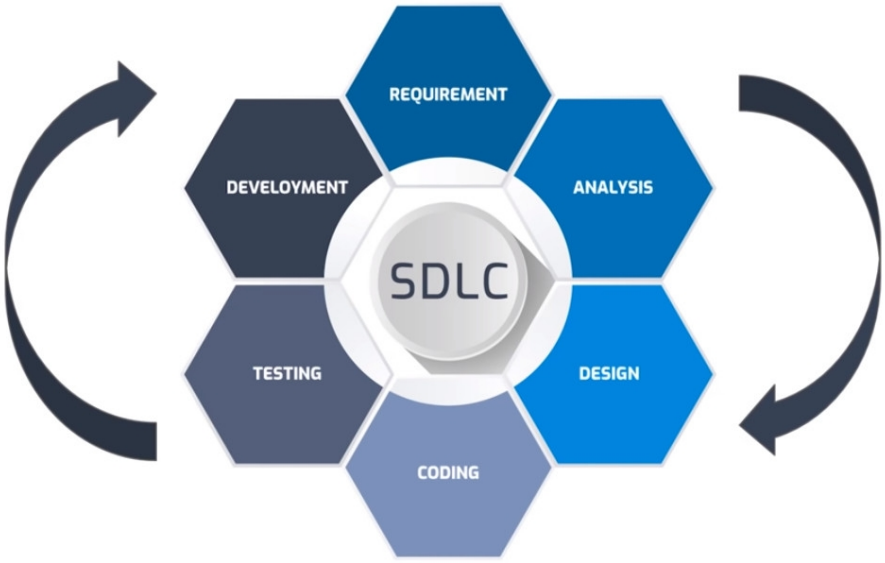
Introduction to Software Testing

Software testing is an activity to check whether the actual results match the expected results and to ensure that the software system is defect-free.

Software Development Life Cycle (SDLC)



Requirement: This is the most important phase. In this phase, you collect the requirements for the application from the client, based on their needs.
Analysis: To define and document requirements and get them approved by the client. This is done using a Software Requirements Specifications Document (SRSD). The SRSD contains all the product requirements to be designed and developed during the SDLC.
Design: This has two parts: **High-Level Design (HLD)** and **Low-Level Design (LLD)**

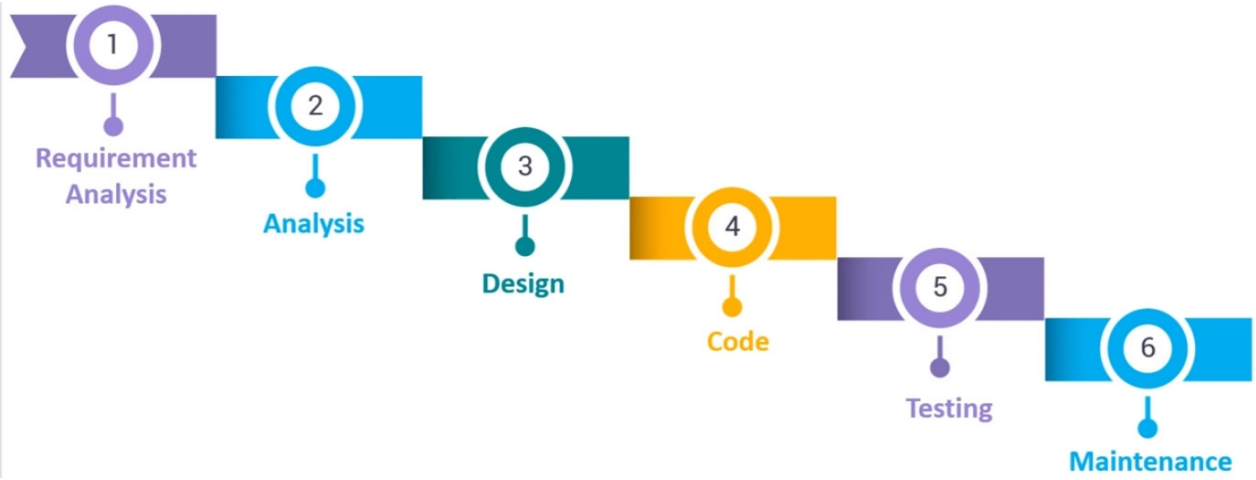
- **HLD** gives the architecture of the software product to be developed and is done by the architects and senior-level developers.
- **LLD** is done by the senior-level developers. The LLD describes how each and every feature of the product should work and how every component should work. This is only the design and not the code.

The the completion of these phases, you now have a completed High-Level Design document and Low-Level Design document which will work as inputs in the the next phase of development.
Coding: In this phase, you start writing the software and code for the product. The outcome of this phase is the source code for the project and the developed product.
Testing: Once coding is complete, it is thoroughly tested for anomalies and defects. This can be done with software or manually. The goal is to ensure the code is error-free. The outcome of testing is a quality product with the testing artifacts.
Deployment: Once the product is determined to be fully tested and of high quality, it is deployed. Once deployed, it will be free to use by the client and this is when user bugs and issues will arise. Fixing these issues comes in the maintenance phase. 100% testing is not possible because the way users use the software is different from how testers test the software. These maintenance fixes are handled on a case-basis.

Models of the Software Development Life Cycle

Waterfall Model

The waterfall model is a project management methodology based on sequential design.



Every next phase of this method is started only when the previous phase is completed.

This method is preferred when quality is preferred over schedule or cost. It is also preferred for projects that are short-term and the requirements will not change.

The Pros of the Waterfall Method

- Requirements don't change, nor does the design or code.
- The requirements are finalized earlier in the life cycle.
- Gives high visibility to the project manager and client about the progress of the application.

Main Drawbacks

- Backtracking is not possible. To backtrack would lead to defects in the product due to overlap in phases.
- The end-product may not be flexible.
- This model can only be used when the requirements are very well-known and fixed.
- Not suitable for long-term projects where requirements may change.

Boehm Spiral Model

The Boehm Spiral Model is a combination of the iterative development process model and sequential linear development model. It is a combination of both the prototype development process and linear development process similar to the waterfall model.

This model places more emphasis on risk analysis. Typically used in large and complex projects where risk is very high. Every iteration starts with planning and ends with the product evaluation by the client.

The Pros of the Boehm Spiral Model

- Allows for changes in the requirements and is suitable for large and complex projects.
- Allows for better risk analysis and is cost-effective due to good risk management.

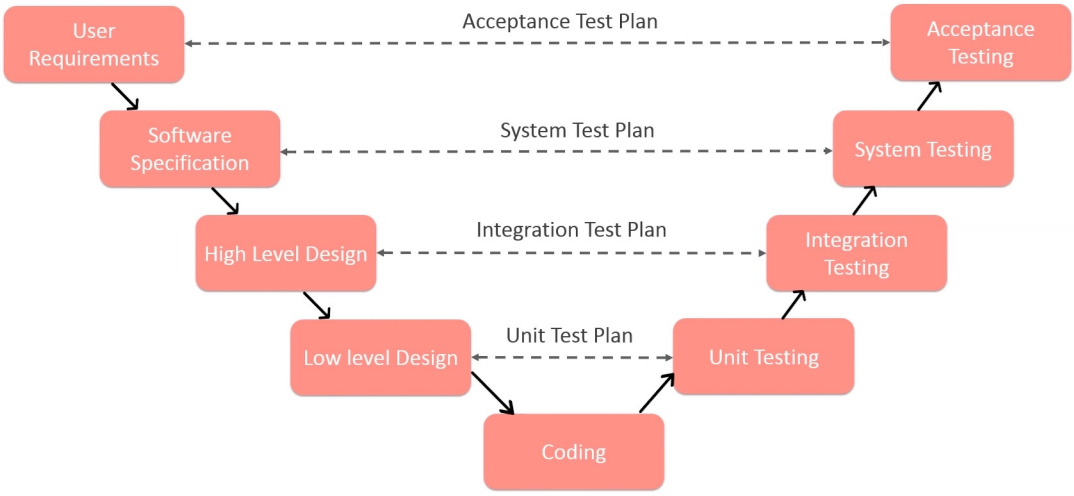
Main Drawbacks

- Not suitable for small projects.
- The success of a project depends on the risk analysis phase.

Principles of Software Testing

1. ### Detection of Bugs
   Testing any software or project can help in revealing defects that may or may not be detected by developers. However, testing of software alone cannot confirm that your developed product or software is error-free. Hence, it is essential to devise test cases and find out as many defects as possible.
2. ### Effectiveness testing
   Until your project or application under test has a straight forward structure having limited input, it won't be likely or achievable to check and test all feasible sets of data, modules, and scenarios.
3. ### Early Testing
   The earlier you begin to test your project or software, the better you will be able to utilize your existing time.
4. ### Defect in Clustering
   At the time of testing, you can observe that a majority of the defects or bugs that are reported are because of a small number of modules inside your software or system.
5. ### Testing is Context-Dependent
6. ### Error-Free Testing is a Myth
   Just because a tester didn't detect or no longer detects any defects in the project, it doesn't indicate or imply that your software is ready for shipping.
7. ### 100% Quality
   At the time of testing modules or working of software, you as a tester need to test whether your software is meeting all of the requirements of the client or not and whether the bugs found during testing have been mended or not. These many factors need to be considered before shipping the software or releasing to the market.

Verification and Validation Model (V-Model)



Verification: A static analysis technique that is testing without using code.
Validation: Dynamic analysis technique where testing is done by executing code.

On the diagram, the activities on the left represent development activities, where the activities on the right represent testing activities. During verification and validation, both development and QA are done simultaneously. There is no discrete phase called “testing”. That is, testing starts right from the requirement phase.

User Requirements

In this phase, how the system is implemented is not important, but what the system is supposed to do is important. Brainstorming, interviews, and walk-throughs are done here to have the objective set clearly.
Verification: Requirement reviews
Validation: Creation of the user acceptance test and it’s test cases
Artifacts: Requirement understanding document and User Acceptance Test test cases (UAT test cases)

Software Specification

At this phase, a high-level design of the software is built and the team studies and investigates how the requirements could be implemented, and the technical feasibility of the requirements is also studied. As well, the team comes up with the modules that would be created based on the software and hardware needs.
Verification: Design reviews
Validation: Creation of system test plan and cases and creation of traceability matrix.
Artifacts: System test cases, feasibility reports, system test plan, hardware and software requirements, and modules to be created.

High-Level Design

Based on the previous step, the high-level design software architecture is created. The modules, their relationships and dependencies, architectural diagrams, database tables, and technology details are all finalized in this phase.
Verification: Design reviews
Validation: Integration test plan and test cases
Artifacts: Design documents, integration test plan and test cases, and database table designs, etc.

Low-Level Design

In this phase, each and every model of the software component are designed individually. Methods, classes, interfaces, data types, etc., are all finalized in this phase.
Verification: Design review
Validation: Creation and review of unit test cases
Artifacts: Unit test cases

Coding/Implementation

In this phase, actual coding is done.
Verification: Code and test case review
Validation: Creation of functional test cases
Artifacts: that are produced include test cases and review checklist

Unit Testing

In this phase, all the unit test cases created in the low-level design are executed. Unit testing is a white-box testing technique where a piece of code is written which invokes a method to test whether the code snippet is giving the expected output or not. In case of any anomaly, defects are logged and tracked.
Artifacts: Unit test execution results

Integration Testing

In this phase, the integration test cases are executed which were created in the architectural design or your high-level design phase. In case of any anomalies, defects are logged and tracked. In integration testing, it validates whether the components of the application work together as expected.
Artifacts: Integration test results

Systems (End-to-end) Testing

In this phase, all the system test cases, functional test cases, and non-functional test cases are executed. In other words, the actual and full-fledged testing of the application takes place here. Defects are logged and tracked for its closure. Progress reporting is also a major part in this phase. The traceability metrics are updated to check the coverage and risk mitigated.
Artifacts: E2E test results, test logs, defect report, summary report, and updated traceability matrices

Acceptance Testing
This phase of testing is basically related to business requirement testing. Here, testing is done to validate that the business requirements are met in the user environment and compatibility testing and sometimes non-functional testing are also done in this phase.
Artifacts: User acceptance test results and updated business coverage matrices

That is why this is called the V-model, where verification is nothing but the development phase, and validation is the testing phase. So when do you use the V-model? It is to be used when the requirements are well-defined and not ambiguous, acceptance criteria are well-defined, the project is short to medium in size, and technology and tools used are not dynamic.

Software Testing Life Cycle
The Software Testing Life Cycle (STLC) identifies what test activities to carry out, what to accomplish, and when to accomplish those activities.

1. ### Requirements Analysis
   This is the very first step in the software testing life cycle. In this step, the Quality Assurance (QA) team understands the requirement in terms of what what they will be testing and figure out the testable requirements. If there are any conflicts or missing or not understood requirements, the QA team follows up with the various stakeholders such as business analysts, system architects, client, or technical manager to better understand the detailed knowledge of the requirement.
2. ### Test Planning
   Test planning is the most important phase of the software testing life cycle where all testing strategy is defined. This test is also known as the test strategy phase. In this phase, the test manager is involved to determine the effort and cost estimates for the entire project. This phase is started once the requirement gathering phase is completed, and based on the requirement analysis you can start preparing the test plan. The result of this planning phase will be the test plan or press strategy or testing effort estimation documents. Once this phase is complete, the QA team can then start with the development of test cases.
3. ### Test Case Development
   In this phase, the testing team writes down the detailed test cases. Along with test cases, the testing team also prepares test data for testing. Once the test cases are ready, they can be reviewed by peer members or the QA lead and also the requirement traceability matrix is prepared. The requirement traceability matrix (RTM) is an industry accepted format for tracking requirements where each test case is marked with the requirement. Using this, the RTM can track backward and forward traceability.
4. ### Environment Setup
   This is a vital part of STLC, where basically the test environment decides on which condition software is tested and is an independent activity that can be started in parallel with test case development.
5. ### Test Execution
   Once the preparation of test case development and the test environment setup is completed, then test execution phase may take place. In this phase, the testing team starts executing test cases is based on the prepared test plan and prepared test cases in the prior step.
6. ### Test Cycle Closure
   Once a test case has passed, it can be marked as passed. If any test case fails, then the corresponding defect can be reported to development team by a bug tracking system, and the bug can be linked for corresponding test cases for further analysis.

Software Testing Methods

Black-Box Testing
Also known as behavioral testing, black-box testing is where the software tests the internal structure, design and implementation, and UI and UX of the product being tested which is not already known to the tester. So, an input is being put into a function where the internal details or implementation of what is being tested are unknown to the user, and an output is being returned.

White-Box Testing
Also known as glass box testing, this type of testing technique deals with the testing of the internal structure, logical design, and implementation of different modules. In this type of test, the internal implementation details are known to the tester.

Grey-Box Testing
In this software testing technique, it combines concepts of both black-box and white-box testing, where the internal implementation of what is being tested is partially known by the user.

Functional & Non-Functional Testing

Functional
- Performed before non-functional testing
- Based on customer requirements
- Describes what the product does
- Unit testing, acceptance testing, smoke testing, integration testing, regression testing

Non-functional
- Performed after functional testing
- Based on customer expectations
- Describes how the product works
- Performance testing, scalability, volume testing, load testing, stress testing

Software Testing Levels

Unit Testing
Unit test cases are produced, testing small chunks of isolated code to ensure proper execution.

Integration Testing
Once the units from unit testing have passed, they are then integrated into more complex tests and again tested against each other.

Regression Testing
In the event an application or code in an application has been changed, regression testing is used to ensure these changes have not adversely affected existing features or functionality of the application.

System (E2E) Testing
Once the integrated tests have passed, the software application as a whole is tested from end-to-end (E2E) as a fully functional application.

Acceptance Testing
Once the application has passed its system test, it is then tested to ensure the product performs according the expected requirements. Often the client is involved to ensure they are satisfied with the final product.

Software Testing Documentation

Documentation Artifacts

Test Plan
Provides the outline strategy which will be implemented for testing the application. Also holds the details on the environment in which the tests will be performed.

Test Scenario
It can be considered as a single-lined statement which notifies the area in which your application will experiment. This artifact is needed to ensure the overall test procedure from start to finish.

In this scenario, you will need to have your use case ID, requirement ID, the scenario you are working in, and the number of test cases required. This allows you to clearly and concisely document all the records in the test scenario.

Test Case
Test cases engage in collected steps and conditions with inputs which can be implemented at the time of testing. This activity focuses on making sure whether a product went through a set of tests or faced by any means such as functionality or other aspects. Many types of test cases are being checked during testing, such as functional test cases, negative error cases, logical and physical test cases, and user interface test cases.

Necessary columns for documenting test cases: Test case ID, test case, description, steps taken for testing a particular software or a particular test case, expected result, actual result, status, comment.

Traceability Matrix
Contains a table which sketches the requirements when your product's SDLC model is being created. Also known as the **requirement traceability matrix**. This artifact can be implemented for forward tracing, which is to go from designing or can be implemented for backward tracing, which is to go from a more advanced phase and work backwards towards the design phase.

When implementing this matrix, you will have n-number of test cases as well as business requirements. The matrix will be put if the result will be passed on both ends (forward and reverse ends).

Defect Management
Defect management can be defined as a process of detecting bugs and fixing them. It is necessary to say bugs occur constantly in the process of software development. They are a part of the software development industry and that is because of the fact that software development is a quite complex process. The process of defect management is usually conducted at the stage of product testing. Without realizing, it would be hard to understand the nature of defect management.

Software testing can be conducted in two different ways. Usually, the developers test the product themselves. However, there is also a type of testing that is based on user involvement. The final users are often provided the ability to report on the bugs they find. However, this isn't a good way of testing, as users will likely not be able to find all defects in the application.

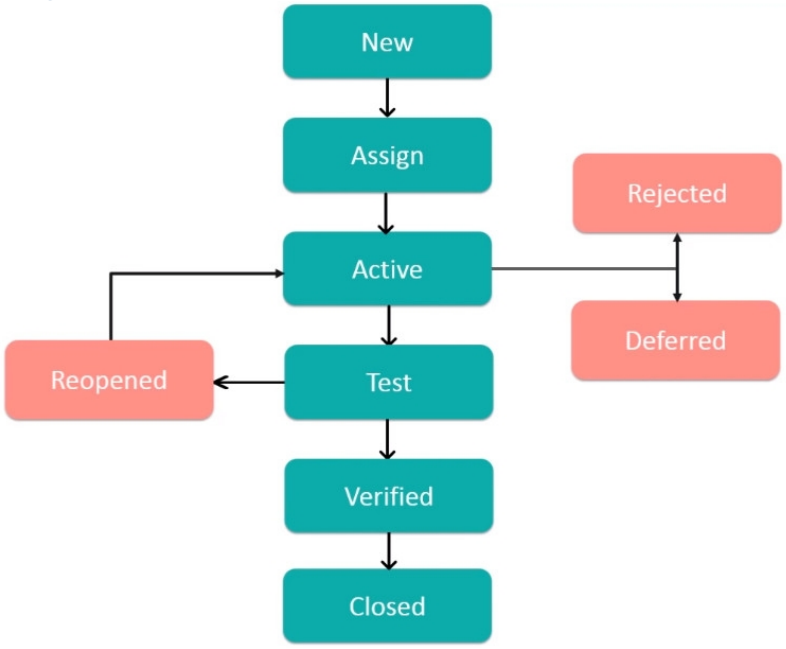
Defect Management Process
Defect Detection
Defect detection can be handled by the software developers or the users. The main goal is to detect all the bugs in the final product.

Formulation of Bug Reports
These are the documents that include all necessary information about certain bugs. Usually, they contain data on the type of bug and the possible direction to its correction.

Bug Fixing
After a bug is fixed, it should be tested again to ensure the application is working properly.

Bug List Creation
This is a document that contains information about all the bugs that occur during the project's performance and the team often uses the bug list because similar bugs have occurred.

Defect/Bug Life Cycle

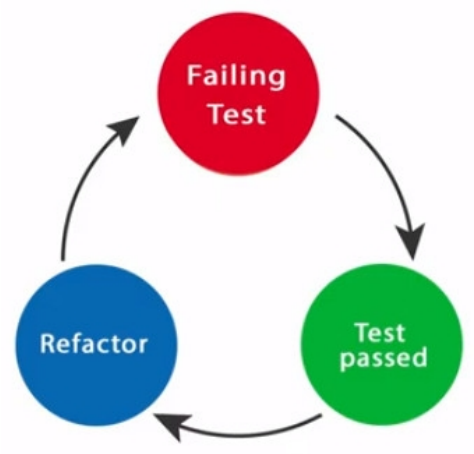


New: When a defect is logged and posted for the first time.
Assign: The lead of the testers approves that the bug is genuine and assigns it to a developer.
Active/Open: The developer has started analyzing and working on the defect fix. Once the developer has made satisfactory coding changes on the defect, they may mark the bug status as fixed, which then the bug is passed to the testing team.
Test: The tester now tests the changed code that they have been given to check whether the defect has been fixed or not. The tester then determines whether the bug has not been sufficiently fixed and reopens the case, or verifies the bug fix has passed, thus can be sent to be verified.
Reopen: The tester determines the bug has not been sufficiently fixed and reopens the defect case.
Verified: After the software has been tested many times to ensure it is free from errors, the case may be marked to be closed.
Closed: This means the bug's status of being tested and verified has been approved and the issue is resolved and closed.
Rejected: If the lead developer feels that the bug is not genuine, they may choose to reject the bug and mark it as rejected.
Deferred: The developer acknowledges the bug, but expects it to be fixed in future releases of the application. Some of the reasons the bug may be set to deferred if a new release is to be issued soon, the priority of the bug is low, or the bug has little or no major effect on the application.

Test Driven Development

Test Driven Development (TDD) is a development life cycle in which you write your test cases before you actually write your implementation.

- First, you have to red state, which means to first write a failing test. This test is written in a way that says what your code should be doing. In other words, how a behavior should operate or what a piece of software or a unit in your code should be doing.
- Next, you make this test pass and you write the minimum amount of code to make to make the red test a green test (test passed).
- The next state is the refactor (blue) state. During refactoring, you will be modifying the code you wrote at the green state to make your test pass, and you're going to refine it to be better structured and overall cleaner than the green state. This, of course, is only when applicable.
- Once the refactoring stage is completed, you return to the failing test and write a new test and repeat the cycle again-and-again until the software is complete.



Following this cycle of development you achieve many things, but most importantly you eradicate the fear of change, because these tests are always true. As your application grows in size and complexity, by testing at every change and inclusion in your application, if a test fails at a certain point in development you can zero-in on the most recent changes to the code to diagnose the issue. This in turn makes refactoring safer and easier in the long run.

Code Kata

A code kata is an exercise in programming which helps programmers hone their skills through practice and repetition. By incorporating this exercise and practice into your everyday programming, you will become more comfortable and confident in all areas in software engineering.

Behavior Driven Development

Behavior Driven Development (BDD) is somewhat similar and complimentary to TDD. However, where TDD focuses on the development of individual functions in code, BDD has a similar process that focuses on the features of the application.

BDD often relates more towards natural language, such as when working with clients or when thinking about general application features and ideas. In order for BDD to work, there needs to be some structure there that bridges the language of programming to natural spoken language.

For example, a client may want to develop a login page when if the login credentials are correct, they are shown the application dashboard. If the login credentials are invalid, they are shown an error message. Although these words are in plain English, the requests of the client are well structured and understood.

These requests, or feature files, are then turned into documentation on the application. But unlike most documentation, these documents will be tested against your program. If this documentation ever gets out of date, then something is liable to break in your testing stage. Any change made to the software gets verified by every example that was created in this process of documenting the requirements as an executable specification.

Not only does BDD streamline the requirements process, but it brings the value of QA to the beginning of development. The ability to execute every requirement against the application allows a team to regularly deploy new changes in days that may normally take weeks or months.

Dependency Injection

Dependency injection is the ability for on object to supply dependencies to another object. To break this concept down into simpler terms, we first need to define the term dependency and how it pertains to programming. A **dependency** is when one entity relies on the value or functionality of another entity in order to survive, function, or perform a task. In programming, if you have an object called ClassA that requires the functionality of an object

called ClassB (ClassA **depends** on ClassB), an instance of ClassB must first be instantiated and made available within ClassA.

So What Exactly is Dependency Injection?

Dependency injection (DI) is the process of passing a dependent object as a parameter to a method, rather than having the method instantiate the dependent object itself. What this means in practice is that the method does not have a direct dependency on a particular implementation; that is, any implementation that meets the requirements can be passed as a parameter. The “injection” refers to the passing of a dependency (a **service**) into the object (a **client**) that would use it.

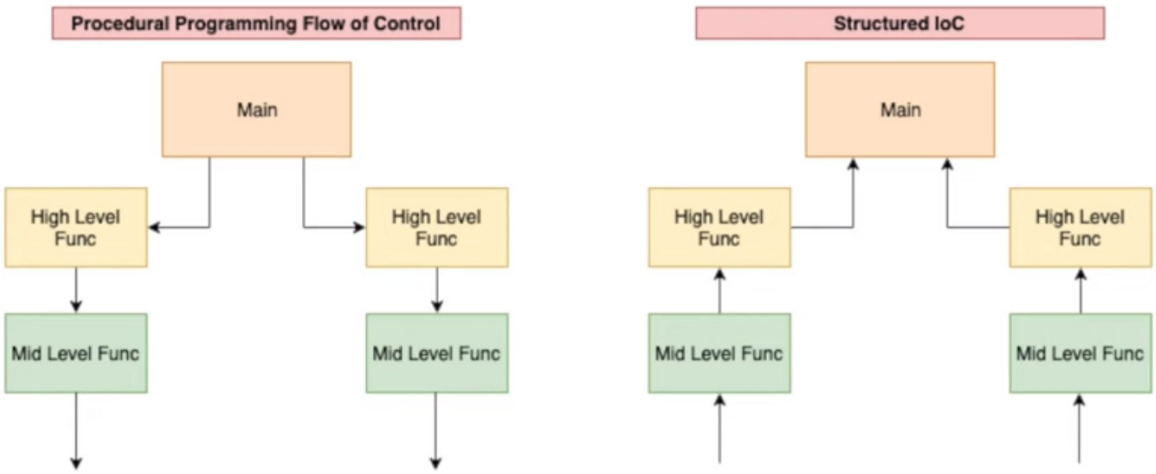
A very simple example: ClassA requires two dependencies that are ClassB and ClassC. While ClassA could instantiate both ClassB and ClassC objects within itself, instead it calls a method, telling it to instantiate and deliver (inject) the two class objects directly. This becomes incredibly useful later when it comes to scaling, refactoring, and testing code.

Dependency injection involves four roles:

- The **service** object(s) being used
- the **client** object that is depending of the service(s) it uses
- the **interfaces** that define how the client may use the services
- the **injector**, which is responsible for constructing the services and injecting them into the client.

Dependency Injection & Inversion of Control

Dependency injection is a method of **inversion of control (IoC)**, wherein IoC inverts the flow of control as compared to the traditional control flow. Therefore, a class should not configure its dependencies statically, but rather an outside class should configure them and pass them as parameters. A class should concentrate on fulfilling its responsibilities within the application and not on creating objects.



Types of Dependency Injection

Constructor

Dependencies are provided through a class constructor.

Setter

Injected method injects the dependency to the setter method exposed by the client. The setter injection is also known as the property injection because it utilizes the getter and setter methods of the class.

Interface

Injector uses an interface to provide the dependency to the client class. In this case, the client must implement an interface that will expose a setter method which will then access the dependency.

Benefits of Dependency Injection

Dependency injection provides the ability to change objects at run time based on requirements given by the client.

1. **Enables an easy way to interconnect the components of an application**
2. **Applications can be easily extended**
3. **Unit testing is made much easier**
4. **Reduces the amount of boilerplate code in an application**

Automation Testing with JUnit

Automation testing is an automated technique where the tester writes scripts and uses suitable software to test against their own software.

JUnit Testing

JUnit is a testing framework for the Java programming language. It is important for test-driven development. It is used to test a small chunk of code, known as a "unit". JUnit promotes the programming convention of "test first, code later."

Unit testing is used to verify a small chunk of code by creating a path, function, or a method. A unit is a natural abstraction of an object-oriented system that is the Java class or object.

The Advantages and Uses of JUnit

- It is one of the best testing frameworks that can be selected for an efficient testing process.
- More application developer IDEs include JUnit.
- JUnit provides a text-based command line.
- JUnitEE test framework enables JUnit to test within the application server's container.
- JUnit is widely adopted by many organizations around the world.
- JUnit is a benchmark for testing in the Java Programming Language.

Features of JUnit

- It allows you to write code faster, which increases quality.
- It is elegantly simple.
- It provides annotations to identify test methods.
- It provides test runners for running the tests.
- Tests can be run automatically.
- Tests can be organized into test suites.

The JUnit Framework

JUnit is a regression testing framework which is used to implement unit testing in Java. This framework also allows quick and easy generation of test data and test cases.

Test Fixtures

A **fixture** is a fixed state of an object which is used as a baseline for running the test cases. **Test fixtures** ensure that there is a well-known and fixed environment in which tests are run so that results are repeatable.
There are two methods under this test fixture:

- `setUp()` `// @Before`
- `tearDown()` `// @After`

Test Suites

If you want to execute multiple test cases in a specific order, it can be done by combining all the test cases in a single origin. This origin is called a **test suite**. A test suite bundles a few unit test cases and then runs them together. To run the test suite, you need to annotate the class using the annotations:

- `@RunWith`
- `@SuiteClasses`

Test Runners
The **test runner** is used for executing the test cases.

- `JUnitCore` is used to execute the tests.
- `runClasses` method is used to run one or several test cases.
- The return type of this method is the result object that is used to access information about the tests.

JUnit Classes
JUnit classes are used when writing and testing JUnits. Some of the important classes are:

- `assert`
- `testCase`
- `testResult`

`testCase` contains the test case that defines the fixture to run multiple test cases.
`testResult` contains methods in order to collect the result of executing a test case.

JUnit Annotations

An **annotation** is a special form of syntactic metadata that can be added to the Java source code for better code readability. Some annotations include:

`@Test`	Tells JUnit that the `public void` method can be run as a test case.
`@Before`	Annotating a `public void` method with `@Before` causes that method to be run before each `@Test` method
`@After`	If you allocate external resources in a `@Before` method, you need to release them after the test runs. Annotating the method with `@After` causes that method to be run after the `@Test` method
`@BeforeClass`	Annotating a public static void method with `@BeforeClass` causes it to be run once before any of the `@Test` methods in the class.
`@AfterClass`	This will perform the method after all tests are run. This can be used to perform clean-up activities.
`@Ignore`	The `@Ignore` annotation is used to ignore the test and that test will not be executed.

JUnit Assert Statements

Assert is a method used in determining the pass or fail status of a test case. In JUnit, all assertions are in the assert class. Only failed assertions are recorded.

List of assert statements (all `void` methods)

`assertEquals(boolean expected, boolean actual)`	Checks that two primitives/objects are equal
`assertTrue(boolean condition)`	Checks if condition is true
`assertFalse(boolean condition)`	Checks if condition is false
`assertNull(Object Object)`	Checks whether an object is null
`assertNotNull(Object Object)`	Checks whether an object is not null
`assertSame(Object1, Object2)`	Tests if two object references point to the same object
`assertNotSame(Object1, Object2)`	Tests if two object references do not point to the same object
`assertArrayEquals(expectedArr, resultArr)`	Tests whether two arrays are equal to each other

Exceptions

You can test whether the code throws the desired exception. The expected parameter is used along with `@Test` annotation. While testing for an exception, you need to ensure that the exception class that you're providing in that optional parameter of the test annotation is the same, because you are expecting an exception from the method that you're testing. Otherwise, the JUnit test will fail.

Parameterized Test

A parameterized test allows a developer to run the same test over-and-over again using different values. The steps to use parameterized tests are as follows:

1. Annotate test class with `@RunWith`

2. Create `@Parameters` that return a collection of objects in the form of an array as a test data set.
3. Create a public constructor that takes in what is equivalent to one row of the test data.
4. Create an instance variable for each and every column of test data.
5. Create test cases using these instance variables as the source of the test data.

Automation Testing with Mocha

Mocha is a feature-rich JavaScript test framework running on Node.js and the browser. Mocha test runs serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases. By design, Mocha is simple, with a relatively thin API that is easy to learn.

- Mocha does **NOT** contain its own assertion library, however there are plenty of great assertion libraries available such as `Node.js assert` and Chai, and you can choose any that you'd like.
- Mocha provides useful testing the hooks `before()`, `after()`, `beforeEach()`, and `afterEach()`.
- Mocha makes it easy to test asynchronous code (Callbacks, promises, and Async/Await).
- Mocha provides extensible reporting.

Testing Fundamentals

Assertions
A comparison which throws an exception upon failure.

Unit Test
Asserts a unit behaves as intended. Typically a single function.

Integration Test
Asserts an aggregate of units or modules behave as intended.

Mocha Uses GLOBALS
A decision by Mocha developers is that Mocha does not have to be namespaced.

Create a Suite

This is a way to organize your tests

```
describe(title, callback)   // Creates a new suite
```

The callback is the body of the function:

```
// test/request-time.spec.js
describe('requestTime middleware', function() {
    // Tests go here
});
```

Mocha does **NOT** have any functionality in it to make assertions. For the first two examples, we will use Node.js' built-in `assert` module. Later, we will be using the Chai library.

Unit Test Using Mocha & Node.js `assert`

First, we will need our middleware:

```
const assert = require('assert');
const requestTime = require('../lib/request-time);
```

… Where is "`const mocha = require('mocha');`"?

Mocha is **GLOBAL** (no "require" required).

Second, we create a test with `it(title, callback)`

```
// test/request-time.spec.js
const assert = require('assert');
const requestTime = require('../lib/request-time');

describe('requestTime middleware', function() {
    it(`should add 'requestTime' property to the 'request' parameter`, function() {
        // call function
        const request = {};
        requestTime(request, null, function() {});

        // make assertion
        assert.ok(req.requestTime > 0);
    });
});
```

An Integration Test (Callback-style)

The next level up after a unit test. It's a higher level view where units are compounded and tested. The middleware **supertest** is used for integration tests against Express servers.

```
// test/unix-timestamp-route.spec.js
const assert = require('assert');
const app = require('../app');
const request = require('supertest');

describe('GET/unix-timestamp', function() {
    it('should respond with JSON object containing timestamp', function(done) {
        // assertion goes here
        request(app).get('/unix-timestamp)
        .expect(200).end((err, res) => {
            if(err)
                return done(err);
            assert.ok(res.body.timestamp < 1e10);
            done();
        });
    });
});
```

An Integration Test (Promise-style)

```
describe…
    it('should respond with JSON object containing timestamp', function(done) {
        request(app).get('/unix-timestamp')
            .expect(200).end((err, res) => {
                if (err)
                    return done(err);
                assert.ok(res.body.timestamp < 1e10);
                done();
            });
    });
```

Testing RESTful APIs Using Mocha & Chai

Chai Assertion Styles

Should

```
chai.should();

foo.should.be.a('string');
foo.should.equal('bar');
foo.should.have.lengthOf(3);
tea.should.have.property('flavors'
).with.lengthOf(3);
```

Expect

```
const expect = chai.expect;

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.lengthOf(3);
expect(tea).to.have.property('flav
ors').with.lengthOf(3)
```

Assert

```
const assert = chai.assert;

assert.typeOf(foo, 'string');
assert.equal(foo, 'bar');
assert.lengthOf(foo, 3);
assert.property(tea, 'flavors');
assert.lengthOf(tea.flavors, 3);
```

Chai Language Chains

When using the BDD styles Should and Expect, there is a list of chainable getters provided to improve the readability of your assertions. These chains include:

- `to`
- `be`
- `been`
- `is`
- `that`
- `which`
- `and`
- `has`
- `have`
- `with`
- `at`
- `of`
- `same`
- `but`
- `does`
- `still`
- `also`

Chai BDD Assertions

Chai assertions are a library of methods used to test whether an expected result of an object, array, value, or function is true or false. If false, an error will be thrown. This is a list of assertions used with the BDD assertion methods in Chai. Note that many of these assertions test conditionally and/or within ranges. It is recommended by the developers themselves to assert an exact amount rather than within a bound.

Full documentation and examples may be viewed here: https://www.chaijs.com/api/bdd/

`.not`	Negates all assertions that follow a chain
`.deep`	Causes all `.equal`, .include, .members, .keys, and .property assertions that follow in the chain to use deep equality instead of strict (===) equality.
`.nested`	Enables dot- and bracket-notation in all `.property` and `.include` assertions that follow in the chain.
`.own`	Causes all `.property` and `.include` assertions that follow in the chain to ignore inherited properties.
`.ordered`	Causes all `.members` assertions that follow in the chain to require that members be in the same order.
`.any`	Causes all `.keys` assertions that follow in the chain to only require that the target have at least one of the given keys. This is the opposite of `.all`, which requires that the target have all of the given keys.
`.all`	Causes all `.keys` assertions that follow in the chain to require that the target have all of the given keys. This is the opposite of `.any`, which only requires that the target have at least one of the given keys.
`.a(type[,msg])`	Asserts that the target's `type` is equal to the given string type. Types are insensitive. `.a` supports objects that have a custom type set via `Symbol.toStringTag`. It's often best to use `.a` to check a target's type before making more assertions on the same target. That way, you avoid unexpected behavior from any assertion that does different things based on the target's type. Add `.not` earlier in the chain to negate `.a`. However, it's often best to assert that the target is the expected type rather than asserting that it isn't one of many unexpected types. `.a` accepts an optional `msg` argument which is a custom error message to show when the assertion fails. The message can be given as a second argument to expect. `.a` can also be used as a language chain to improve the readability of your assertions. The alias `.an` can be used interchangeably with `.a`.
`.include(val[, msg])`	Depending of the variable type, .include asserts that the given val is a member, or subset, of the target. Full documentation: https://www.chaijs.com/api/bdd/#method_include
`.ok`	Asserts that the target is a truthy value. However, it's often best to assert that the target is strictly (===) or deeply equal to its expected value. Add `.not` earlier in the chain to negate `.ok`.
`.true`	Asserts that the target is strictly (===) equal to `true`.
`.false`	Asserts that the target is strictly (===) equal to `false`.
`.null`	Asserts that the target is strictly (===) equal to `null`.
`.undefined`	Asserts that the target is strictly (===) equal to `undefined`.
`.NaN`	Asserts that the target is exactly `NaN`.
`.exist`	Asserts that the target is not strictly (===) equal to either `null` or `undefined`. However, it's best to assert that the target is equal to its expected value.
`.empty`	Various implementations to see if a string, array, map, or non-function object is empty. Full documentation: https://www.chaijs.com/api/bdd/#method_empty
`.arguments`	Asserts that the target is an `arguments` object.

`.equal(val[, msg])`	Asserts that the target is strictly (===) equal to the given `val`. The alias `.eq` can be used interchangeably with `.equal`.
`.eql(obj[,msg])`	Asserts that the target is deeply equal to the given `obj`.
`.above(n[,msg])`	Asserts that the target is a number or a date greater than the given number or date n respectively. However, it's often best to assert that the target is equal to its expected value.
`.least(n[,msg])`	Asserts that the target is a number or a date greater than or equal to the given number or date n respectively. However, it's often best to assert that the target is equal to its expected value.
`.below(n[,msg])`	Asserts that the target is a number or a date less than the given number or date n respectively. However, it's often best to assert that the target is equal to its expected value.
`.most(n[,msg])`	Asserts that the target is a number or a date less than or equal to the given number or date n respectively. However, it's often best to assert that the target is equal to its expected value.
`.within(start, finish[,msg])`	Asserts that the target is a number or date greater than or equal to the given number or date `start` and less than or equal to the given number or date `finish` respectively. However, it's often best to assert that the target is equal to its expected value.
`.instanceof( constructor[msg ])`	Asserts that the target is an instance of the given `constructor`.
`.property(name[ ,val[,msg]])`	Asserts that the target has a property with the given key `name`.
`.ownProperty Descriptor`	Asserts that the target has its own property descriptor with the given key name. Enumerable and non-enumerable properties are included in the search. Lots more documentation here: https://www.chaijs.com/api/bdd/#method_ownpropertydescriptor
`.lengthOf(n[, msg])`	Asserts that the target's `length` or `size` is equal to the given number `n`.
`.match(re[, msg])`	Asserts that the target matches the given regular expression `re`
`.string(str[, msg])`	Asserts that the target string contains the given substring `str`.
`.keys`	Asserts that the target object, array, map, or set has the given keys. Only the target's own inherited properties are included in the search. When the target is an object or array, keys can be provided as one or more string arguments, a single array argument, or a single object argument. In the latter case, only the keys in the given object matter; the values are ignored. Tons of documentation here: https://www.chaijs.com/api/bdd/#method_keys
`.throw`	Throws an error based on criteria. Many examples in the documentation: https://www.chaijs.com/api/bdd/#method_throw
`.respondTo( method[, msg])`	.respondsTo asserts that the target has a method or its prototype property has a method with the given name method as a parameter. Several use cases can be seen in the documentation here: https://www.chaijs.com/api/bdd/#method_respondto
`.itself`	Forces all `.respondTo` assertions that follow in the chain to behave as if the target is a non-function object, even if it's a function. Thus, it causes `.respondTo` to assert that the target has a method with the given name, rather than asserting that the target's prototype property has a method with the given name.
`.satisfy`	Invokes the given `matcher` function with the target being passed as the first argument, and asserts that the value returned is truthy. The alias `.satisfies` can be used interchangeably with `.satisfy`.
`.closeTo( expected, delta[, msg])`	Asserts that the target is a number that's within a given +/- `delta` range of the given number expected. However, it's often best to assert that the target is equal to its expected value.
`.members(set[, msg])`	Asserts that the target array has the same members as the given array `set`. Has many use cases that can be seen in the documentation: https://www.chaijs.com/api/bdd/#method_members
`.oneOf(list[, msg])`	Asserts that the target is a member of the given array `list`. However, it's often best to assert that the target is equal to its expected value.
`.change(subject [,prop[,msg]])`	When one argument is provided, `.change` asserts that the given function subject returns a different value when it's invoked before the target function compared to when it's invoked afterward. However, it's often best to assert that `subject` is equal to its expected value.
`.increase`	When one argument is provided, `.increase` asserts that the given function `subject` returns a greater number when it's invoked after invoking the target function compared to when it's invoked beforehand. .increase also causes all `.by` assertions that follow in the chain to assert how much greater of a number is returned. It's often best to assert that the return value increased by the expected amount, rather than asserting it increased by any amount.
`.decrease`	When one argument is provided, `.decrease` asserts that the given function `subject` returns a lesser number when it's invoked after invoking the target function compared to when it's invoked beforehand. `.decrease` also causes all `.by` assertions that follow in the chain to assert how much lesser of a number is returned. It's often best to assert that the return value decreased by the expected amount, rather than asserting it decreased by any amount.
`.by(delta[,msg] )`	When following a .increase, .decrease, or .change assertion in the chain, .by asserts that the subjected assertion is increased, decreased, or changed respectively by the given delta.
`.extensible`	Asserts that the target is extensible, which means that new properties can be added to it. Primitives are never extensible.
`.sealed`	Asserts that the target is sealed, which means that new properties can't be added to it, and its existing properties can't be reconfigured or deleted. However, it's possible that its existing properties can still be reassigned to different values. Primitives are always sealed.
`.frozen`	Asserts that the target is frozen, which means that new properties can't be added to it, and its existing properties can't be reassigned to different values, reconfigured, or deleted. Primitives are always frozen.
`.finite`	Asserts that the target is a number, and isn't `NaN` or positive/negative `Infinity`.
`.fail`	Throws a failure.

References:
Software Testing for Beginners: https://www.youtube.com/watch?v=T3q6QcCQZQg
Test Driven Development for Beginners: https://www.youtube.com/watch?v=y8TcPr73Bwo
Behavior Driven Development in 4 minutes: https://www.youtube.com/watch?v=ydddSkVz_a8
What is Dependency Injection? : https://www.youtube.com/watch?v=O9mqe53syGc
Dependency Injection & Inversion of control: https://www.youtube.com/watch?v=EPv9-cHEmQw
JUnit Tutorial: https://www.youtube.com/watch?v=SDwqcFwvwY0
Testing Node.js with Mocha: https://www.youtube.com/watch?v=Bs68k6xfR3E
Testing a RESTful API with Mocha and Chai: https://www.youtube.com/watch?v=I4BZQr-5mBY
Getting Started with Node.js and Mocha: https://semaphoreci.com/community/tutorials/getting-started-with-node-js-and-mocha

Further Reading:
Software Testing Help: https://www.softwaretestinghelp.com/
Mocha.js Documentation: https://mochajs.org/
Chai.js Documentation: https://www.chaijs.com/
Write Mocha tests with TypeScript: https://journal.artfuldev.com/unit-testing-node-applications-with-typescript-using-mocha-and-chai-384ef05f32b2
How to Write Testable Code and Why it Matters: https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters
Regression Testing: https://www.guru99.com/regression-testing.html