

Testing is a MUST

Version 0.8

Harald Glab-Plhak<hglabplhak@icloud.com>

November 8, 2025

About automated test methods and more.....

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 2 |
| 1.1 | The different Test Methods in Test automation (UNIT -Test , Functional Test, Performance Functional Test) | 2 |
| 1.2 | UNIT Testing in an automated environment | 3 |
| | Chapter: 1 An Introduction to test types and automation of tests | 3 |
| 2 | The different ways to divide tests in categories | 3 |
| 2.1 | From the technical point of view | 3 |
| 2.2 | Another way to make categories is delivery, user and release oriented | 3 |
| 3 | A example how to design and realise an automated test | 4 |
| 3.1 | And here are a few tests for it to see how UNIT Testing works: | 6 |
| 3.2 | First Resume | 6 |
| 4 | How to design Automated Functional Tests | 7 |
| 4.1 | The Basics | 7 |
| 5 | Performance Testing | 7 |
| 6 | Release Testing | 8 |
| 6.1 | Installation / Integration / Migration Testing | 8 |
| 6.2 | Regression Testing | 9 |
| 7 | An example for our scenario | 9 |
| 7.1 | The Development / Test Environment | 9 |
| 7.2 | Test case definition cookbook | 10 |
| 8 | ANOTHER POINT WHY NOT TESTS FIRST Test Driven Development | 11 |
| 8.1 | Test driven development a short view. | 11 |
| 8.2 | How to realise this thoughts | 11 |
| 8.3 | Example of such a test in Java | 11 |
| | Chapter: 2 Deep dive into test automation | 13 |

| | | |
|-------------------|--|-----------|
| 9 | At First again the testing types and the way of automation for the types | 13 |
| 9.1 | UNIT Tests | 13 |
| 9.2 | Functional Tests | 13 |
| 9.3 | Use Case Tests | 13 |
| 9.4 | FeatureTests | 13 |
| 9.5 | Minor Release / Major Release Tests | 13 |
| 9.6 | Upgrade / Migration Tests | 13 |
| 9.7 | Performance Tests | 14 |
| 9.8 | PenetrationTests / Security Tests - Mostly Service Tests | 14 |
| 9.9 | Regression Tests | 14 |
| 9.10 | Usability Tests | 14 |
| 9.11 | AutomatedTests | 14 |
| 9.12 | InteractiveTests | 14 |
| 9.13 | AcceptanceTests | 14 |
| 9.14 | Resume | 14 |
| 10 | Look at UNIT testing first | 15 |
| 11 | Functional Tests development in parallel | 16 |
| 12 | Tests for Use Cases | 16 |
| 13 | Setup / Build automated test environment | 16 |
| 13.1 | Example with tool chain | 16 |
| Chapter: 3 | Clean code and coding rules to make testing easier | 18 |
| 14 | A little view at coding styles necessary for testable code a few examples | 18 |
| 14.1 | At first have a look on JAVA | 18 |

1 Introduction

1.1 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 whether input and output is correct. Here we use **Functional Tests**. The ¹.
- 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** ²

¹**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.

²**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*³ and the best is to let them run after each change and prior deployment of each minor and major release

1.2 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

2 The different ways to divide tests in categories

2.1 From the technical point of view

1. The easiest technical difference is:
 - |*BLACK BOX Test*: This kind of test only knows about the interface to test that means the defined Input and the awaited Output
 - |*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.

2.2 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
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.

³*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

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.

3 A example how to design and realise an automated test

As an algorithmic 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;
    }
}

```

3.1 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() {

    }
}
```

3.2 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 an exponential growth of possibilities what would have gone wrong.

The main error by people who have less experience in Testing is to check the parameters compound in one test function. ***THIS IS A NOT TO DO!!!!***

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 these 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.

4 How to design Automated Functional Tests

4.1 The Basics

For the automated Functional Testing we can set up an 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.

5 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 we 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 in the best case automatised 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.

6 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.

6.1 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.

6.2 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.

7 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 necessary setup for this storage solutions.

7.1 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

7.2 Test case definition cookbook

To write test cases with 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 that a legal hold can be set for the document. In addition he needs by the law a retention time of 10 y for the document in this time the document has to be protected from deletion. During the whole time it 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 in 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*⁴) 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 (Elliptic 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 too 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 and 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

⁴ *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

Thats all for the first lesson. Thank you!

8 ANOTHER POINT WHY NOT TESTS FIRST Test Driven Development

8.1 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.

8.2 How to realise this thoughts

To realise this thoughts 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.

8.3 Example of such a test in Java

```
public class ATest extends BaseOfTests {

    private Integer derfault = 110;
    /* C TOR*/
    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. The point is that in Test Driven Development we do not think at first about the technical details and afterwards wrap the technical details some kind of interface. No the opposite is the case: We think about which values are passed to a function and how the outcome must look like. Same for the used parameter - data types and the return value(s) datatype(s). We think about datastreams and the steps which work on them and each "worker" has well defined parameters and a well defined outcome for the specific values given as arguments. And NOW we can think about the technical details while we code inside a well defined interface.

Chapter 2: Deep dive into test automation

9 At First again the testing types and the way of automation for the types

9.1 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.

9.2 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.

9.3 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

9.4 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.

9.5 Minor Release / Major Release Tests

Release Tests are tests where the whole features and Use Cases are done for a release which has to be shipped. This includes Upgrade / Migration Tests, Performance Tests, UNIT Tests, Functional Tests and so on. The focus will be to do as much tests as possible automated so that they all times can be repeated in a safe way and to safe time.

9.6 Upgrade / Migration Tests

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.

9.7 Performance Tests

The focus of performance tests is 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 the customers environment. In the best case all other applications driven by the customer run in parallel as it is in reality.

9.8 PenetrationTests / 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.

9.9 Regression Tests

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

9.10 Usability Tests

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.

9.11 AutomatedTests

Automated tests are done for the parts which can be accessed in a programable way there are also tools making it possible to automate some UI tests not only backend functionality. Automated tests are normally embedded in a system which is designed to have all steps from check in to compile and deliver and test.

9.12 InteractiveTests

Interactive tests are defined for using the UI and they are defined to see if the functionality is completely accessible in a way that all tasks can be done by a user.

9.13 AcceptanceTests

Acceptance tests are tests to see how good a interface or a UI is accepted by the one who uses it. These tests are partly strict defined and on the other hand there are tests where the user gets a task and can fill out a form with different criteria. These criteria are chosen in a way that we can see how satisfied the user is with the way how he can come to the solution. They are defined in a way that they state if it is easily possible to use the functionality.

9.14 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.

10 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*.⁵ 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>() {
            @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
```

⁵UNIT 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

```

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))

```

11 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.

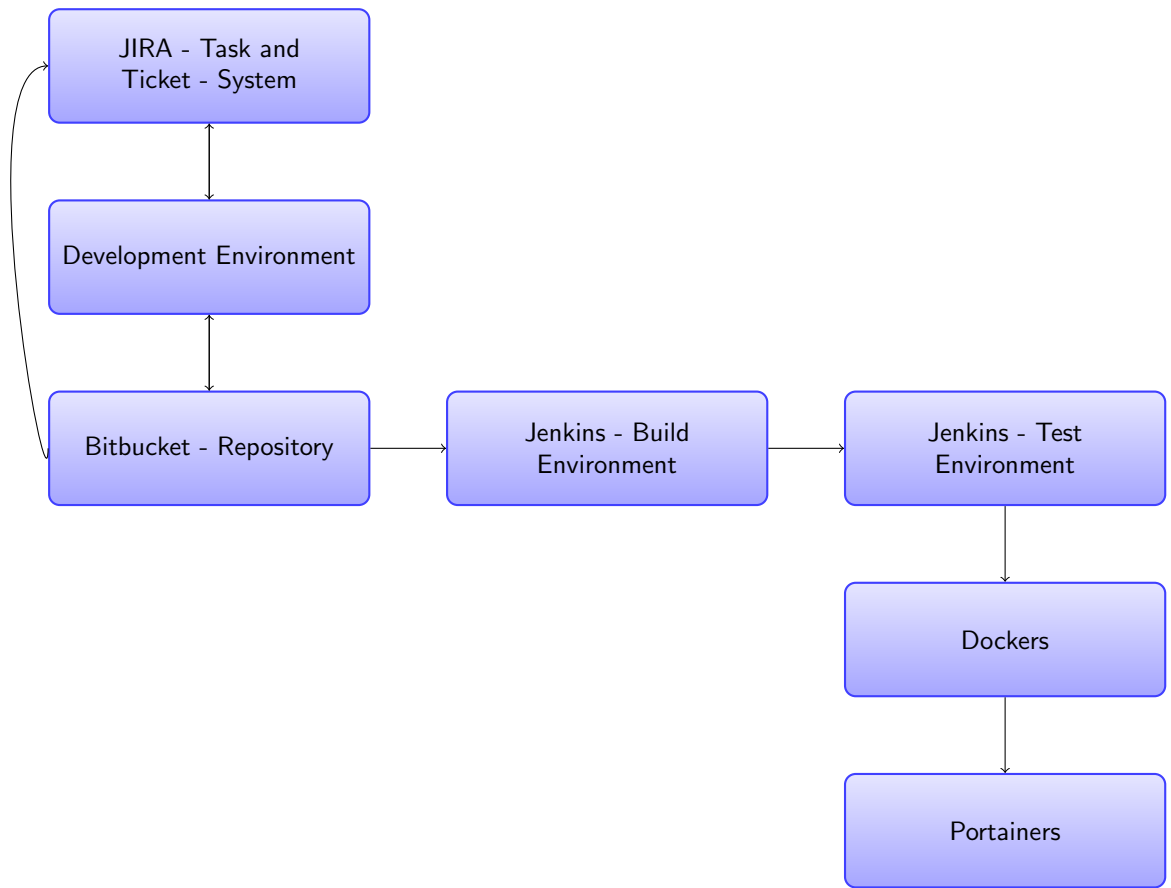
12 Tests for Use Cases

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.

13 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 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.

13.1 Example with tool chain



Chapter 3: Clean code and coding rules to make testing easier

14 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 Java's different styles of the switch statement:

14.1 At first have a look on JAVA

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 with errors is more difficult

```
/**
 * 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:

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