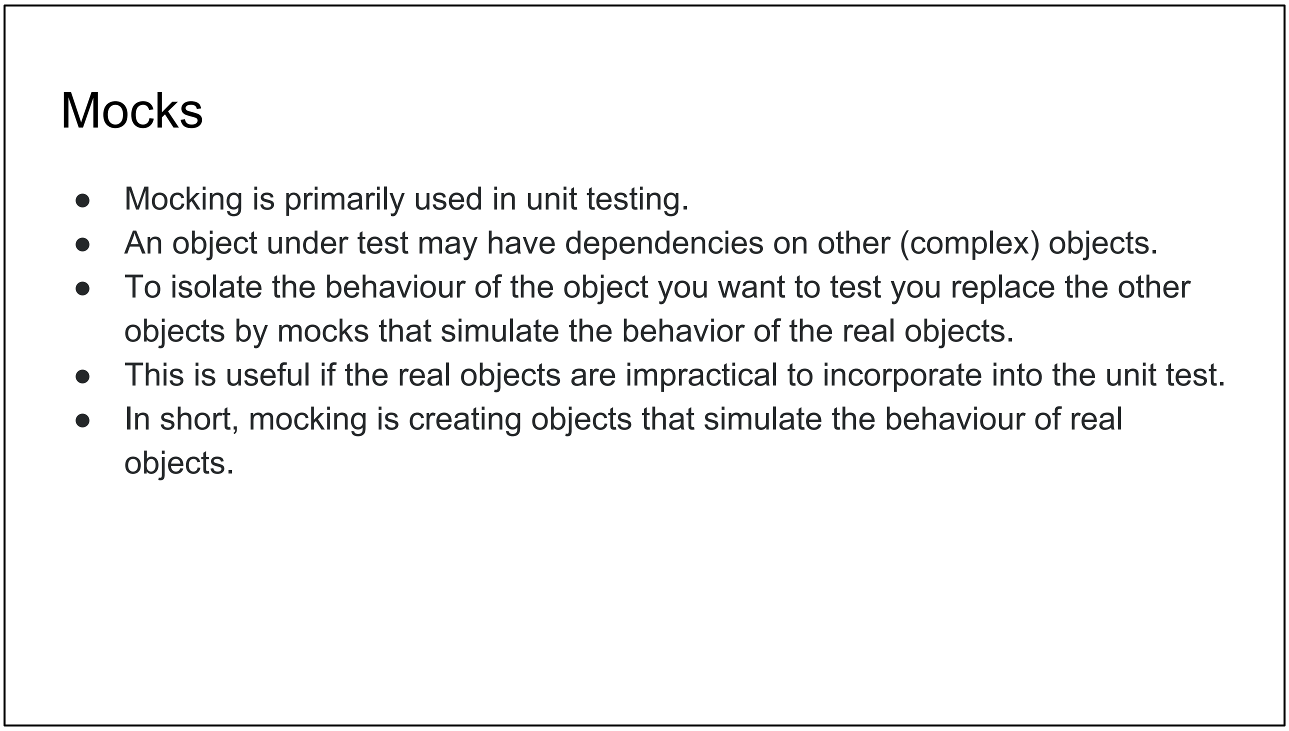
- Unit testing, a testing technique using which individual modules are tested to determine if there are any issues by the developer himself.
- It is concerned with functional correctness of the standalone modules.
- The main aim is to isolate each unit of the system to identify, analyze and fix the defects.
- Reduces Defects in the Newly developed features or reduces bugs when changing the existing functionality.
- Reduces Cost of Testing as defects are captured in very early phase.
- Improves design and allows better refactoring of code.
- Unit Tests, when integrated with build gives the quality of the build as well.

Unit Test LifeCycle



UNIT TEST LIFE CYCLE
Check Out Code from Repository
Make Changes
Execute Unit Tests
Fix Defects and Re-execute Unit Tests
Code review
Check in Code into Repository

- Black Box Testing - Using which the user interface, input and output are tested.
- White Box Testing - used to test each one of those functions behaviour is tested.
- Gray Box Testing - Used to execute tests, risks and assessment methods.



Mocks

- Mocking is primarily used in unit testing.
- An object under test may have dependencies on other (complex) objects.
- To isolate the behaviour of the object you want to test you replace the other objects by mocks that simulate the behavior of the real objects.
- This is useful if the real objects are impractical to incorporate into the unit test.
- In short, mocking is creating objects that simulate the behaviour of real objects.

At times you may want to distinguish between *mocking* as opposed to *stubbing*. There may be some disagreement about this subject but my definition of a stub is a "minimal" simulated object. The stub implements just enough behaviour to allow the object under test to execute the test.

A mock is like a stub but the test will also verify that the object under test calls the mock as expected. Part of the test is verifying that the mock was used correctly.

To give an example: You can stub a database by implementing a simple in-memory structure for storing records. The object under test can then read and write records to the database stub to allow it to execute the test. This could test some behaviour of the object not related to the database and the database stub would be included just to let the test run.

If you instead want to verify that the object under test writes some specific data to the database you will have to mock the database. Your test would then incorporate assertions about what was written to the database mock.

Integration Testing

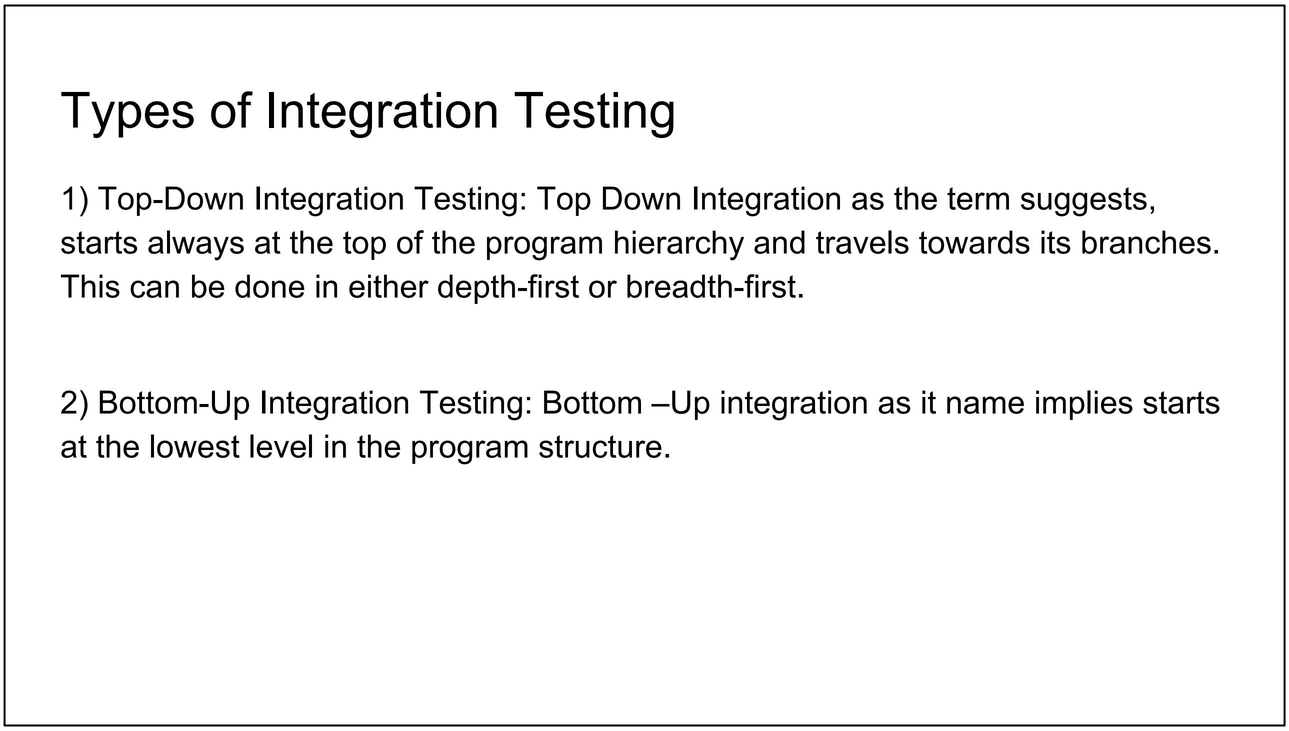
- Individual software modules are integrated logically and tested as a group.
- A typical software project consists of multiple software modules, coded by different programmers.
- Integration Testing focuses on checking data communication amongst these modules.
- It is also termed as **'I & T'** (Integration and Testing), **'String Testing'** and sometimes 'Thread Testing'.

Although each software module is unit tested, defects still exist for various reasons like:

- A Module in general is designed by an individual software developer whose understanding and programming logic may differ from other programmers. integration Testing becomes necessary to verify the software modules work in unity
- At the time of module development, there are wide chances of change in requirements by the clients. These new requirements may not be unit tested and hence system integration Testing becomes necessary.
- Interfaces of the software modules with the database could be erroneous
- External Hardware interfaces, if any, could be erroneous
- Inadequate exception handling could cause issues.

Integration Testing

- **Integration testing** is executed to establish whether the components interact with each other consort to the specification or not.
- Integration testing in large refers to joining all the components resulting in the complete system. It is further performed by the developer or the software Tester or by both.
- Example- checking that a Payroll system interacts as required with the Human Resource system.



Types of Integration Testing

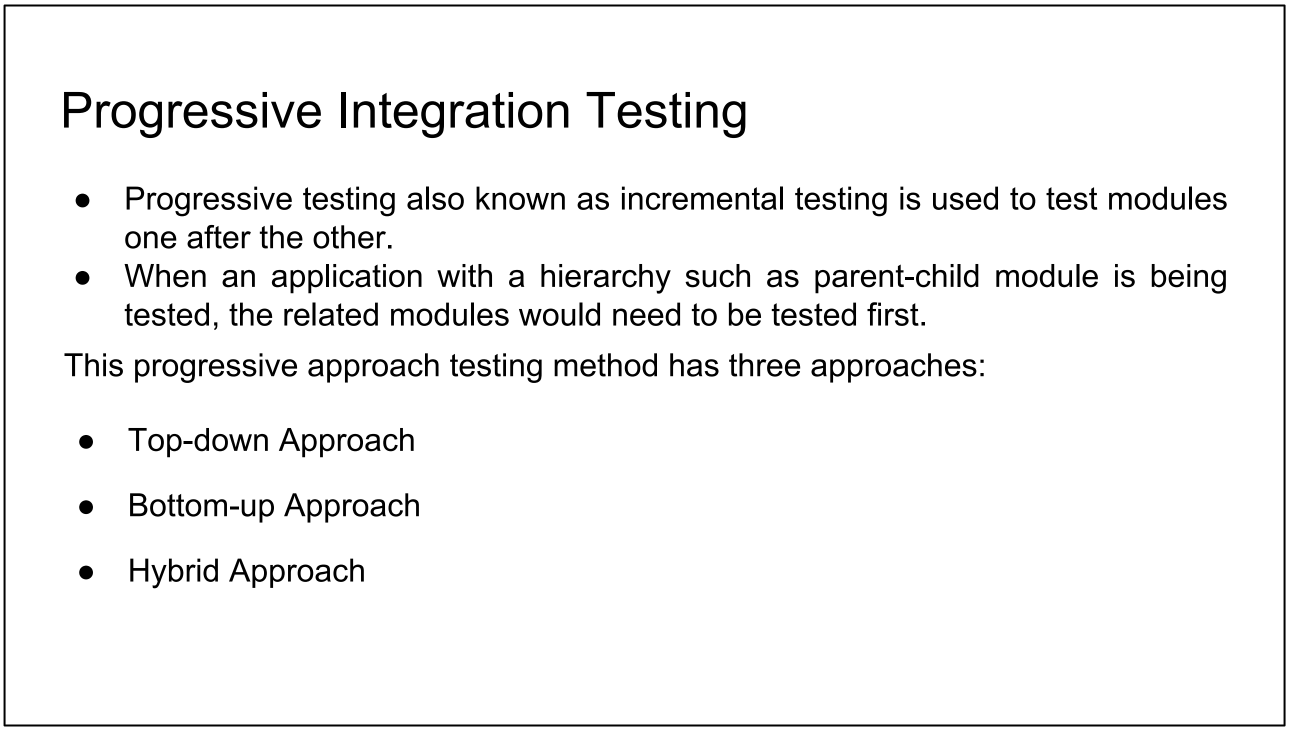
1) Top-Down Integration Testing: Top Down Integration as the term suggests, starts always at the top of the program hierarchy and travels towards its branches. This can be done in either depth-first or breadth-first.

2) Bottom-Up Integration Testing: Bottom –Up integration as it name implies starts at the lowest level in the program structure.

Integration Testing Example

For example you have to test the keyboard of a computer than it is a unit testing but when you have to combine the keyboard and mouse of a computer together to see its working or not than it is the **integration testing**. So it is prerequisite that for performing **integration testing** a system must be unit tested before.

Black-box test case design tactics are the most typical during integration, although limited amount of testing of white box may be used to ensure description of major control paths.



Progressive Integration Testing

- Progressive testing also known as incremental testing is used to test modules one after the other.
- When an application with a hierarchy such as parent-child module is being tested, the related modules would need to be tested first.

This progressive approach testing method has three approaches:

- Top-down Approach
- Bottom-up Approach
- Hybrid Approach

Progressive testing also called as incremental testing. Testing modules one after another in an incremental manner. In another word we test one module in the complete manner before attaching a new module to it and after adding new module we again test both modules as one module.

The goal behind Incremental testing is to give feedback to the developer after developing one module.

End-to-End Testing

- Term “*End to End testing”* is defined as a testing method which determines whether the performance of an application is as per the requirement or not.
- It is performed from start to finish under real-world scenarios like communication of the application with hardware, network, database and other applications.
- The main reason for carrying out this testing is to determine various dependencies of an application as well as ensuring that accurate information is communicated between various system components.
- It is usually performed after completion of functional and system testing of any application.

End to End testing of a Gmail account will include following steps:

1. Launching Gmail login page through URL.
2. Logging into Gmail account by using valid credentials.
3. Accessing Inbox. Opening Read and Unread emails.
4. Composing a new email, reply or forward any email.
5. Opening Sent items and checking emails.
6. Checking emails in Spam folder
7. Logging out of Gmail application by clicking ‘logout’

PHPUnit

What is PHPUnit

- PHPUnit is a unit testing framework written in PHP, created by Sebastian Bergman.
- Part of the xUnit family of testing frameworks.
- While there are other unit testing frameworks for PHP (such as SimpleTest or Atoum) PHPUnit has become the de facto standard.
- Major frameworks, such as Zend, Symfony and Cake, and many other PHP projects such as Doctrine have test suites written with PHPUnit.

PHPUnit is a collection of utilities (PHP classes and executable files) which makes not only writing tests easy (writing tests often entails writing more code than the application actually has – but it's worth it), but also allows you to see the output of the testing process in a nice graph which lets you know about code quality (e.g. maybe there's too many IFs in a class – that's marked as bad quality because changing one condition often requires rewriting as many tests as there are IFs), code coverage (how much of a given class or function has been covered by tests, and how much remains untested), and more.

Working with Selenium and other tools

Non-Functional Testing

Non-Functional Testing

- In **non-functional testing** the quality characteristics of the component or system is tested.
- Non-functional refers to aspects of the software that may not be related to a specific function or user action such as scalability or security. Eg. How many people can log in at once?
- Non-functional testing is also performed at all levels like **functional testing**.

Types of Non-Functional Testing

- Reliability testing
- Usability testing
- Efficiency testing
- Maintainability testing
- Portability testing
- Baseline testing
- Compliance testing
- Documentation testing
- Endurance testing
- Load testing
- Performance testing
- Compatibility testing
- Security testing
- Scalability testing
- Volume testing
- Stress testing
- Recovery testing
- Internationalization testing and Localization testing

Reliability testing

- Reliability Testing is about exercising an application so that failures are discovered and removed before the system is deployed.
- The purpose of reliability testing is to determine product reliability, and to determine whether the software meets the customer’s reliability requirements.

Usability testing

- In usability testing basically the testers tests the ease with which the user interfaces can be used. It tests that whether the application or the product built is user-friendly or not.

- Usability testing includes the following five components:
    - **Learnability:** How easy is it for users to accomplish basic tasks the first time they encounter the design?
    - **Efficiency:** How fast can experienced users accomplish tasks?
    - **Memorability:** When users return to the design after a period of not using it, does the user remember enough to use it effectively the next time, or does the user have to start over again learning everything?
    - **Errors:** How many errors do users make, how severe are these errors and how easily can they recover from the errors?
    - **Satisfaction:** How much does the user like using the system?

Performance Testing

- Performance testing is testing that is performed, to determine how fast some aspect of a system performs under a particular workload.
- It can serve different purposes like it can demonstrate that the system meets performance criteria.
- It can compare two systems to find which performs better. Or it can measure what part of the system or workload causes the system to perform badly.

Scalability Testing

It is the testing of a software application for measuring its capability to scale up in terms of any of its non-functional capability like load supported, the number of transactions, the data volume etc.

Endurance/Availability Testing

- Endurance testing involves testing a system with a significant load extended over a significant period of time, to discover how the system behaves under sustained use.
- For example, in software testing, a system may behave exactly as expected when tested for 1 hour but when the same system is tested for 3 hours, problems such as memory leaks cause the system to fail or behave randomly.

Security Testing

- Security testing is basically to check that whether the application or the product is secured or not.
- Can anyone came tomorrow and hack the system or login the application without any authorization.
- It is a process to determine that an information system protects data and maintains functionality as intended.

Stress Testing

- It involves testing beyond normal operational capacity, often to a breaking point, in order to observe the results.
- It is a form of testing that is used to determine the stability of a given system. It put greater emphasis on robustness, availability, and error handling under a heavy load, rather than on what would be considered correct behavior under normal circumstances.
- The goals of such tests may be to ensure the software does not crash in conditions of insufficient computational resources (such as memory or disk space).

Jenkins

Jenkins and Virtual Machines

Jenkins - Testing with Selenium RC