Testing is a MUST
Version 0.8

Harald Glab-Plhak<hglabplhak@icloud.com>

November 27, 2025

About automated test methods and more.....

Contents

1 Information

This document and the whole project including the source code is licensed under the MIT license. The only restriction is that it MUST not be used commercial. A copy and further enhancements are allowed as long as the license and the copy right is saved. I intended that these test descriptions project is used for educational purposes. I trust in the integrity of those who use or duplicate the project and develop on this project.

2 Abstract

The target of this project is to give an overview of testing and test automation. The project also shows pitfalls and a style of development making testing and automation of tests easier. The project is made to show up how testing can be introduced in a project or product so that there is a common sense in the teams what things and processes can be useful to have a real test culture. In a world there the projects and products get more and more complex in cause of using cloud technologies, machine learning and KI as well as massive parallel execution it is also more and more necessary to develop automated tests and processes which allow us to get the complex things handled. Also the definition of test cases down to the last detail are important as well as a detailed logging of test results. These all is defined more and more detailed and there are many test-theories and project development and integration theories and practises as DevOPS with CI/CD. Here the short descriptions as footnotes taken from the KI of Google. ***DevOPS***[1] and ***CI/CD***[2] . All these things are themselves very complex I try in this document to show up a path through this 'jungle' .

3 Introduction

3.1 The different Test Methods in Test automation (UNIT -Test , Functional Test, Performance Functional Test)

In order to get the goal of a clean and correct code and functionality we must have a look on the program logic in different views.

- We have a view where we look for a complete well defined transaction in the program and have to check wether input and output is correct. Here we use ***Functional Tests***.[3].
- The other view is more that we look at the atomic functions processing more or less complex logic. Here we also look for input / output of the functionality in that unit but we also keep in mind the internal logic and the functions which are called before / after and / or if third-party functionality is called which can cause that we not really test our own logic / flow of control. These kind of tests are called ***Unit Tests*** [4]

[1] ***DevOPS:*** is a set of practices and a culture that combines software Development and IT Operations to shorten the software development lifecycle and provide continuous delivery with high quality. It breaks down the traditional silos between these teams, fostering collaboration and shared responsibility to accelerate the delivery of software while improving reliability. Key practices include automation, continuous integration and delivery (CI/CD), and using tools to streamline the process from planning through deployment and monitoring.

[2] ***CI/CD:*** Continuous integration and continuous delivery (CI/CD) have revolutionized how software teams work together. Gone are the days of code integration headaches and repeated manual processes. CI/CD makes modern software development possible fast, reliable, and automated.

[3] ***Functional Tests.***: These tests to check a complete request or transaction to an application. They treat the methods called inside and the logic inside the borders of the transaction as a ***BLACK BOX***. So it does not matter what the different functions classes or whatever process in what place even if the result is correct.

[4] ***Unit Tests***: Unit Tests are the tests with which small Units are tested e.g. public functions

- And now to the last view I will mention here where are for sure many more. We have to look how high the throughput of our application must be and if we reach that goal with our logic and the fact how we realised it in detail. These are ***Performance Tests***[5] and. the best is to let them run after each change and prior deployment of each minor and major release

3.2 UNIT Testing in an automated environment

So now our development team coded something and the specification is clear and clean transformed to code. Ups what shall we do now ? We have to test it and it gets more and more and more. The code is for quality reasons debugged by the developers but we need something more efficient. So we search for a test-framework where we are able to define Tests for the specific functions. This test-framework can handle all these things if we use it right. The advantage of this way testing things is that the tests are coded in Python / RUST / Haskell / Java / Clojure / Racket / Scheme48 and other Scheme and LISP dialects / C / C++ and so on and executing them another time without code-change they will do the same and exactly the same. This is one problem for manual testing. The Tester is a human being and we are full of mistakes in our doing. For manual tests we have later a look on it you have to create for each test a step by step list which shows up a detailed description for each step. For ***UNIT Tests*** we also need a description what will happen step by step but we tell our coded test to do it for us thousands of times. Now have a closer look on UNIT Testing in Java and Clojure. For Java the most popular framework will be ***JUNIT*** this framework is grown to a very large extensive test environment.

Chapter 1: An Introduction to test types and automation of tests

4 The different ways to divide tests in categories

4.1 From the technical point of view

1. ***Simple technical difference:***
   - ***BLACK BOX Test***: This kind of test only knows about the interface to test that means the defined Input and the awaited Output
   - ***GREY BOX Test***: This kind of test only knows about the interface to test. and about some inside details that means the defined Input and the awaited Output with the background that some details are known. So it is something between BLACK-(GREY)-WHITE BOX. This kind of test category is found very often in reality.
   - ***WHITE BOX Test***: Here the test also takes care about the way the function is implemented in technical detail
2. ***UNIT Tests:*** These Tests check the functionality on a level of for example public functions on an object. Functions where access can be gained only by a hack (private functions) should never be tested directly in the normal case doing this is bad style.
3. ***Functional Tests:*** These Tests check the functionality of a greater part of. the application e.g. One specific transaction.

4.2 Another way to make categories is delivery, user and release oriented

1. ***Acceptance Test:*** Here the project or product is tested to be informed about usability and acceptance by the user

[5] ***Performance Tests***: These tests are designed in a way that the performance of Units as well as the performance of requests and transactions or e.g. import / export in bulk can be tested

2. ***Regression Test:*** This test is done always after one or more changes. Either in the service or its environment

3. ***Release Test:*** This tests are done for. each major or minor release. Here. the point is that it is tested if the new functionality works and if the existing things also work again. What is also tested is the migration from the old to the new release.

4. ***Performance Tests:*** In these tests the overall or partial performance is good and sufficient for the needs of the application and if e.g. a change in a function influences performance negative.

5 A example how to design and realise an automated test

As an algoritmic fuction example for testing we use the widely spreaded sorting algorithm Quick-Sort. I know this is a example of a very low level functionality which is covered already in the JDK since a long time e.g. : SortedList or the sort feature in collection streaming. But I use it because it is a simple example.

```
public void quickSort(int arr[], int begin,  int end) {
    if (begin <. end) {
        int partitionIndex = partition(arr, begin, end);
        quickSort(arr, begin, partitionIndex - 1);
        quickSort(arr, partitionIndex + 1, end);
     }
}
```

HINT for beginners: to get it running you need the partition method. For the beginners this algorithm is in APENDIX II.
So ok here we have now something to test. What we need now is a testing framework containing assertion functions and annotations to annotate test methods. For this we use the most popular Java Testing framework JUNIT. The most important annotations in JUNIT are the following method annotations. The methods need to be public:

- ***@Test***: Annotate the test method as a method containing a test
- ***@Before***: Execute before each test
- ***@After***: Execute after each test

Here first a example of a very tiny test:

```
@Test
public void simpleQSortTest() {
                Integer[] theArrayToSort = {6, 7, 4, 2, 78, 45 ,1 ,5,
                    15};
                Integer[] expectedSorted = {1, 2, 4, 5, 6, 7, 15, 45,
                    78};
                quickSort(theArrayToSort, 0, (arrayToSort.length - 1));
                assertThat(arrayToSort, is(expectedSorted)); // if
                    this assert fails quicksort does not
// work correctly
}
```

Here in this test the easiest case is tested. It is tested if the array given is sorted after quicksort. The other cases like given wrong data by type e.g. Strings instead of Integer are not tested. In our case this won't work because the whole thing won't even compile in that case. So to show how such a Test will work let us elaborate the whole function a bit. Here is our new function which is much more flexible :

```
public class QBubbleMerge<T> {
    private QBubbleMerge() {
    }

    public static Integer[] quickSort(Integer[] arr, SortDirection
       direction, Integer begin, Integer end) {
        int partitionIndex = partition(arr, direction, begin, end);
        quickSort(arr, direction, begin, partitionIndex - 1);
        quickSort(arr, direction, partitionIndex + 1, end);
        return arr;
    }

    private static Integer partition(Integer[] arr, SortDirection
       direction, Integer begin, Integer end) {
        int pivot = arr[end];
        int i = begin - 1;

        for(int j = begin; j < end; ++j) {
            Boolean swapit = Boolean.FALSE;
            switch (direction.ordinal()) {
                case 0 -> swapit = Boolean.FALSE;
                case 1 -> swapit = arr[j].compareTo(pivot) >= 0;
                case 2 -> swapit = arr[j].compareTo(pivot) <= 0;
            }

            if (swapit) {
                ++i;
                Integer swapTemp = arr[i + 1];
                arr[i + 1] = arr[end];
                arr[end] = swapTemp;
                return i + 1;
            }

            Integer swapTemp = arr[i];
            arr[i] = arr[j];
            arr[j] = swapTemp;
        }

        ++i;
        return i;
    }

    public static void main(String[] args) {
        quickSort(SortData.SORT_DATA, QBubbleMerge.SortDirection.DESC,
           0, SortData.SORT_DATA.length);
        quickSort(SortData.SORT_DATA, QBubbleMerge.SortDirection.ASC,
           0, SortData.SORT_DATA.length);
    }

    public static enum SortDirection {
        DEFAULT,
        DESC,
        ASC;
```

```
    }
}
```

5.1 And here are a few tests for it to see how UNIT Testing works:

Here are a scelleton how to test the UNIT with JUNIT in Java (The code will be discussed in the next chapters / sections):

```
package io.examples.sort;


import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class QBubbleMergeTest {

    private QBubbleMergeTest() {

    }

    @BeforeAll
    public static void beforeAll() {

    }


    @BeforeEach
    public void beforeEach() {

    }


    @Test
    public void testQuickSortSimple() {

    }

    @Test
    public void testBubbleSortSimple() {

    }

    @Test
    public void testMergeSortSimple() {

    }
```

5.2 First Resume

Ok now we have some Tests checking random some aspects. To get a real good coverage when testing a function we first have to know it's specification that means what is the input and what is the output of a function we test. We also have to know what happens in the function when processing the input

to get either the valid output or a clear error which really explains what is missing or wrong. The more parameters we have the more the complexity tests for the function grows. If we when have a function with 5 parameters and each parameter has 15 combinations of values (The parameter may be an Object reference) we get a exponential growth of possibilities what would went wrong.

The main error by people who have less experience in Testing is to check the parameters together only in one test function. ***THIS IS A NOT TO DO!!!!***. A better way is to make a test for each parameter.

To get the complexity broken down we should make tests where 4 parameters are in a good range and only one is out of range or has a value which does not fit the four others parameters values. So you might argue what is with the combination where two parameters or three parameters have values which do not fit in combination. For this cases we can write some generic tests which walk via recursion through each valid combination. But do not test the case in that way where one or two parameters have a value out of their range which is defined by the functionality and which values and combination of values the function covers. Even if you plan to do test driven development this kind of proceeding gets more important for otherwise you do not know in the end in which case and why the function fails. To be sure all works you have to write ***UNIT Tests***for each public method in each class.
The private methods are tested indirect. A bad style is (with only a very very few exceptions - should never happen) to access private functions for the reason of testing them via privileged access.
Ok this works in Java up to 1.8. But one thing has to be clear:***For making good functional unit or other testing in a proper way it should never be necessary to use algorithms which break the normal ways of coding. No good developer will design a program and the time he is ready use a function which accesses another function which is e.g. private and the language is not designed to do it in that way.***
If you see such code begin looking at the design of this part delete the code and rewrite it completely following the design in a clean way. Why write it completely new. One reason is if you do after these functions are written test driven development for some parts such parts will lead you off the road of a clean design.

6 How to design Automated Functional Tests

6.1 The Basics

For the automated Functional Testing we can set up a environment with e.g. at least ***JUNIT***. Now you may think JUNIT is only for Unit testing But for sure we can use it because the things we call inside a test are flexible. Now we do not call functions inside the Test but we call for example our WebService Application and check the HTTP and extended Result given by the response data for example JSON in a RESTful service call.
In functional testing we have actively to forget about the implementation details only the interfaces like WebServices count. The input has to be defined and the output for each input has to be known. The thing between both is in Functional Testing a big black box. This are the basics. The setup for a Functional Testing is dependent on the application we have.

7 Performance Testing

Looking at the performance testing it gets a bit more tricky. For getting a real good Performance Testing we have to look at the System in at least two different ways:

1. Looking at specific functionality:

    - When looking at specific functionality like database access in bulk or at archiving large files or many files it is not so hard. We have to set up the system in a way that the factors which disturb the running test are minimised. The advantage is we see how performant this special features are if no other things are kept in mind and no other functionality like background jobs etc. is configured. The disadvantage is that we look at the functionality in some kind of a bubble ;-)
    - Ok now we can set up the system if we look at file archiving with the specific archiving method (SFTP , Local Storage, Cloud Storage....) exactly for this style and method of archiving. We can then use a Testing Tool like e.g. Grafana ... There are many testing tools configurable in a way that the requests can be fired in milliseconds to a WebApplication as example.
    - It is common sense if you look e.g. on archiving to test the same scenario with the same document count and document size with the different archiving methods so that the whole thing gets 1:1 comparable

2. Looking at the System as a complex with many functions run in parallel and may be also on different Nodes / partly in a cloud and so on:

    - If twe look at the system in that way we have to check and look how our system is most commonly used by the customer. In this way then the system is configured. May be with other long running background Jobs or Archive Hooks or other Extensions as for example encrypt data immediately at the point reading it.
    - The system setup is n the best case automised may be via DevOPS.
    - Here it is also I think the best idea if you test different archive drivers to use comparable Sizes of Documents and the same count of documents as well as in this case the file type off the document playing a role if it is not pure I/O.

The first thing necessary for each way of performance testing is to be able to run the application on a dedicated machine or cluster to be sure that no other application which is not in connection with the application we test wastes the resources used for the test. The second thing is to do the tests in a way with different set ups but with a comparable amount and size of data. The other thing is that in both scenarios the decision at which point the measuring takes place is essential this is nearly the greatest issue something potentially can went wrong so that there is nothing (no value) which tells you that the system is really performant.

8 Release Testing

- In the main sense release testing is a kind of testing looking at the complete system the new and old features. The changes / enhancements for configuration and database set up as well as the test if the whole thing can be migrated in acceptable time and in a way the system of the customer is not corrupted - One other thing which has to be tested for each feature is the backward compatibility in each way because no customer will accept it when he has also to migrate all the other elder services he uses since years with success. For release testing the following tests must take place.

8.1 Installation / Integration / Migration Testing

1. Test the installation for the standard way and in the way how the most customers use the system. After that test the installation of each extension which may exist and the other different settings

2. Dependent on the License model you use you also have to test the different configurations of your license model

3. The Integration test on the other side includes also the most common settings in connection to other modules or applications running on the system.

4. The migration test is in the main thing a test which checks if database and configuration migration scripts do what they have to do and if migration is really possible or if something is broken so especially the migration test is part of the release test but it won't be a mistake not to run it the first time with the publication of a new release to avoid such mistakes prior and have more time to fix it if something went wrong.

8.2 Regression Testing

1. The regression test is a test which is made in periods for the running system instances in the best case for each used customer setup. This test ensures that the system runs without problems even though the services and OS versions or VM versions changed.

2. Continuously testing means the incremental testing during development. For this tests the automated way of testing is always the best option because no developer would have the time to test all these things manually. One exception here is the test of things which include the GUI. Up to now I do not know any framework for automated GUI testing which is running without having a great effort in the setup of the tool.

9 An example for our scenario

Think about the following Rest Service:

- We have a service call to create documents
- We have a service call to update documents
- We have a service call to get / query documents
- We have a service call to delete documents

Our Application implements this call in a Apache Tomcat with a database in background and a SFTP archive (Azure, Hitachi, Hadoop, Centera will be also used in real world even today but this is for our example to complex due to the neccessary setup for this storage solutions.

9.1 The Development / Test Environment

- Source Control e.g. BItBucket
- Project Task/Requirements Management e.g. Jira
- Runtime Environment e.g. Docker
- Runner for tests e.g. Jenkins
- Running Docker Remote e.g. Portainer
- JUNIT Styled and Based TestLauncher Proprietary developed Test Runner. Now we can think about how to set up and run the whole thing. NO at first we have to define the Test Cases. Because we need this cases to decide how this test cases can be run . Some cases are really good for automation and other need manual testing

9.2 Test case definition cookbook

To write test cases whith a good coverage of the system requirements is the most difficult thing in doing good testing for functionality performance setup and so on. Now as example we use a functionality:The user likes to store a document / encrypt / sign it and make sure thata legal hold can be set for the document. In addition he needs by the law a retzention time of 10 y for the document in this time the document has to be protected from deletion. During the whole time irt has to be impossible to change the document. What he gets as input is the document in paper form.

We need the complete specification for the functionality: Workflow: Store Business Document Description: The customer likes to save a document which arrives in paper form inn a way that it is signed encrypted and has a retention time of 10 years (type: UN_DELETEABLE READONLY)

1. The document has to be scanned and the scan result has to be put to the incoming document stack storage. To ensure the document is good for long term archiving we use the file format PDF and to be sure the sub-form PDF/A - the /A stands for archive
2. For storing the document the service grabs the document from stack stores into a long term archive system (in our case ***Centera***[6]) and sets the ***ARCHIVED*** flag
3. The document is encrypted and signed in the same step it is read by the application this is done by using special data-streams like they exist e.g. in the IAIK framework or in BouncyCastle
4. Signing parameters are EC (Eliptic Curve) with the strongest usual encryption depth.
5. The encryption is done by CMS (Crypted Message Syntax) Envelop
6. Now to make a change much more difficult the document gets a hash value with strength 512 which is stored in a document hash tree
7. Our retention management that there is a new document with the retention 10 years DO_NOT_DELETE after (10 years + 1 day) ERASE
8. Now after the retention mark is set we set the document capable for a possible legal hold
9. Now all things are committed and the transaction of saving a document is completed

Oh nice we have only nine steps to test what a luck.Oh dear don't be happy to early in each step I see without analysing it step by step about 5 test cases. Starting at data transfer. The different returns a scan can deliver nd the document conversion to PDF/A. Oh we are talking about the first step and we do not have analyzed each parameter which can be wrong yet. How to analyze the steps: Let us take as example step 4 and 5 because testing encryption and. hash values is very funny ;-).

- Step 4: Check the configuration (Hash Tree strength, Algorithm Type , Strength, Provider (e.g. IAIK correctly configured ?) so break it down
    1. call get-crypto-conf
    2. the crypto provider config - check initialisation result code
    3. check if hash-tree generation is turned on
    4. check if signing is on
    5. check if encrypt is on
    6. check the hash tree digest strength
    7. check if signing algorithm is correct
    8. check if the value for the encryption algorithm is ok

[6] ***Centera***: Centera is a hard and software for archiving documents in ***BLOBS*** identified by a key.

- Step 5: Check the outcome and incoming data before and after sign / encrypt:
  1. Check if the document is a real PDF/A
  2. run the Sign/Encrypt view the outcome
  3. decrypt the document look if outcome is a correctly signed document
  4. check the signature via verification e.g. by using DSS-ESIG (Electronic European Standardised Signing and encrypting) as well as signature and document verification.
  5. now check if the document is found in the HashTree check the Hash value and Strength

And now you see two steps and so much to check. Hope I have not forgot something Now you can take this defines steps think about how this steps and actions are triggered in the application and what kind of matcher checks you need to do the test automated.

10 ANOTHER POINT WHY NOT TESTS FIRST Test Driven Development

10.1 Test driven development a short view.

The idea behind Test Driven Development is that of course tests are very important to the quality and the comfort of a software. The method is not very old. In former times there was a running gag in the community of developers: Testing is something for cowards. Nowadays it is clear that the software gets more and more flexible and complex, The fact that most things run in parallel and there it is possible in cloud solutions is another reason to test software very intensive for being sure that the software is running even if there are maybe 40 tasks running in parallel mode.This leads to the conclusion that a software without tests will get useless.Now one had the idea that the interfaces are one of the most important thing which ensures that the design is clear and clean, To get the coders , designers and all the ones who deal with the code in a mode of being forced to develop in a way that at first the interface has to be designed in a way all fits together the idea of writing tests (at first with empty stubs which define the interfaces and see if all things will work interlocked that means the return of the last step is exactly the thing the next step needs and defines. And so the Test Driven Development was born.

10.2 How to realise this thoughts

To realise this thougts we have today test frameworks like **JUNIT** in Java or simply **deftest** in Clojure. Inside the different test methods which we are able to define in **JUNIT** with the **@Test** annotation the logic can be designed and here there are also the asserts if the last call was successful and delivers the correct input for the following task.

10.3 Example of such a test in Java

```
public class ATest extends BaseOfTests {

private Integer derfault = 110;
 /* CTOR*/
 public ATest() {
}

@Test
public aSpecificTest ()
{
```

```
                Integer result = myFirstMethod(this.default); /*here
                   it seems that there is a addition of 86,,, but what
                   the function does really is hidden for us and for
                   sure in the beginning it's empty*/
                String resultStr =  mySecondMethod (result);
                assertThat(resultString is("196"));
}
}
```

And out of these tests stubs for the interface are created

```
public interface MyAppIfc {

        public Integer myFirstMethod (Integer input);

        public String mySecondMethod (Integer toConvert);
}
```

And the class implementing our great highly sophisticated interface ;-) may be look as follows

```
public class MyRealApp implements MyAppIfc {

        @Override
        public Integer myFirstMethod (Integer input) {
                        Integer result = (input + 86);
                        return result;
        }

        @Override
        public String mySecondMethod (Integer input) {
                return Integer.toString(input);
        }
}}
```

And now we can test the thing out of the box for the test is already there.

Chapter 2: Test driven development

11 Short introduction

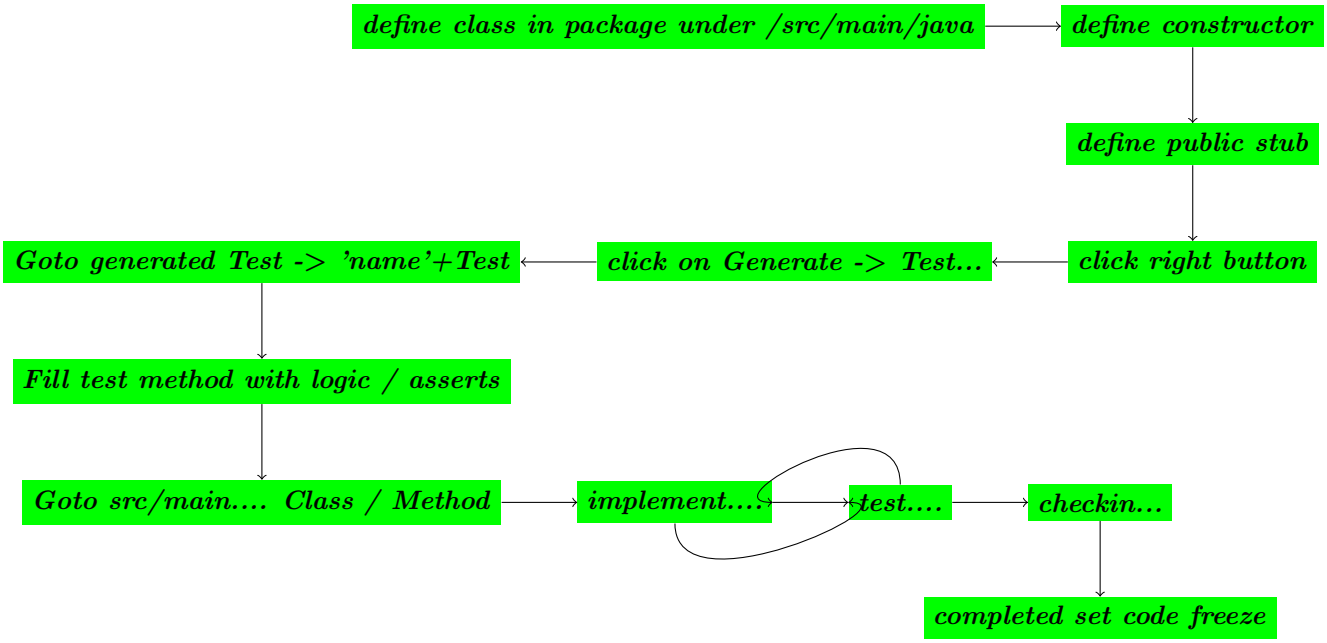
After that we can think about how test driven development can be used to have a well defined interface and to code a application. Test driven development is a kind of development where we define in opposite to the prior example the tests first. Test driven development has the advantage that we see immediately if the function we designed works like it should work because the test shows already up the call and return value of a function how we designed it. Instead of adapting the test to our functionality we write the functionality in the way that we have a test first written according to the requirement.

12 Make the proof of concept

To show how test driven development can be done in a project we make a tiny project called 'Calculator' this calculator is a calculator for mathematical expressions with a GUI. The project is defined in Java with a numeric tower and SWING as graphical interface. Here first the tests are defined and after that the functions are implemented. After done so there are Functional Tests and defined Interactive Tests.

12.1 How to do Test Driven Development - Diagram

Example with Java and IntelliJ / Maven



12.2 How to do Test Driven Development - Step by Step Description

Example with Java and IntelliJ / Maven

1. ***define class in package under /src/main/java*** -> usually we start with any kind of new code in our case here we define a new class.
2. ***define constructor*** -> technical necessary...
3. ***define public stub*** -> this stub defines the interface with parameters and return value.
4. ***click right button*** -> IntelliJ specific...
5. ***click on Generate -> Test...*** Now a new test class is defined in src/test/java/'package' with the name of the class to test and the suffix test.
6. ***Goto generated Test -> 'name'+Test*** -> now we define tests for our stub according to the requirements.
7. ***Fill test method with logic / asserts***-> we have to take care that the asserts and the test logic is clean and proper and reflects the requirements 100%.
8. ***Goto src/main.... Class / Method***-> now we go back to the implementation
9. ***implement....*** -> implement and go to test.
10. ***test....*** -> test the implementation and if necessary refine the tests. If all is ok go to the next step if not go back.
11. ***checkin....*** -> in this step we checkin the code and the tests. The checkin must be at least compilable.
12. ***completed set code freeze*** -> now we declare the task as ready but if a code review shows errors or unclean code it may be re-opened.

12.3 See how this works in our example with the calculator

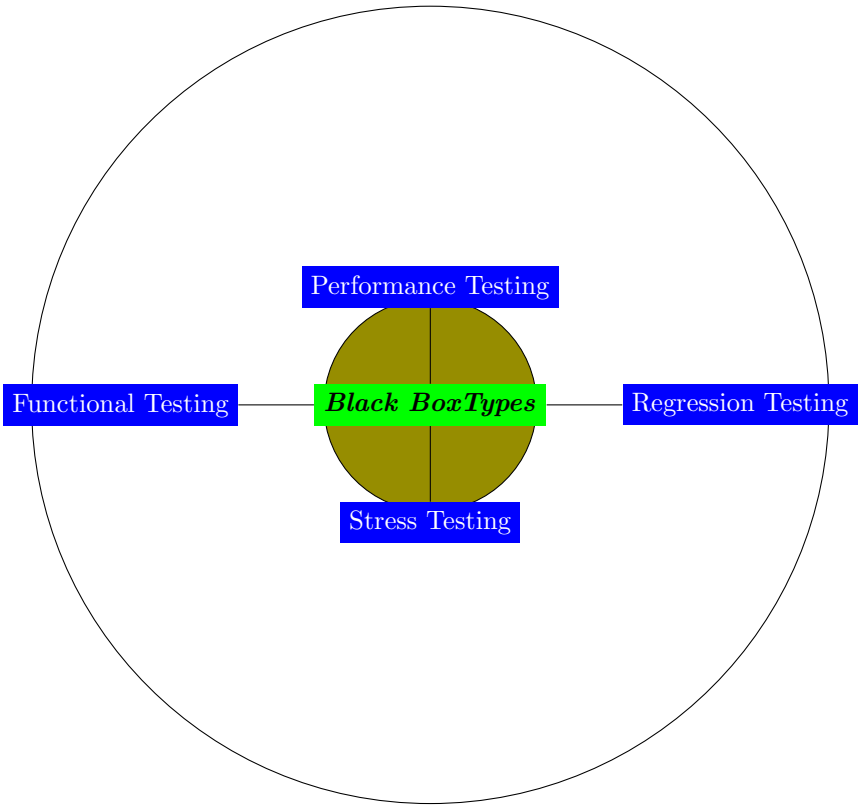
Chapter 3: The test types in detail

For the test types you will see there are different ways of categorisation. I will show them without telling the one or the other way is better, because in each discipline of data and computer science there are always different ways of thinking. Here is the categorisation we had in my former company: ***Functional Testing*** checks if the entire system works as intended, while ***Unit Testing*** ensures individual code components function correctly. Scope: Functional Testing evaluates entire workflows and features, whereas Unit Testing isolates and tests specific functions or code segments.
REMARK:*For the other categorising and describing the alternate test types I used also input from* Solutions Hub – see web link(1.)

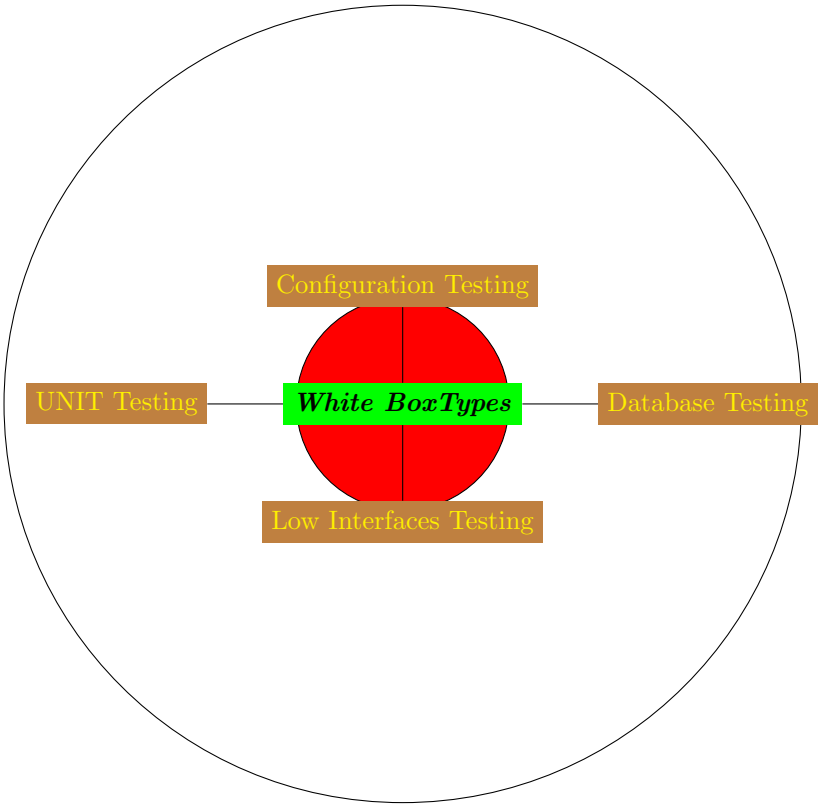
13 Test types in overview often used

Here at first the part how many companies define the different test:
There is a separation between the tests in a way that the test types are divided by defining UNIT Testing which is a kind of White Box test designed and executed by the developer. For very important UNIT tests some Jenkins-Test Jobs can be defined. The other Tests are Black Box Tests as Functional Tests, Performance Tests, Stress Tests and so on. For these tests it is good to define Jenkins-Test-Jobs with these defined Jobs also Regression and RE-Testing can be done.

13.1 Black Box testing type



13.2 White box testing type



14 Test types description

1. ***Functional / Performance / Stress testing as example for definition*** - These test types are defined for checking the correct functionality and to see if the application runs functionally correct, stable and with sufficient performance. Normally these tests are defined with the aspect to ignore the internal way how the application is developed.

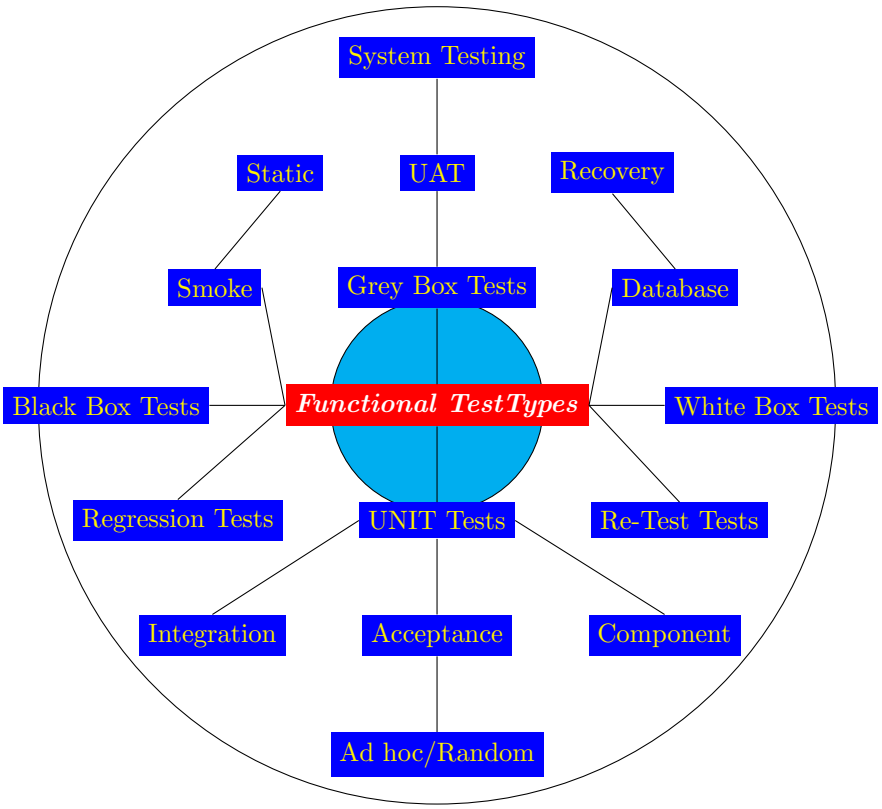
2. ***UNIT testing definition*** - These tests are designed to check the functionality of atomic functions inside the project. With full knowledge how the functions work internal.

3. ***Interactive testing definition*** - Interactive testing is the type of testing where we have a detailed test definition without the possibility to automation. The user or a tester has to follow the steps. and write a protocol about the results. Best place for that is a tool like JIRA

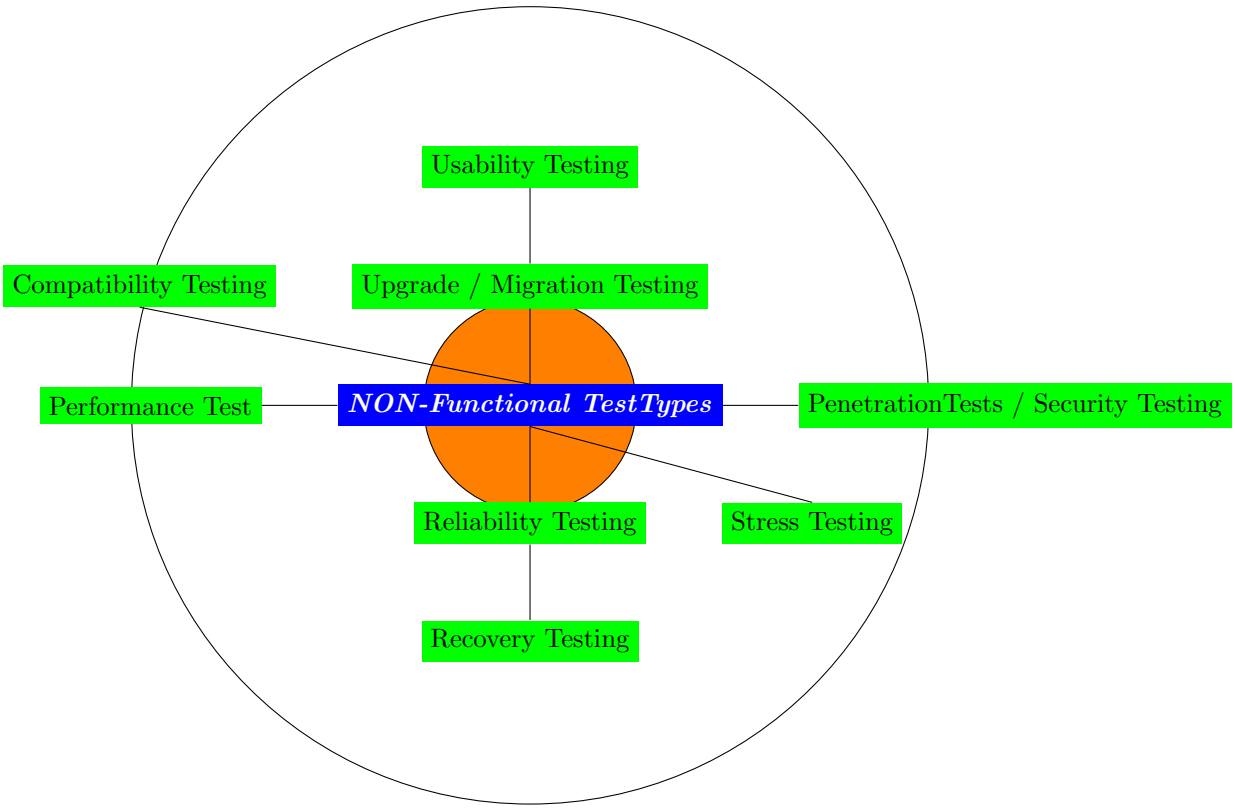
4. ***Security testing definition*** - The security tests check the secure data transfer normally according to eIDAS in Europe or e.g.: NIST in USA. And there has to be also a test if changes to the system are tracked correctly so that there is no way to falsify something.

15 Alternate way to define and categorise test types in overview

15.1 Functional test-types overview (ALTERNATE)



15.2 NON-Functional test-types overview (ALTERNATE)


Usability Testing
Compatibility Testing
Upgrade / Migration Testing
Performance Test
NON-Functional TestTypes
PenetrationTests / Security Testing
Reliability Testing
Stress Testing
Recovery Testing

16 Test types description

16.1 Functional Tests

Functional tests have the focus on testing the functionality from outside for example calling services of an application in an application server. These tests MUST not know any implementation details but only in the case of the example the REST Service Interface or the Interface of whatever to test. The functional Tests are best placed in a functional tests project separated from the project where the development takes place. So they are clean separated and there is also not the danger for the one who develops them to have a deep look at the code.

1. ***UNIT Tests:*** The focus of UNIT Tests is on the testing of the code units functions, methods of a class or a functional language in object oriented languages where we have private and public e have to look that we only test the public entries directly and the private entries as far as possible only indirect so that we do not break the language security mechanism of private access. The UNIT tests are written based on the knowledge of implementation details.

2. ***Use Case Tests:*** The Use case tests are the next abstraction level. The focus is on testing the things in that units the use cases are defined in the projects specification. For example workflow for a banking transaction. These tests are sometimes automated and some projects need to do interactive / UI tests for that

3. ***FeatureTests:*** Features are new functionalities in a product or project the features can be a technical change with no interface change but they can also change the interfaces. This is the reason to test features in different ways. The one possibility only changes UNIT tests the other also changes for example rest services or the UI. Performance and regression tests have to be performed in both scenarios.A feature is part of a minor or major release.

4. ***Black Box Tests:*** Black Box Tests are tests of the functionality where the Tester who develops them only knows the interface, no matter if the interface is a library functionality a REST service or a UI interface

5. ***Grey Box Tests:*** Grey Box Tests are between Black Box Tests. The Tester knows some details about the implementation. For example he knows how the entry parameters are checked or he knows details about the structure used (module tree) for implementing a requirement.

6. ***White Box Tests:*** White Box Tests are tests where the Tester knows all the details about the implementation down to the last if condition or while loop....

7. ***Static Tests:*** Static tests are a software testing practice that examines code and documentation without executing the program to find defects early in the development cycle. This is done by reviewing work products like source code, design documents, and requirements for inconsistencies, typos, security vulnerabilities, and deviations from standards. Static testing helps save time and money by catching and fixing errors before they escalate, improving code quality and maintainability.

    - ***What is tested:*** Static testing can be applied to various work products, including requirements, design documents, and source code..
    - ***What is found:*** It can identify defects such as: Syntax errors, dead code, and unused variables Logic flaws and incorrect data/control flows Security vulnerabilities like buffer overflows Ambiguities, contradictions, and omissions in requirements How it's performed: Static testing can be done manually through reviews and walkthroughs or automatically using tools called static analyzers.

8. ***System Tests:*** A system test is a phase in the software development process where the complete, integrated software is tested to ensure it meets all specified requirements. It evaluates the system

as a whole, focusing on how all the individual components work together, rather than testing them in isolation. This type of testing is typically conducted after unit and integration testing, and is often done using black-box testing methods to simulate a user's perspective.

9. ***Component Tests:***

10. ***UAT Tests:*** User Acceptance Tests are performed by the Users. The Users check how good a UI and how it is implemented fits their needss and if the look and feel is good. Since the projects and products are for the user these tests count as functional testing type

11. ***Re-Tests:*** Re-Tests are tests which are done if something was developed and tested .Â Imagine the tests shown up three errors so the tests checking these three conditions are re-executed after some fixes in the code

12. ***Database Tests:*** Database tests are tests for Database Schemas and database data structures e.g. in NoSQL, SQL databases. These data organisations are tested with the view on how good they support the functionality and the enhancements for the functionality..

13. ***Recovery Tests:*** Tests for a functional correct data recovery / failover tests are down there in the NON - Functional area.

14. ***Integration Tests:*** Integration tests are done to. see if the product or project is able to be integrated into the target system and its applications and infrastructure.

15. ***Smoke Tests:*** Tests for looking if one of the 'OK' paths is working for a functionality. For getting more complete also Exception tests are needed.

16. ***Acceptance Tests:*** In counter to the UAT these tests check the acceptance of the project or product against the specification of the functionality and if in each case the shortest way to do a task is realized

17. ***Ad Hoc / Random Tests:*** These tests are like the name tells us tests which are not in the usual testing scheme. Sometimes we have scenarios where the test by scheme is not sufficient so it is good to test things which are really unexpected e.g. Crazy inputs like prime numbers for a buffer size

18. ***Regression Tests:*** Regression tests are focused on checking if the functionality of the application works inn the same way after introducing features fixing bugs making changes (add ons) in the configuration and so on.

16.2 NON-Functonal Tests

1. ***Performance Testing:*** The focus of performance tests s on how fast a high amount of data or requests can be processed. To execute performance tests a realistic setup is necessary because if the tests run in a 'bubble' there is no way to distinct how performant the application really works in an environment like it is found at thee customers environment. In the best case all otherÂ applications driven by the customer run in parallel as it is in reality.

2. ***PenetrationTesting / Security Tests - Mostly Service Tests:*** These tests are focused on checking the stability of the project in the way that requests to the software are placed in a way that the application can crash if it is not developed in a clean secure way. One of the possibilities is to send one or more requests which seem to be valid but which attack the application with incorrect data. Another possibility is that we send a mass of requests like in a DDOS Attack. There is no restriction in thinking about ways to put the whole thing out of order. Sometimes you have to think like the evil to get it secure.

3. ***Upgrade / Migration Testing:*** Some changes have an effect on external programs and data definitions like in a DBMS. These changes have to be upgraded automatically in best case. Here we have to decide if the Upgrade has to be possible from one release back or more releases back. What also needs a upgrade tests with a view on backward compatibility are Configurations either in the database e.g.: as CLOBS or XML, Properties or INI files

4. ***Reliability Testing:*** Checks if the system can perform consistently without failure over a specified period.

5. ***Usability Testing:*** Assesses how easy, efficient, and satisfying the system is for users to operate.They are defined in a way that they state if it is easily possible to use the functionality. Usability tests are in the best case not only done for the UI but also for the interfaces used by developers and administrators in this case we see the configuration strictly as part of the interface.

6. ***Recovery Testing:*** Verifies the system's ability to recover from crashes or failures.

7. ***Compatibility Testing:*** Verifies that the software works across different environments, browsers, or devices.

8. ***Stress Testing:*** Determines the breaking point of the system by overloading it beyond its normal capacity.

16.3 Resume

In general we can say that all these things have to be tested in different ways and that there is more than one possibility to test all these things. Some of the test scenarios are destined to be tested in a way that most of the parts are automated. This automation can be defined from the time point of check in to the point building a tested delivery. In the following chapters and sections we like to describe a way of automation which covers nearly all parts which has to be tested before delivery.

Chapter 4: Deep dive into test automation

17 Look at UNIT testing first

The automation of Tests can take place on different levels of Test execution. At first take a look on test automation of ***UNIT Tests***.[7] The UNIT tests should be written by the developers coding the functions. There are two possibilities. Either the developer coding the function writes the UNIT tests during coding or the UNIT tests are written by the developer who does the code review which I think is very useful. I would prefer to do both things. First coding the function or write the UNIT test and code the function that it fits. After both things are written a code review should take place. Now the reviewer sees if there is something wrong in the code and / or if UNIT tests are missing and he can write further tests. In best case he writes tests which state if the errors he found are fixed. In most cases UNIT tests are per definition White Box Tests. Here again our sort example. A typical simple UNIT test will be in Java with JUNIT:

```
@Test
 public void testQuickSortSimple() {
      Integer[] arrayToSort = newArray();
      Integer[] arrayToSortComp = newArray();
      Arrays.<Integer>parallelSort(arrayToSortComp, new
         Comparator<Integer>() {
```

[7] UNIT Tests are normally placed in the same project where the development takes place. The Tests are done on the programming language function level testig functions and / or classes and the implementation from the complete technical point of view

```
            @Override
            public int compare(Integer o1, Integer o2) {
                return o1.compareTo(o2);
            }
        });
        QBubbleMerge.quickSort(arrayToSort, ASC, 0, LAST_INDEX);
        assertThat(Arrays.compare(arrayToSort, arrayToSortComp),
           is(0));
        arrayToSort = newArray();
        arrayToSortComp = newArray();
        Arrays.<Integer>parallelSort(arrayToSortComp, new
           Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2.compareTo(o1);
            }
        });
        QBubbleMerge.quickSort(arrayToSort, DESC, 0, LAST_INDEX);
        assertThat(Arrays.compare(arrayToSort, arrayToSortComp),
           is(0));
    }
```

And in Python with the package unittest which is very similar:

```
import unittest
import copy
from sort.algorithms.QBubbleMergeSort import QBubbleMergeSort
from sort.algorithms.SortData import SORT_DATA
from sort.algorithms.SortData import LAST_INDEX

class QBubbleMergeSortTest(unittest.TestCase):
    def test_quicksort_simple(self):
        toSortAsc = copy.deepcopy(SORT_DATA)
        toSortAsc.sort()
        toSortAscQuick = copy.deepcopy(SORT_DATA)
        QBubbleMergeSort.quickSort(toSortAscQuick, 0, LAST_INDEX)
        self.assertEqual(set(toSortAsc), set(toSortAscQuick))
```

18 Functional Tests development in parallel

Functional Tests are tests not only of one function in the technical sense but are tests to check if for example a defined REST request works in a proper way. The best way to define this tests is to have a separate test project. The functional tests are per definition black box tests. For the functional tests the technical definition of a REST request or client action ore something else is necessary. These tests are a abstraction level between UNIT and Use Case Tests.

19 Tests for Use Cases

The tests for Use Cases is a test for a system / project / product requirement to check if the requirements defined by the customer are all implemented in a proper way. One example will be a store of a document with signature like described above. To check if this works more than one functional tests may be executed in a transaction bracket.

20 Setup / Build automated test environment

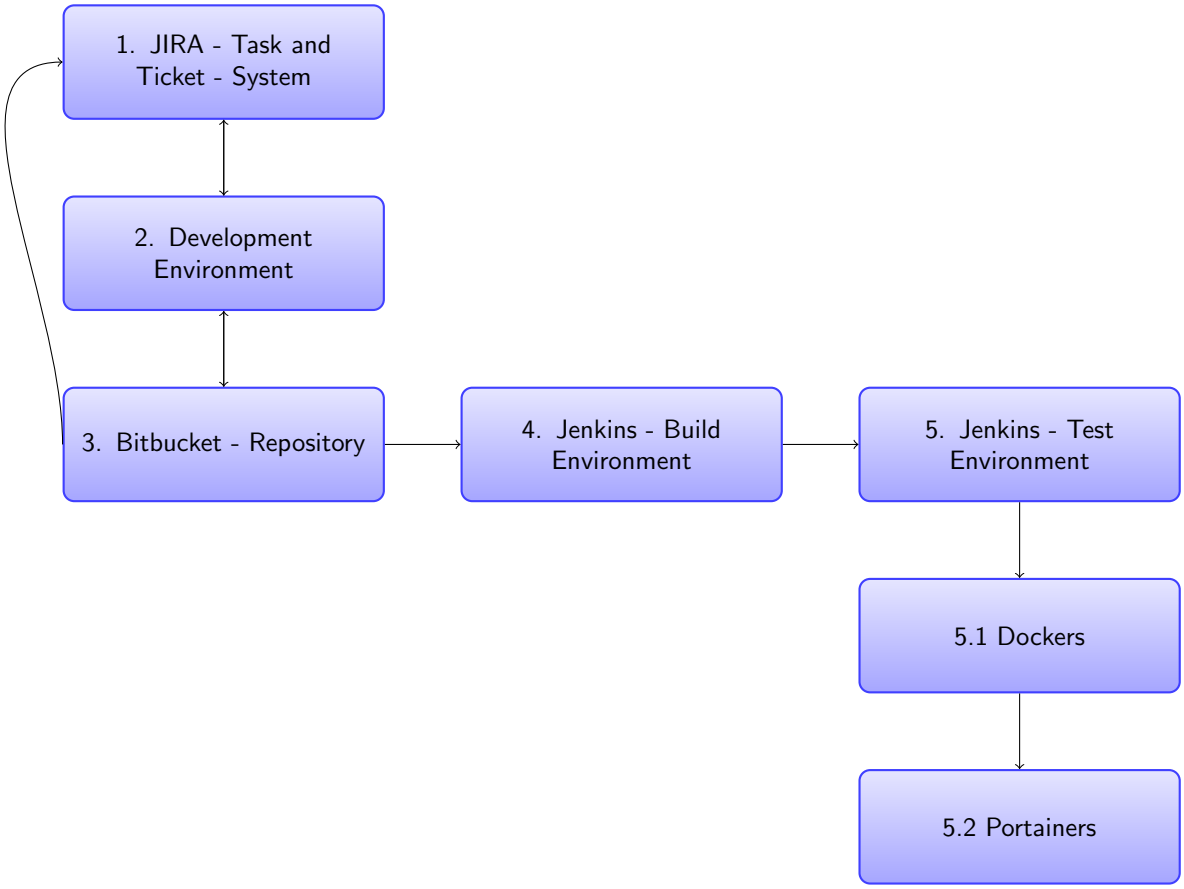
To get automated tests in a structured designed way we have at first to take a look on which tools we need and use for getting the test process chained. Here is an example of tools and build automated tests with them this starts with the right ticket and task documentation tool. In our case we use JIRA.

20.1 Example with toolchain

A toolchain is a set of software development tools used to build and otherwise develop software. Often, the tools are executed sequentially and form a pipeline such that the output of one tool is the input for the next. Sometimes the term is used for a set of related tools that are not necessarily executed sequentially. (Wikipedia)
Our toolchain is a chain of tools we use for defining use cases and requirements which are broken down to tasks and sub-tasks for development. The development is done in an development environment like IntelliJ or Eclipse IDE and after being ready with a step it is committed to a Versioning Control System (VCS) like Bitbucket / GitHub (GIT). If the check in is done a Jenkins Build Job is triggered which builds the project / sub project. After this build the Jenkins in which the tests run is triggered so that the test run follows automatically. After that a deployment can be build. Either submitting the build in the Build Jenkins or we can use the test run afterwork to trigger the build of an assembly.
The tools we use :

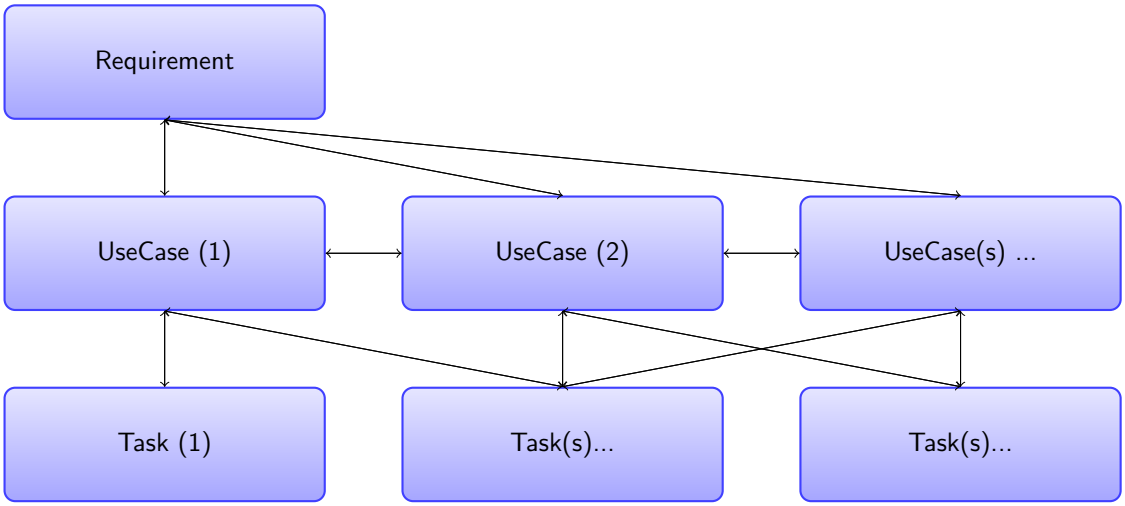
1. JIRA: This tool is used to define and document the requirements, use cases, tasks , epics, tests, bugs and so on. It is the tool to define the project as well as the ticket system. Also teams and sprints can be defined in JIRA
2. Development Environment: E.G. IntelliJ... Tool for developing the project (Coding).
3. Versioning Control System: E.G. Bitbucket. Tool for versioning code and set up branches and releases – the control system can be configured with JIRA so that check ins are tracked by task in JIRA
4. Build / Tests Tool: A tool like Jenkins can be used to set up build processes and test processes
5. Docker Containers can be used to set up systems and services for testing. Also the test run itself can be started in a separate docker

OVERVIEW - TOOLCHAIN



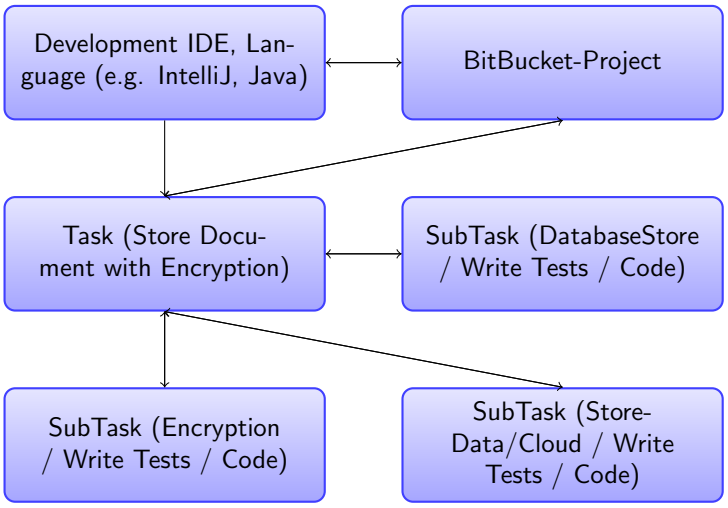
Here we have a short overview on the whole tool-chain. Later after showing the different parts we will look how a typical workflow from Requirement to the Deploy will look like.

1. JIRA-Task and Ticket-System



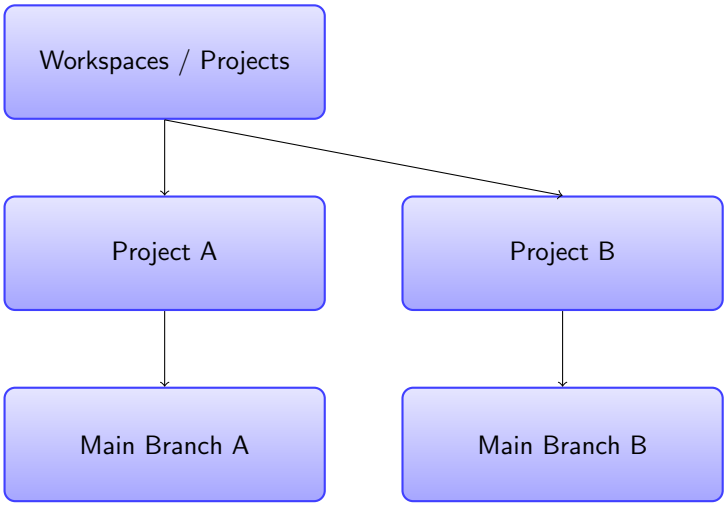
We see here that in the end we have nearly something like a spider-net between the tasks and use cases. The reason is that there are many tasks which are the base for tasks which are specialised tasks for the use case xyz.

2. Development Environment



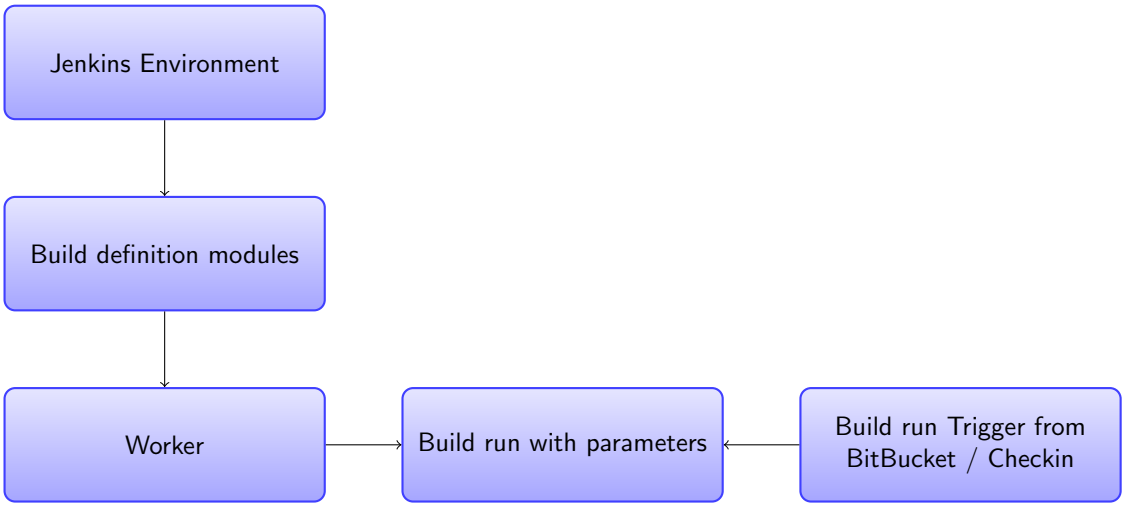
A typical workflow in the development environment will be that the developer gets a task and hast to break down the tasks in classes functions and configurations. The developer will choose one of the development tools which are allowed by the standard of the company or department. In this development environment the code is developed in a sandbox and later on the developer commits the code step by step to the version control system (e.g.: GIT) and then the development of the next function is done. A development environment should be choosed im a way that not only the pure coding is possible and that it supports some goodies e.g. Test Driven Development. In JIRA for example the developer can comment the task if necessary and he sees the output (what checked-in in GIT) if he uses the feature of JIRA and BitBucket that the commits are grouped by task or defect. Normally a development Tool like IntelliJ supports all these things.

3. Bitbucket-Repository



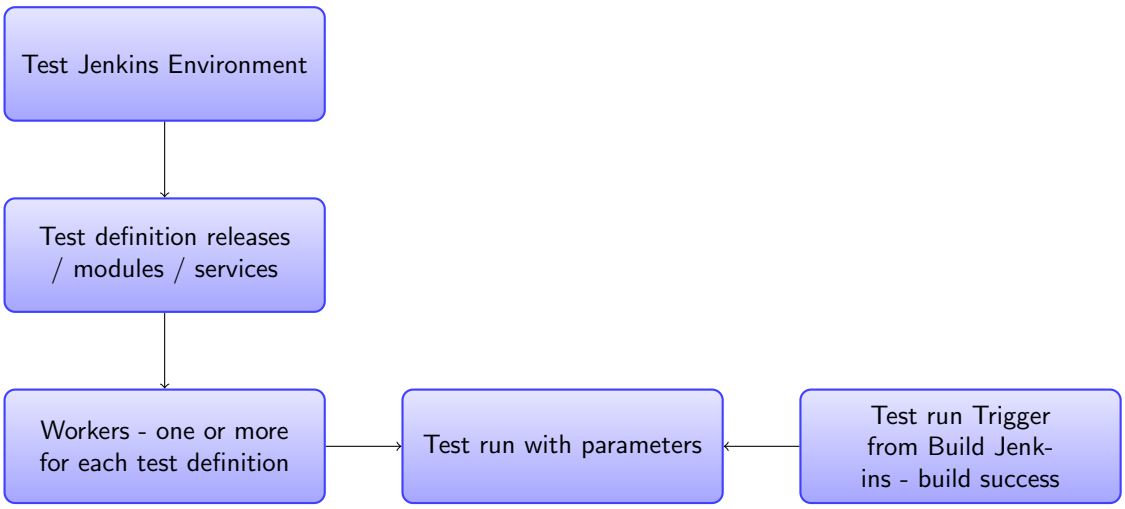
The Bitbucket System holds GIT repositories. Each repository is a project or a sub-project which also may be dependent from one to the other. The projects can have a main Branch or several other branches which hold a specific step of development which is later on merged to the main branch. It is usual to make a code freeze for a release and after that run the major or minor release tests.

4. Jenkins-Build Environment



The build environment is defined in a Jenkins instance. For each module or project is at least one build for the trunk defined and a release build. The build is triggered by a check in to BitBucket. After the build is successful the Tests in another Jenkins are triggered if defined.

5. Jenkins-Test Environment



In the test part tests for many different projects or parts of projects are defined as test run. These test run definitions have 1..n workers assigned in these workers docker environments are established to run in Portainers. One of these docker environments is always the test run itself. The other belong to the application or service to be tested.

20.2 The workflow in the tool-chain

Chapter 5: Clean coding and project design as part of quality to make testing ea

21 Rules for a clean project with proper testability

One important thing to get a clean logic and a good quality is the design o the project in a way that the different parts/components can be tested separately as well as in a way that no mocking inside the project is necessary. Here some design patterns and rules as well as coding rules are very useful. First we like to take a look at the project design.

21.1 How to design a application for getting a clean testable project - in overview

There is more than one pattern to get an application clean with proper logic and a good testability:
The first pattern we will show is the ***MVC*** Model View Controler. [8] This pattern is very useful for standalone applications like a "big fat client" or something like Thunderbird (A mail program).Here a example: Model View Controller app example
For services / cloud computing the common sense is to think in a way which is a bit different from MVC. Let us have a look:
Here we define separate layers in the application. In the following list represented from highest to lowest ->

1. ***Business Logic:*** In the business logic the implementation of the business part takes place. The business part ids the part with the logic according to the requirements and the view of the user. We avoid to implement backend things which do not belong to this logic because they are the technical realisation depending on the used systems.

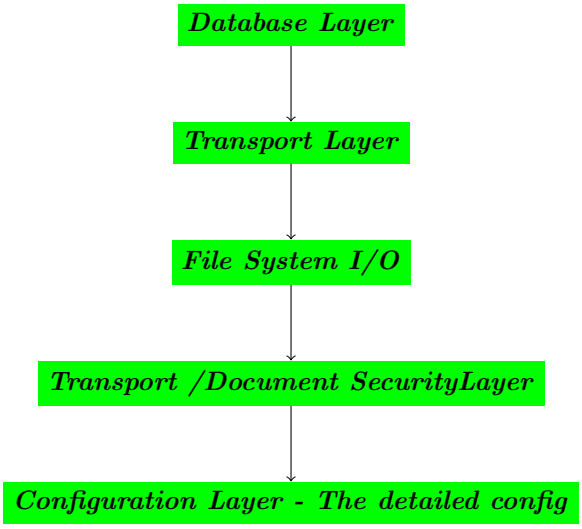
2. ***Backend:*** Here in the technical part we have the view on the technical details dependent from the used products / systems and the internal data model. This part contains e.g. also select statements to access a specific database like MySQL or DB2 or MS Sql-Server.

3. ***Separation:*** The separation between the two parts is done through develop and define a API in form of interfaces which are implemented with several different backend 'processes' and so they decouple the one layer from the other completely.

Let us have a closer look on the structure of the backend so that we have a backend layer which allows clean testing.

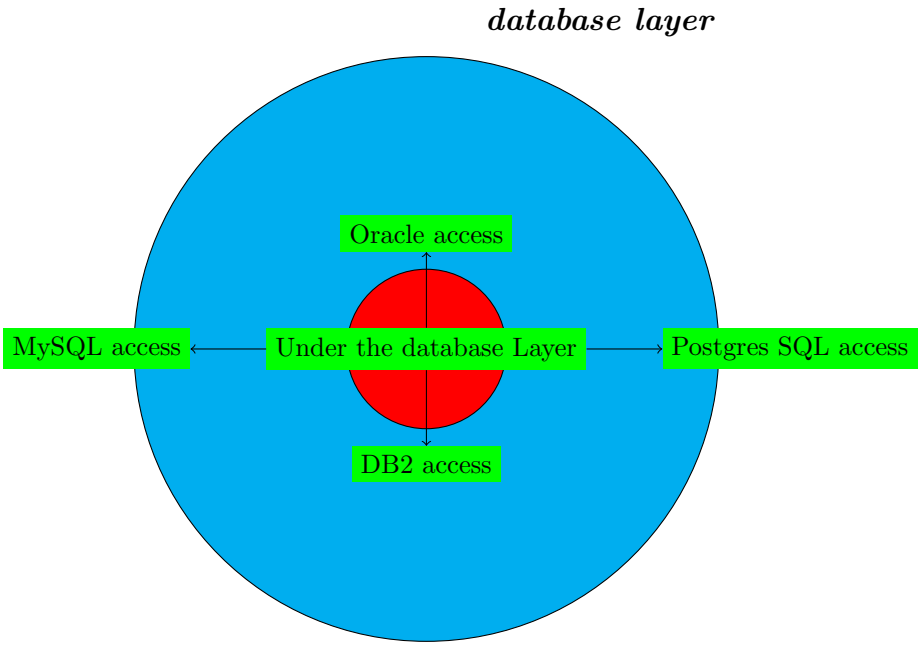
[8] ***MVC:*** ***M***odel ***V***iew ***C***ontroler is a model to separate the view (Maybe GUI) from the model which is the data model. And then use the controler to define the logic manipulating the data and build workflows for example

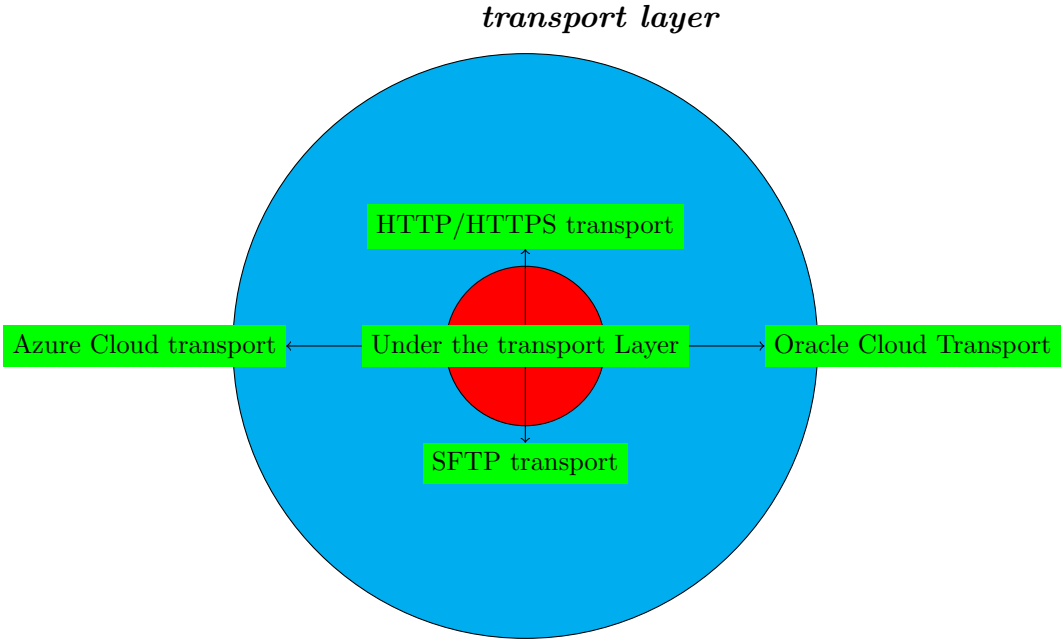
The backend itself:



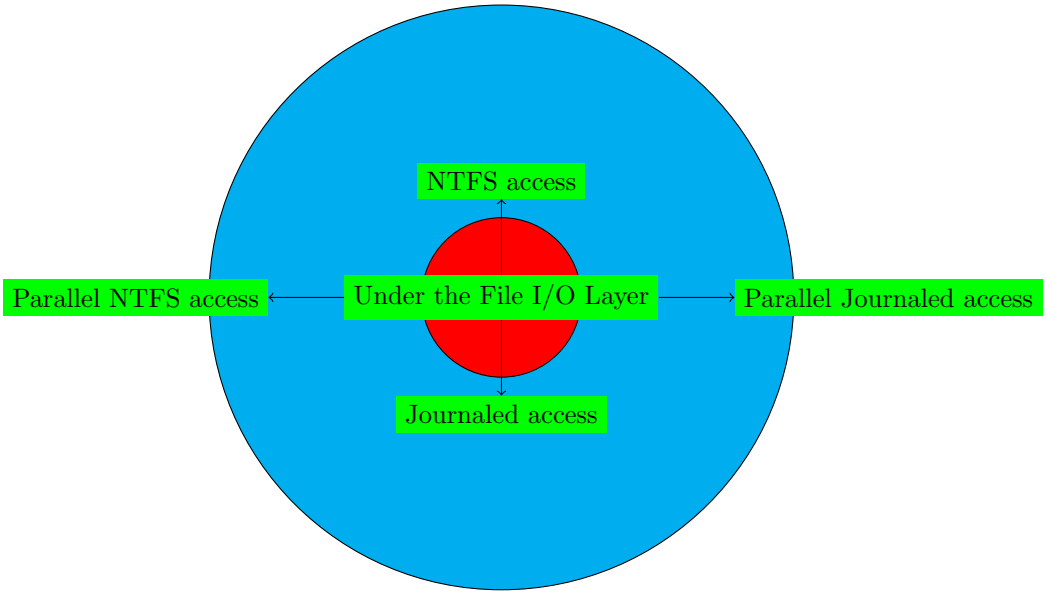
- ***Database Layer:*** The database layer is a layer in which the general access to SQL / NoSQL (e.g.) databases is defined. Here the definition which is valid for all databases of a specific kind is done.
- ***Transport Layer :*** In this layer the definition of the transport of data over different protocols takes place. Not only the mentioned but also cloud connectors and archive backends as a few examples can be implemented.
- ***File System I/O:*** Here the I/O functionality to access file systems (directories, files) takes place.
- ***Transport /Document SecurityLayer:*** In this layer the logic for a secure transport of data takes place as well as the logic to protect documents with hashes evidence records and signatures.
- ***Configuration Layer - The detailed config:*** The config is stored in each project in a defined structure for example XML-BLOBS here in this layer the technical details how to create, update, read and delete the configuration or parts of it is defined and implemented.

The interface to the backend and interfaces in the backend:

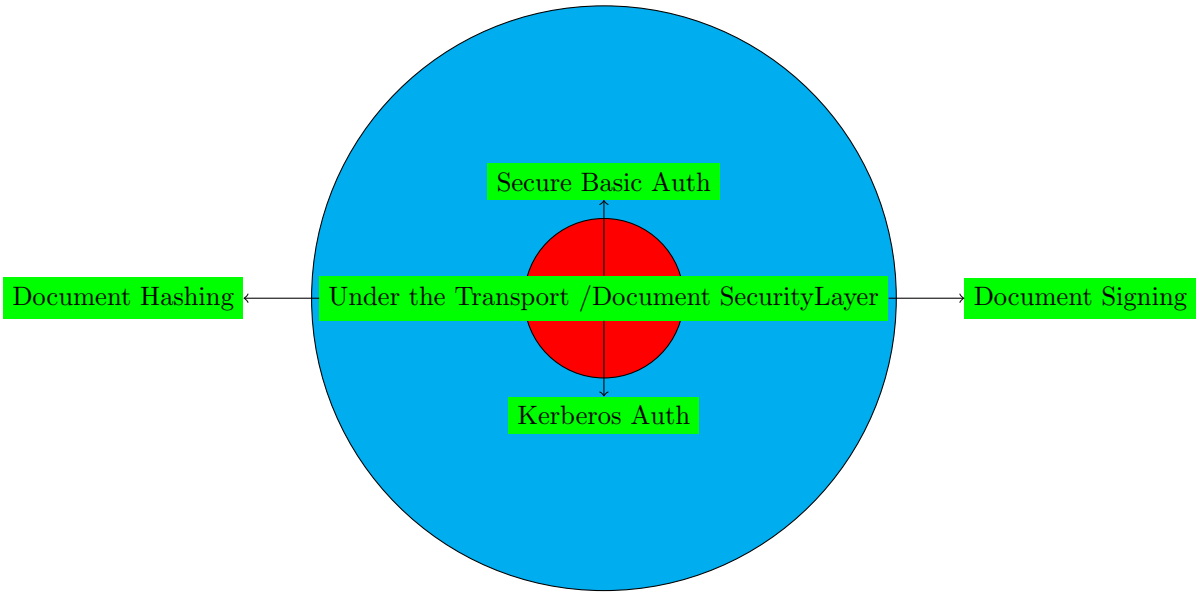




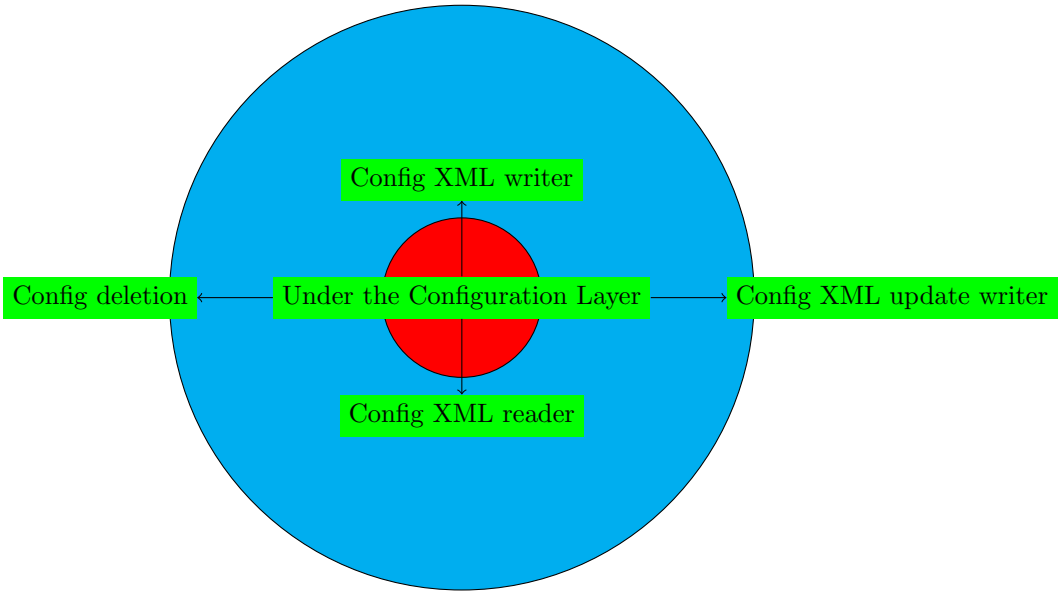
File system I/O layer



Transport /Document SecurityLayer - Following eIDAS



Configuration Layer - The detailed config



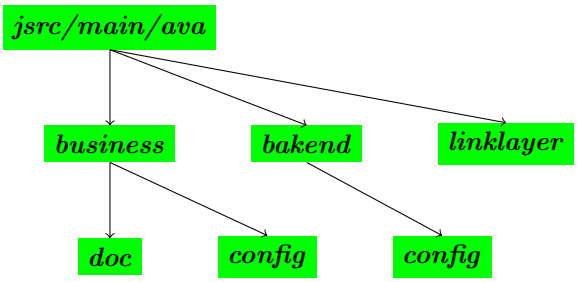
The shown things are an example for what can be defined in a backend of a service or an application. The example does not show the complete possibilities and not all products which can be used there.

21.2 How to design a application for getting a clean testable project - in detail (description)

Now we want to have a deeper look on the components inside the backend and how they are connected to the business logic so that the one who codes the business logic does not have to think about the technical details. At first have a look on the way where the logic is separated in business logic and backend logic which are then connected.

let us have a look at the structure and the code to discuss it.

The high level project structure



The code stubs for configuration to see the intention

The code of the interface

```
package io.github.hglabplh_tech.linklayer;

import io.github.hglabplh_tech.DOCTypes;

/**
 * This interface holds the public functions for the configuration
    access
  * @author Harald Glab-Plhak (C) 2025
 *
 */
public interface ConfigurationIfc {   /**
     * The HTTP / HTTPS port
     * @return the port number as integer
     */
    int getPort();

    /**
     * The database connect string
     * @return the database connect parameters as string
     */
    String getDBConnectString();

    /**
     * The default type for import / export / create
     * @return the default type enum
     */
    DOCTypes getDefaultType();
}
```

The code of the config application class

```
package io.github.hglabplh_tech.business.config;

import io.github.hglabplh_tech.DOCTypes;
import io.github.hglabplh_tech.linklayer.ConfigurationIfc;

/**
 * This class holds the configuration settings for our system in
    business view
 * @author Harald Glab-Plhak (C) 2025
 *
 */
public class AppConfig implements ConfigurationIfc {

    /** HTTP port number*/
    private final int port;

    /**the database connection string*/
```

```
    private final String dbConnectString;

    /**the default document type when storing documents*/
    private final DOCTypes defaultDocType;

    public AppConfig(int port, String dbConnectString, DOCTypes
       defaultDocType) {
        this.port = port;
        this.dbConnectString = dbConnectString;
        this.defaultDocType = defaultDocType;
    }



    @Override
    public int getPort() {
        return this.port;
    }

    @Override
    public String getDBConnectString() {
        return this.dbConnectString;
    }

    @Override
    public DOCTypes getDefaultType() {
        return this.defaultDocType;
    }
}
```

The code of the config backend class

```
package io.github.hglabplh_tech.backend.access;

import io.github.hglabplh_tech.DOCTypes;
import io.github.hglabplh_tech.linklayer.ConfigurationIfc;

/**
 * This class is between the real configuration definition and the
    access to the config resource
 */
public class Configuration implements ConfigurationIfc,
   AccessCtxAndConnIfc {
    /** HTTP port number*/
    private final int port;

    /**the database connection string*/
    private final String dbConnectString;

    /**the default document type when storing documents*/
    private final DOCTypes defaultDocType;
```

```
    public Configuration(int port, String dbConnectString, DOCTypes
        defaultDocType) {
        this.port = port;
        this.dbConnectString = dbConnectString;
        this.defaultDocType = defaultDocType;
    }

    @Override
    public int getPort() {
        return this.port;
    }

    @Override
    public String getDBConnectString() {
        return this.dbConnectString;
    }

    @Override
    public DOCTypes getDefaultType() {
        return this.defaultDocType;
    }

}
```

Generated config XML(JAXB class)

```
package io.github.hglabplh_tech.config;

import java.math.BigInteger;
import jakarta.xml.bind.annotation.XmlAccessType;
import jakarta.xml.bind.annotation.XmlAccessorType;
import jakarta.xml.bind.annotation.XmlElement;
import jakarta.xml.bind.annotation.XmlRootElement;
import jakarta.xml.bind.annotation.XmlType;


/**
 * <p>Java class for anonymous complex type.
 *
 * <p>The following schema fragment specifies the expected content
    contained within this class.
 *
 * <pre>
 * &lt;complexType&gt;
 *   &lt;complexContent&gt;
 *     &lt;restriction
    base="{http://www.w3.org/2001/XMLSchema}anyType"&gt;
 *       &lt;sequence&gt;
 *         &lt;element name="port"
    type="{http://www.w3.org/2001/XMLSchema}integer"/&gt;
```

```
 *         &lt;element name="dbConnectString"
    type="{http://www.w3.org/2001/XMLSchema}string"/&gt;
 *         &lt;element name="defaultDocType"
    type="{http://www.w3.org/2001/XMLSchema}string"/&gt;
 *       &lt;/sequence&gt;
 *     &lt;/restriction&gt;
 *   &lt;/complexContent&gt;
 * &lt;/complexType&gt;
 * </pre>
 *
 *
 */
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "port",
    "dbConnectString",
    "defaultDocType"
})
@XmlRootElement(name = "appconfig")
public class Appconfig {

    protected int port;
    @XmlElement(required = true)
    protected String dbConnectString;
    @XmlElement(required = true)
    protected String defaultDocType;

    /**
     * Gets the value of the port property.
     *
     */
    public int getPort() {
        return port;
    }

    /**
     * Sets the value of the port property.
     *
     */
    public void setPort(int value) {
        this.port = value;
    }

    /**
     * Gets the value of the dbConnectString property.
     *
     * @return
     *     possible object is
     *     {@link String }
     *
     */
    public String getDbConnectString() {
        return dbConnectString;
```

```
    }

    /**
     * Sets the value of the dbConnectString property.
     *
     * @param value
     *     allowed object is
     *     {@link String }
     *
     */
    public void setDbConnectString(String value) {
        this.dbConnectString = value;
    }

    /**
     * Gets the value of the defaultDocType property.
     *
     * @return
     *     possible object is
     *     {@link String }
     *
     */
    public String getDefaultDocType() {
        return defaultDocType;
    }

    /**
     * Sets the value of the defaultDocType property.
     *
     * @param value
     *     allowed object is
     *     {@link String }
     *
     */
    public void setDefaultDocType(String value) {
        this.defaultDocType = value;
    }

}
```

The reasons of such a separation

TheÂ reason for making this separation is that the tests for separated components (separated by functionality) makes the different components better testable because the tests can be keeped simple in the way that we do not have to know in a test for a business object how the archive connector works and on the other side the archive connector is independent from the logic above.In this example weÂ see that also Mocking Methods can be avoid better. It is a target to avoid mocking as far as ever possible although many people in our world tell we really need much mocking. Mocking a function is only ONE of MANY ways to solve a dependency problem and it does not make it really every time easier. I will take a deeper look on it.

The other important point is that the use of a new database or service is easier because there is only the last step which has to be implemented for that way with base functions which are defined in an

interface. This way reduces errors and makes a great part of the software system independent. The advantage is that if you have a new Storage or Transmission with system dependencies you have only to test the implementation of the transport or storage interface and it is nearly all times the case that the rest of the product works well with it in cause of the strict separation via using a interface. Examples for this way are in Java the Collection and Stream and especially the InputStream / OutputStream and Reader / Writer interfaces.

21.3 More about Mocking and 'Fake' Libraries / Services

Mocking with Mockito or other mocking frameworks

Mockito is a mocking Framework for Java normally used together with JUNIT. The mocks of Mockito give us several possibilities to say how the mocked functions have to react dependent on the way they are called.One of the other possibilities is to count the method calls to specific methods of the mocked object. This sounds good for many developers because there is a way to get rid of function calls to resources or other external things which need a long time and often you hear the argument that in case of mocking only the function logic is tested and not the logic of external components. But this is only one side of the medal. On the other hand the mock logic is dependent from the internal implementation of the function that means all times the function changes significant the places where it is mocked have also to be changed.

One other argument against mocking is that the system behaviour is changed if functions are mocked. Not only in case of calling a mock function but also in case of counting calls to functions etc. the behaviour is changed "You change the system functionality and behaviour even by observation". Here an example for method counting and its pitfalls:

And here one thing that does not work without rewriting so that it can be mocked

The app code

```
public File getTempFile() {
        return this.tempFile;
    }


    public List<String> getContent() {
        return Collections.unmodifiableList(this.content);
    }
      public List<String> readTempTestTextfile() {
        return this.readTestTextFileCorrectFMock(this.getTempFile());
    }

    public boolean writeTempTestTextfile() {
        return this.writeTestTextFile(this.getTempFile(),
           this.getContent());
    }
```

The test code

```
@BeforeEach
    public  void beforeEach() {
        mockedVersion = Mockito.mock(MockedSample.class,
           Mockito.CALLS_REAL_METHODS);
    }
  @Test
    public void writeTestTextFile() throws IOException{
        boolean ok = mockedVersion.writeTempTestTextfile();
        List<String> result =  mockedVersion.readTempTestTextfile();
        result.forEach(e -> System.out.println(e));
        int countRead = result.size() + 1; // now we are all times ok
           with it
        verify(mockedVersion, times(countRead)).readOneLine(any());
    }
```

And calling this test leads with Java 17 leads to the following error message OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended

java.lang.NullPointerException: Cannot invoke ”Object.getClass()” because ”list” is null
It seems that calling a function within a mock which accesses private members fails.

Another problem is the count of calling methods inside a function of the application which is mocked. The count can be affected by e.g. someone changes the incoming data. Here an example in which the check which can be broken very easy. Here an example:

The app code

```
public List<String> readTestTextFileCorrectFMock(File inFile) {
        List<String> fileCollection = new ArrayList<>();
        try {
            BufferedReader reader;
            if (inFile != null) {
                reader = this.getFileInStream(inFile);
            } else {
                reader = this.getTempFileInStream();
            }
            String line = this.readOneLine(reader);
            while (line != null) {
                fileCollection.add(line);
                line = this.readOneLine(reader);
            }
        } catch (IOException ex) {
            throw new IllegalStateException("error", ex);
        }
        return fileCollection;
    }
```

The test code

```
@Test
    public void readTestTextFileDataDep() throws URISyntaxException,
        IOException {
         MockedSample mockedVersion2 = Mockito.mock(MockedSample.class,
            Mockito.CALLS_REAL_METHODS);
         URL inputURL = this.getClass().getResource("/test.txt");
         File in = new File(inputURL.toURI());
         List<String> result =
            mockedVersion.readTestTextFileCorrectFMock(in);
         verify(mockedVersion, times(1)).getFileInStream(any());
         verify(mockedVersion, times(5)).readOneLine(any());

         inputURL = this.getClass().getResource("/testTwo.txt");
         in = new File(inputURL.toURI());
         result = mockedVersion2.readTestTextFileCorrectFMock(in);
         verify(mockedVersion2, times(1)).getFileInStream(any());
         verify(mockedVersion2, times(9)).readOneLine(any());
    }
```

In the test code we see that the different count of the read functions has its reason in two different files. But the point is that there maybe the problem that anyone changes the test file without think about that a few lines more or less changes the outcome of the test. One solution here can be:

```
    @Test
     public void writeTestTextFileAndRead() throws IOException{
         boolean ok = mockedVersion.writeTestTextFile(this.tempFile,
                 this.content);
         List<String> result =
            mockedVersion.readTestTextFileCorrectFMock(this.tempFile);
         result.forEach(System.out::println);
         int countLines = this.content.size(); // now we are all times
            ok with it
         verify(mockedVersion, times(countLines)).writeOneLine(any(),
            any());
         verify(mockedVersion, times(countLines +
            1)).readOneLine(any());
    }
```

Here we introduce a variable for the count initialised by getting the list size. But sometimes we do not have this information in access easily so be careful which method calls you really count. These things can also happen with other logic used when mocking objects or methods. The most mocking frameworks have many objects and mockito for example has an integrated test runner. So the mocking frameworks themselves are very complex and there are many ways to fall in a pit. This is one of the reasons you should not count completely on mocking to solve problems but it should be seen as one tool for testing. Do not mock because mocking is cool but because in one place it is really needed. And here is the general advice: Keep the code as simple as possible. Here one example about really mock because the count example is some kind of observation:

```
@Test
public void testThrowException() {
     Object o = Mockito.mock(Object.class);
     Mockito.when(o.toString()).thenThrow(RuntimeException.class);
                               .thenThrow(IllegalArgumentException.class);
```

```
    assertThatThrownBy(() ->
       o.toString()).isInstanceOf(RuntimeException.class);
    assertThatThrownBy(() ->
       o.toString()).isInstanceOf(IllegalArgumentException.class);
}


public void testMyFunction() throws Exception {
    MyInterface mock = mock(MyInterface.class);
    when(mock.myFunction(anyString())).thenAnswer(new Answer<String>()
       {
    @Override
    public String answer(InvocationOnMock invocation) throws Throwable
       {
        Object[] args = invocation.getArguments();
        return (String) args[0];
    }
    });

    assertEquals("someString",mock.myFunction("someString"));
    assertEquals("anotherString",mock.myFunction("anotherString"));
}
```

Examples from stackoverflow and GitHub free to copy

In this example we can see a glance of the complexity this mocking frameworks have. You can spend time writing whole 'mocking' programs ;-). This is a problematic thing because if the tests are too complex there is the danger that we test the test in the end not the application because the test logic has always to be adapted to the logic of the application in if this test logic is complex the adaptation is complex and time wasting and can have errors in the test we do not recover.

'Fake' components

Fake components are components which have the same interface as the real component but do anything but returning values which are the right values to test the functionality. In the project / product I worked on the last years there was a component which was to expensive to buy it for tests only so we written a fake of the library doing reverse engineering and the component library we wrote was configurable for the parameters the return values and what should happen e.g. throw Exception XYZ. Let us use a interface working with Key Value Pairs and CSV. We have a 'Real World ' Implementation. In this example the advantages and disadvantages will be shown. Our tiny application has a 'library' part which is 'mocked away'.

The real component

The component with the same interface ad returns for test reasons only

21.4 The top 10 NOT TODO's in tests

In this place I will show some things which should never happen or never being done writing and designing tests. It is important that tests as well as the target of a test is always clear, clean and strictly described and implemented.

- Don't test against a only partly- or unknown target

- Don't get too fancy when implementing a test keep the test simple
- Do not check more than one aspect (parameter with different values, transaction with different kinds of input and result, ...) otherwise you do not know why exactly the test is failing
- Don't use too much mocking if it is not necessary
- Don't use a test description before you have double checked it if the description is not complete and proper the result cannot be good
- Don't make tests which have no description at all
- Don't be worried about ask the architect or developer questions about the functionality if you are not sure
- A not to do is testing only the values like a developer would set them e.g. buffer-sizes for file transfer
- Do not forget to test what happens if some things are not initialised correctly
- Do not test like it is all times good weather and don't test in a nutshell (a unrealistic environment)

21.5 A little view at coding styles necessary for testable code a few examples

Here a little excursion regarding the coding style and how it can make testing of the code nearly impossible: One important thing is to have a good style of coding doing conditional branches in the code: For this have a look at Javaś different styles of the switch statement:
The output of the following functions is printed that way : The thing is printed by;

```
StringBuffer buffer = new StringBuffer()
                  .append("Normal Simple: " + pureOldSwitchCase(SIMPLE))
                  .append(" ; ").append("Normal Triple: " +
                     pureOldSwitchCase(TRIPLE))
                  .append(" ; ").append("Normal Double: " +
                     pureOldSwitchCase(DOUBLE))
                  .append(" ; ").append("Normal Fall through: " +
                     pureOldSwitchCase(FALLTHROUGH))
                  .append(" ; ").append("Normal default: " +
                     pureOldSwitchCase(NONE))
                  .append("\n\n")
                  .append("With trap Simple: " +
                     pureOldSwitchCaseWithTraps(SIMPLE))
                  .append(" ; ").append("With trap Triple: " +
                     pureOldSwitchCaseWithTraps(TRIPLE))
                  .append(" ; ").append("With trap  Double: " +
                     pureOldSwitchCaseWithTraps(DOUBLE))
                  .append(" ; ").append("With trap  Fall through: " +
                     pureOldSwitchCaseWithTraps(FALLTHROUGH))
                  .append(" ; ").append("With trap  default: " +
                     pureOldSwitchCaseWithTraps(NONE))
                  .append("\n\n")
                  .append("Upgraded Simple: " +
                     upgradedOldSwitchCase(SIMPLE))
                  .append(" ; ").append("Upgraded Triple: " +
                     upgradedOldSwitchCase(TRIPLE))
                  .append(" ; ").append("Upgraded Double: " +
                     upgradedOldSwitchCase(DOUBLE))
                  .append(" ; ").append("Upgraded Fall through: " +
                     upgradedOldSwitchCase(FALLTHROUGH))
                  .append(" ; ").append("Upgraded default: " +
                     upgradedOldSwitchCase(NONE))
                  .append("\n\n")
                  .append("Upgraded Simple with traps: " +
                     upgradedOldSwitchCaseWithTraps(SIMPLE))
                  .append(" ; ").append("Upgraded Triple with traps: " +
                     upgradedOldSwitchCaseWithTraps(TRIPLE))
                  .append(" ; ").append("Upgraded Double with traps: " +
                     upgradedOldSwitchCaseWithTraps(DOUBLE))
                  .append(" ; ").append("Upgraded Fall through really
                     fall: " +
                     upgradedOldSwitchCaseWithTraps(FALLTHROUGH))
                  .append(" ; ").append("Upgraded default with traps: "
                     + upgradedOldSwitchCaseWithTraps(NONE))
                  .append("\n\n")
                  .append("new Simple Expr: " +
                     newSwitchCaseExpr(SIMPLE))
```

```
                .append(" ; ").append("new Triple Expr: " +
                    newSwitchCaseExpr(TRIPLE))
                .append(" ; ").append("new Double Expr: " +
                    newSwitchCaseExpr(DOUBLE))
                .append(" ; ").append("new Fall through Expr: " +
                    newSwitchCaseExpr(FALLTHROUGH))
                .append(" ; ").append("new default Expr: " +
                    newSwitchCaseExpr(NONE))
                .append("\n\n");
        System.out.println(buffer);
```

Let us first have a look at an old school nearly clean switch block

```
/**
     * In this function the break statements are set correctly and
        strict.
     * Except the FALLTHROUGH: case that is no good style but in most
        cases acceptable.
     * @param rule the switch case criteria
     * @return the result for the selected case
     */
     public static Integer pureOldSwitchCase(ExecRules rule) {
        Integer x = 8;
        Integer y = 12;
        Integer result = 0;
        switch (rule) {
            case SIMPLE:
                result = x +3;
                break;
            case FALLTHROUGH: // this is seen as ok but no good style
                y *=9;
                result += 2 * y;   /*as you see here the break is
                    missing the consequence is that the
                 execution falls through until the next break is set
                    here before case TRIPLE: */
            case DOUBLE:
                result += y;
                break;   /*break sets the continuation to the
                    statement after the switch actively*/
            case TRIPLE:
                result += 7;
                break;
            default:
                result += 3;
        }

        return result;
    }
```

The output calling it with all parameters will be:
Normal Simple: 11; Normal Triple: 7 ; Normal Double: 12 ; Normal Fall through : 324 ; Normal default: 3

Let us first have a look at an old school dirty coded switch block

```
/**
     * In this function the break statements are set wild or never.
        That kind of
     * coding style leads to the fact that especially beginners cannot
        change the code so that all things work again.
     * There is no need to write such irritating code
     * Except the FALLTHROUGH: case that is no good style but in most
        cases acceptable.
     * @param rule the switch case criteria
     * @return the result for the selected case
     */
    public static Integer pureOldSwitchCaseWithTraps(ExecRules rule) {
        Integer x = 8;
        Integer y = 12;
        Integer result = 0;
        switch (rule) {
            case SIMPLE:
                result = x +3;
            case FALLTHROUGH:
                y *=9;
                result += 2 * y;
            case DOUBLE:
                result *= 2;
                result += y;
                break;
            case TRIPLE:
                result += 7;
            default:
                result += 3;
        }

        return result;
    }
```

The output calling it with all parameters will be:
With trap Simple: 562 ; With trap Triple: 10 ; With trap Double: 12 ; With trap Fall through: 540 ; With trap default: 3

Let us first have a look at an new clean coded switch block

```
 /**
     * Here we see the 'new' keyword yield with which a result is
        given back and which also, like break, sets the continuation
     * to go on after the switch. In this case here each branch ends
        with a yield so we have a clear and clean logic.
     * @param rule the switch parameter
     * @return the switch result
     */ public static Integer upgradedOldSwitchCase(ExecRules rule) {
        Integer x = 8;
        Integer y = 12;
        Integer result = 0;
        return switch (rule) {
            case SIMPLE:
                yield x +3;
```

```
            case FALLTHROUGH:
                yield y * 9;
            case DOUBLE:
                yield result + y;
                /*yield result = result + y; marked as unreachable */
            case TRIPLE:
                yield result + 7;
            default:
                yield result + 3;
        };
```

The output calling it with all parameters will be:
Upgraded Simple: 11 ; Upgraded Triple: 7 ; Upgraded Double: 12 ; Upgraded Fall through: 108 ; Upgraded default: 3

Now let us have a look at an new tricky coded switch block

```
/**
     * Here one fallthrough takes place which is no clean coding since
        each branch should deliver
     * one defined return via yield
     * @param rule the switch value
     * @return the result of the switch
     */
    public static Integer upgradedOldSwitchCaseWithTraps(ExecRules
        rule) {
         Integer x = 8;
         Integer y = 12;
         Integer result = 0;
         return switch (rule) {
             case SIMPLE:
                 yield x +3;
             case FALLTHROUGH: // this is seen as ok but no good style
                 y *=9;
                 result += 2 * y;
             case DOUBLE:
                 result *= 2;
                 yield result + y;
             case TRIPLE:
                 yield result + 7;
             default:
                 yield result + 3;
         };

    }
```

The output calling it with all parameters will be:
Upgraded Simple with traps: 11 ; Upgraded Triple with traps: 7 ; Upgraded Double with traps: 12 ; Upgraded Fall through really fall: 540 ; Upgraded default with traps: 3

Now let us see how the switch expression works here the continuations are implicit and so coding which is too fancy is nearly impossible

```
/**
  * Here we see a switch EXPRESSION instead of a switch statement.
     All continuations are implicitly set
  * @param rule the switch parameter
  * @return the switch expression return
  */
 public static Integer newSwitchCaseExpr(ExecRules rule) {
     Integer x = 8;
     Integer y = 12;
     Integer result = 0;
     return switch (rule) {
         case SIMPLE ->
                 x + 3;   /*only THIS statement is executed and
                    produces a  defined result */
          case FALLTHROUGH ->
                 y * 9;
         case DOUBLE -> /*only THIS statement is executed and
            produces a  defined result */
                 result + y;
         case TRIPLE ->   /*only THIS statement is executed and
            produces a  defined result */
                 result + 7;
         default ->.  /*only THIS statement is executed and
            produces a  defined result */
                 result + 3;
     };
 }
```

The output calling it with all parameters will be:
new Simple Expr: 11 ; new Triple Expr: 7 ; new Double Expr: 12 ; new Fall through Expr: 108 ; new default Expr: 3

Chapter 6: Further material

22 Code examples and logic

In this part some code snippets and explanations take place which were only mentioned in the article. Here the list:

1. The Model View Controller Architecture

22.1 The MVC Model as clean separation of functionality

23 References

1. GITLAB CD/CI Guide and descriptions
2. Definition of Functional and NON Functional Testing
3. Regression Testing Tool and Description Regression Test IO
4. A interesting article about mocking.... Is it really useful to mock all ??? ;-)