Software testing approaches: comparing Black box and White box testing techniques

BAHRI Ahmed & DHAOU Cyrine
Télécom SudParis

Software testing aims at assuring the software quality. Given the knowledge of the system under test, one can derive the test suites under various testing assumptions. Black box testing concerns the case when only the access to the interfaces is provided; white box testing, on the other hand, 'assumes' that the source code is given and can be inspected. The goal of the project is to compare these two paradigms and draw certain conclusions about the fault coverage, test suite generation performance, etc. The analysis is planned to be supported by the experimental evaluation.

Keywords: Testing technique, functional testing, White-box, Black-box, Static testing, Dynamic testing

1. Introduction:

Testing is a vital part of software development. Different types of software testing are performed to accomplish different targets when testing a software product, application,or a program with the point of discovering the defect to influence our software product to perform well. If testing is efficiently done, it will expel all the defects from the software product and determine how well it works. Software testing assures the quality of the software. The quality, as defined by the ISO/IEC 25010[1], relies on functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability.

According to Dr. Natalia Kushik "Software testing is the process of applying inputs, observing output responses and drawing conclusions."[2]

Different types of testing in software testing exist each having, a characterized test objective, test system, and test expectations.

The objective of having different types of testing in software engineering is to certify the Application under Test for the characterized test objective.

There are different ways of classifying software testing techniques. In our report, we will start by dividing them into Static testing and Dynamic testing techniques. We will make another classification relying on the testing methodologies. Then we will classify them into black box and white box testing techniques according to testing assumptions.

This study emphasizes the need to investigate various testing techniques in software testing field. We have conducted a review obtained from state of the art and Experimental testing.

1.1. Static and Dynamic Software testing techniques:

We can divide software testing techniques into two categories: Static testing and Dynamic testing.

Static testing: The authors of [3] define Static testing to be the process of examining the software without executing it. The techniques are concerned with the analysis and checking of system such as the requirements documents, design diagrams and the program source code, either manually or automatically, without executing the code.

Why do we use static testing?

Static testing allows us to test the software without having to execute which can be useful when the software is slow to execute or when we can't execute it at all. Examining the code or verifying the requirements, can permit the detection of defects at an early state and may represent a good way to get fewer defects at a later stage of the testing process and that way reduce the testing cost in terms of processing, memory and time.

What does static testing test?

According to the authors of [src: https://www.guru99.com/testing-review.html] [4] static testing can be used in:

- Unit Test Cases
- Business Requirements Document (BRD)
- Use Cases
- System/Functional Requirements
- Prototype
- Prototype Specification Document
- DB Fields Dictionary Spreadsheet
- Test Data
- Traceability Matrix Document
- User Manual/Training Guides/Documentation
- Test Plan Strategy Document/Test Cases
- Automation/Performance Test Scripts

Static testing can be divided in syntactic or semantic testing:

Syntactic testing may include the reviews held by a design team to check that the refinements of accepted requirements are proceeding as desired through each transformation stage reviews. A static software inspection is a formal evaluation of the work items of a software product. The technique has proved to be an effective technique for the design, code and test phases. A software inspection is led by an independent moderator with the intended purposes of effectively and efficiently finding defects in the development process, recording these defects as a basis for analysis and initiating re-work to correct such defects.

Semantic testing includes formal methods such as proof of correctness. Proof of correctness is a mathematical method of verifying the logic of a program or program segment. In order to be able to do so, the specification must be expressed formally as well. We achieve this by expressing the specification in terms of two assertions which comes before and after the program's execution, respectively. Next, we prove that the program transforms one assertion (the precondition) into the other.

We have noted that static testing is usually white box, but it can also be used in black box testing by examining the specifications and requirements and looking for errors to make sure the requirements are not violated.

Dynamic Testing is defined as a software testing type, which checks the dynamic behavior of the code. Dynamic testing involves the execution of a piece of software with test data and a comparison of the results with the expected output. In other words, working with the system with the intent of finding errors.

Dynamic testing techniques are generally divided into the two categories: black and white box testing as defined by the authors of [3].

Black box testing: Input data is designed to generate variations of outputs without regard to the logic actual functions. The results are predicted and compared to the results to determine the success of the test. Black box testing is used to check if the product conforms to its specifications. The problem is that we don't know how much of the code we are testing, and the software can be performing an undesirable task that we are not able to detect.

White box testing opens the box and looks at the specifications of the application to verify how it works. Tests use logic specifications to generate variations of processing and to predict the resulting outputs. We can examine the code in detail and achieve a level of test coverage. But if the program is not performing one of its desired tasks function, we might not be able to detect that.

The methods for generating test inputs are either deterministic or probabilistic.

Deterministic methods for generating test inputs usually take advantage of information on the target software in order to provide guides for selecting test cases, the information being depicted by means of test criteria. Both functional and structural testing strategies use a systematic means to determine subdomains.

Statistical testing is based on an unusual definition of random testing. We are assuming that statistical testing is model based. It aims to provide a "balanced" coverage of a model of the target software, no part of the model being seldom or never exercised during testing. With this approach, the method for generating statistical test patterns combines information provided by a model of the target software, that is, by a test criterion with a practical way of producing large sets of patterns, that is, a random generation.

Random testing: Test data is selected according to the way in which software is operated.

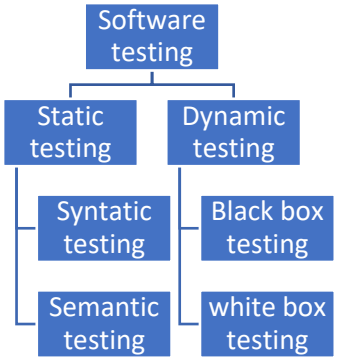
Why do we use dynamic testing?

Relying on the researches made by the authors of [5], we think that dynamic testing might reveal defects that are not covered through static analysis. Dynamic testing is often used for detecting security threats to verify the robustness of an application for example.

Disadvantages of dynamic testing:

According to the authors of [5]:

- Dynamic testing can be time consuming because it executes the application/software or code which might require a big amount of resources
- Dynamic testing might increase the cost of project/product because it does not start early in the software lifecycle and hence any issues fixed in later stages can result in an increase of cost.



We can consider static and dynamic testing as two complementary parts to be performed to test a software. The static testing is the verification process and the dynamic testing is the validation process. In software testing, static testing would be the first measure and dynamic testing the second measure used to verify if the software products meet the requirement specifications.

2. Testing methodologies:

According to the authors of https://www.guru99.com/functional-testing-vs-non-functional-testing.html, we can divide software testing in functional testing and non-functional testing techniques:

Functional testing is performed to verify that all the features developed are according to the functional specifications, and it is performed by executing the functional test cases, in functional testing phase, system is tested by providing input, verifying the output and comparing the actual results with the expected results. There are different types of testing in functional testing and we will consider the following testing techniques:

- Unit testing: Unit is a small piece of code which is testable, Unit testing is performed at individual unit of software by developers.

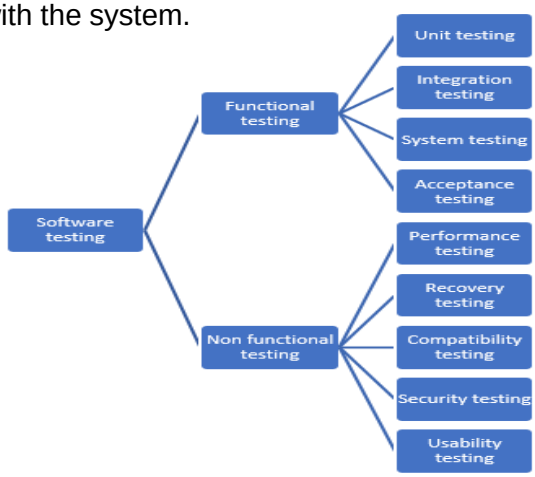
- Integration testing: Integration testing is the testing which is generally performed after unit testing and is performed by combining all the individual units which are testable by developers or testers.

- System testing: System testing is performed to ensure whether the system performs as per the requirements and is generally performed when the complete system is ready, it is performed by testers when the Build or code is released to QA team.

- Acceptance testing: Acceptance testing is performed to verify whether the system has met the business requirements and is ready to use or ready for deployment and is generally performed by the end users.

Non-Functional testing: checks the performance, reliability, scalability and other non-functional aspects of the software system. It should be performed after functional testing. Non-Functional testing describes how good the product works.

- Performance testing: Performance testing is performed to check whether the response time of the system is normal as per the requirements under the desired network load.
- Recovery testing: Recovery testing is a method to verify how well a system can recover from crashes and hardware failures.
- Compatibility testing: Compatibility testing is performed to verify how the system behaves in different environments.
- Security testing: Security testing is performed to verify the robustness of the application. The purpose is to ensure that only the authorizes users/roles are accessing the system.
- Usability testing: Usability testing is a method to verify the usability of the system by the end users to verify how comfortable the users are with the system.



3. Black box and white box testing techniques:

3.1. Black Box Testing Techniques:

Definition:

The authors of (src: https://www.guru99.com/black-box-testing.html) define Black box testing to be a testing technique in which functionality of the Application Under Test (AUT) is tested without looking at the internal code structure, implementation details and knowledge of internal paths of the software. This type of testing is based entirely on software requirements and specifications. In black box testing, we assume that the tester has only access to the interfaces and we focus on the inputs and the outputs of the software system.

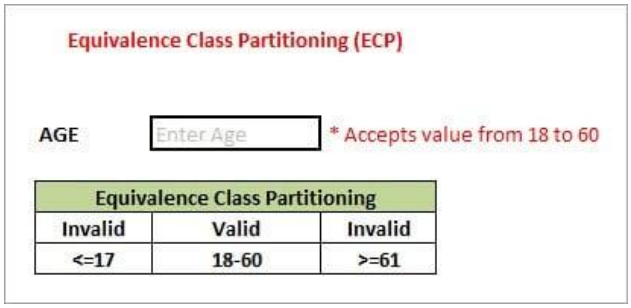
Programmers usually focus on the positive testing by validating the expected behavior of the source code. Tester adds values to black box testing by making both positive and negative testing because they know that most of the flaws are found from the unexpected behavior.
Here are some of the techniques based on [Dr Moh. Ehmer Khan [6]] research of Black box testing. Notice that every example in this part is hypothetical testing to understand the technique and it is not a tool of testing.

- Equivalence class technique

As mentioned in the essay of “Black Box And White Box Comparison Computer Science Essay” (published in 2018, UK): In this technique the software data is divided into partitions of data and test cases are derived from it. This technique ensures that test cases are derived to cover partitioned data. According to this method one test case is enough for each partition to check the behavior of the program. By using equivalence class technique test cases can be reduced because the test case of a partition will not find any new faults in the program. The recommended black box approach for selecting equivalence class values includes values at the

beginning, in the middle and end of the data range. In this way, we can maintain the test coverage while we can reduce a lot of rework and most importantly the time spent. [7]

Example 1: Assume we have to test a field which accepts Age 18-60



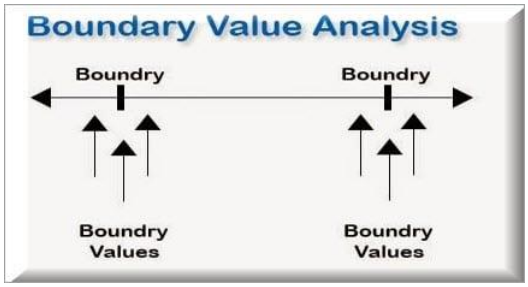
As presented in the above image, an "AGE" text field accepts the numbers between 18 and 60. So we divide the domain into three set of classes or groups.

We have thus reduced the test cases to only 3 test cases based on the formed classes and thus we cover all the possibilities. So, testing with one value from each set is enough to test the above scenario.

- Boundary Value Analysis technique

The same essay mentioned above [Black Box And White Box Comparison Computer Science Essay [7]] In this technique the analysis is extended of beginning and ending inputs value possibilities of an equivalence class. The boundary values are around the beginning and ending of a portion and functional errors from input and output data occurs around these boundaries. The boundary analysis begins by identifying smallest value increment in a specific equivalence class. This increment of smallest value is called boundary value epsilon. It is used to calculate +/- value around the beginning and ending values in an equivalence class.

Example 2:



If we want to test a field where values from 0 to 500 should be accepted then the boundary values will be: 0-1, 0, 0+1, 500-1, 500, and 500+1. Instead of using all the numbers from 0 to 500, we just use 0, -1, 1, 499, 500, and 501.

- Expected result coverage Technique

Based on the author "Black Box and White Box Comparison Computer Science Essay" ,the expected result coverage technique focuses on output test values for related input value. The author of Black Box and White Box Comparison Computer Science Essay claims that To get the expected results we need to find the business rules in the application requirement. The difference between expected results and actual results for any combination of input should cause further analysis to determine if the difference is faulty test design, unclear ambiguous rules or program error. [7]

Example 3:

Take an example of a bank that gives interest rate for the Male senior citizen as 10% and for rest of the people 9%.

Decision Table / Cause-Effect

Decision Table	Rule 1	Rule 2	Rule 3	Rule 4
Conditions				
C1 - Male	F	F	T	T
C2 - Senior Citizen	F	T	F	T
Actions				
A1 - Interest Rate 10%				X
A2 - Interest Rate 9%	X	X	X	

In this example condition, C1 has two values as true and false, condition C2 also has two values as true and false. The number of total possible combinations would then be four. This way we can derive test cases using decision table.

- *Error Guessing*

In this technique, Tester judge where bugs or errors can be hidden. Tester tests the application with the past experience and creates test cases to validate the different tasks. There is no specific tool used for this technique. [Black Box And White Box Comparison Computer Science Essay, UK 2018 [7]]

Few common mistakes that developers usually forget to handle:

- Divide by zero.
- Handling null values in text fields.
- Accepting the Submit button without any value.
- File upload without attachment.
- File upload with less than or more than the limit size

3.2. White Box Testing Techniques:

White box testing can be used to verify the correctness of the software's statements or the correctness of the code paths, or conditions, or loops and even data flow. The prerequisites of white box testing include the software requirements, use cases, the executable program, its data and its source code. Based on our state of the art, here are the techniques most used for white box testing. [7, 8]. So, all our definitions in this part is a conclusion from these two articles.

- *Statement Coverage Technique*

This technique focuses on determining what percentage of the source code lines in a program has been executed. A hypothesis in this context is that the higher the source code test coverage, the fewer will be the defects found later. The practical conclusion is the that new, unexecuted code lines are just a software time bomb waiting to explode at the most inopportune moment in production [7]

Example 4:

Consider the below simple pseudo-code:

INPUT C
IF C>100
PRINT "ITS DONE"

For Statement Coverage – we would only need one test case to check all the lines of the code. That means:

If we consider TestCase_01 to be (C=400), then all the lines of code will be executed.

Now the question arises: Is that enough?

What if we consider the Test case as C=45?

Because Statement coverage will only cover the true side, for the pseudo code, only one test case would NOT be sufficient to test it. As a tester, we have to consider the negative cases as well.

- Branch coverage technique

This technique focuses on determining what percentage of source code branch (true/false) logic in a program has been execute. Testing simple condition branches before the compound condition branches require fewer initial test actions. The developer needs to choose a test value that will force a true branch and any test value that will force a false branch, just two test values per branch.[7]

Example 5:

So now the pseudocode becomes:

NPUT C
IF C>100
PRINT "ITS DONE"
ELSE
PRINT "ITS PENDING"

As we can see Statement coverage is not sufficient to test the entire pseudo code, we would require Branch coverage to ensure that we have a wider coverage.

So, for Branch coverage, we would require two test cases to complete the testing of this pseudo code

TestCase_01: C=400
TestCase_02: C=30

With this, we can see that each and every line of the code is executed at least once.

- Compound Condition Coverage Technique

This technique extends the branch coverage technique to branches with compound conditions, ones that contain combinations of Boolean operators AND, OR and NOT along with pairs of parentheses, possible nested. In this technique, the challenge is to identify all the test value combinations that will evaluate to true and false for every simple condition and every Boolean combination of simple condition. [7]

- Path Coverage Technique

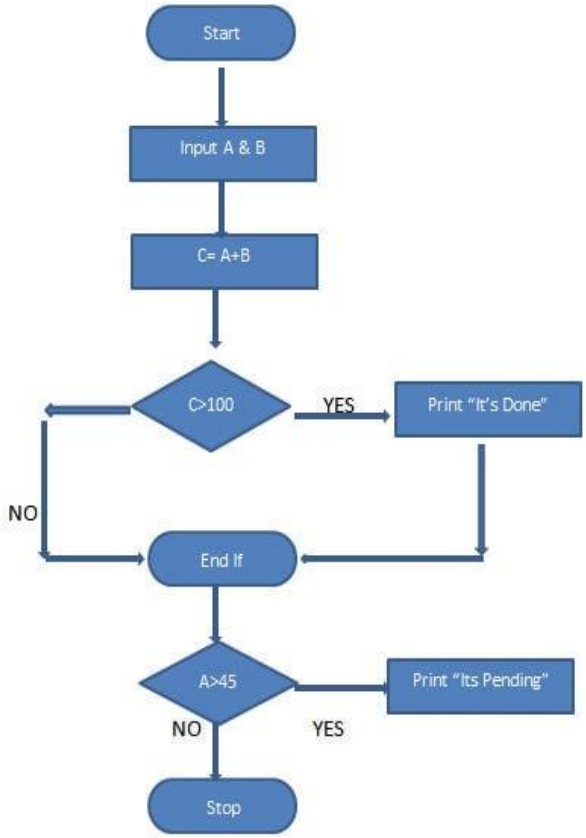
This technique focuses on determining what percentage of source code paths in a program have been traversed completely. "A source code path is the sequence of program statements from the first executable statement through a series of arithmetic, replacement input/out, branch and looping statements to a return/stop/end/exit statement." If there are 943 different paths through a program and we can determine manually or automatically that we have executed 766 of them, then we have achieved 81% path coverage. The underlying hypothesis is that the higher the path test coverage, the fewer will be the defect found later. [7]

Example 6:
Consider this pseudocode:

INPUT A & B
C = A + B
IF C>100
PRINT "ITS DONE"
END IF
IF A>50
PRINT "ITS PENDING"
END IF

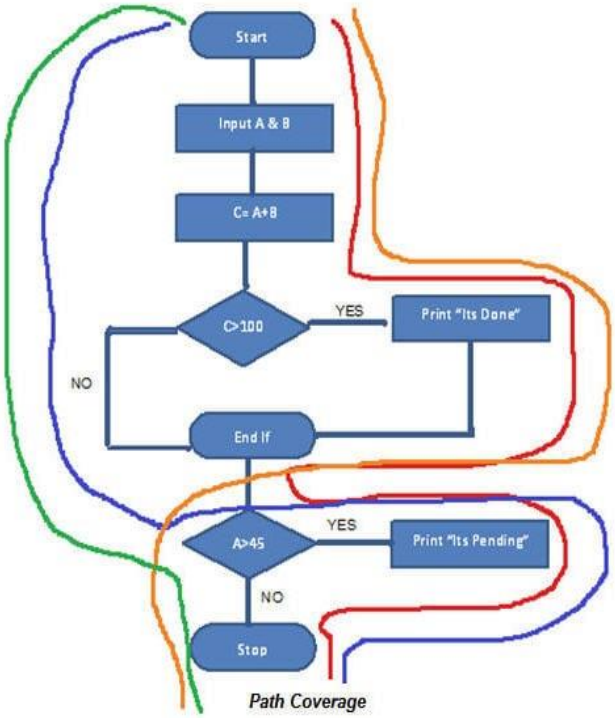
Now to ensure a full coverage, we would require 4 test cases.
How? Simply – there are 2 decision statements, so for each decision statement, we would need two branches to test. One for true and the other for the false condition. So, for 2 decision statements, we would require 2 test cases to test the true side and 2 test cases to test the false side, which makes a total of 4 test cases.

To simplify these let's consider below flowchart of the pseudo code we have:



Path Coverage

In order to have the full coverage, we would need following test cases:
TestCase_01: A=50, B=60
TestCase_02: A=55, B=40
TestCase_03: A=40, B=65
TestCase_04: A=30, B=30

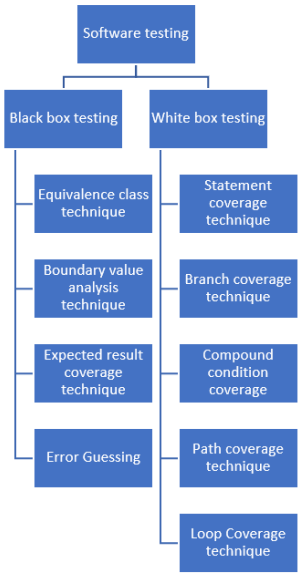


Path Coverage

So, the path covered will be:
Red Line – TestCase_01 = (A=50, B=60)
Blue Line = TestCase_02 = (A=55, B=40)
Orange Line = TestCase_03 = (A=40, B=65)
Green Line = TestCase_04 = (A=30, B=30)

- Loop Coverage Technique

This technique focuses on determining what percentage of source code loop in a program has been cycled completely. In the programming languages, there are several loops like DO..WHILE, FOR, WHILE..DO and UNTIL. Some of the loops are a clever construct of IF statements and subsequent returns to these IF statements. The objective of the loop testing is to force the program through the loop zero time, one time, n/2 times (n is terminal loop value it can be predetermined or not ,depending on the program) (specify that we are not relying on the number of iterations)n times and n+1 times. The one-time loop, n/2-time loop and n-time loop validate expected loop response at the beginning, middle and end of the longest loop. The zero-time and n+1-time loop tests for unexpected and inappropriate looping conditions.[7]



Deriving test suites:

There are different ways of deriving a test suite:

Without mathematics:

Test cases are provided semi-randomly which does not guarantee fault coverage.

With mathematics:

Software requirements are transformed into a formal model:

Model based testing (MBT) is a testing approach where test cases are automatically generated from models. The authors of (https://saucelabs.com/blog/the-challenges-and-benefits-of-model-based-testing) define the models as the expected behavior of the system under test that can be used to represent the testing strategy. According to the authors of (https://www.guru99.com/model-based-testing-tutorial.html), Behavior can be described in terms of input sequences, actions, conditions, output and flow of data from input to output. It should be practically understandable and can be reusable; shareable must have a precise description of the system under test.

Examples of Model based Testing:

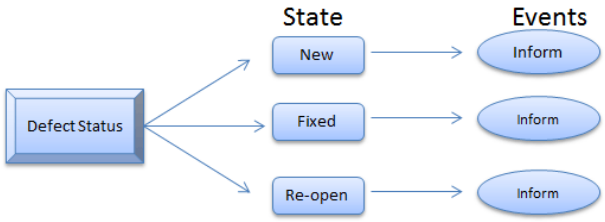
Finite State Machines:

This model helps testers to assess the result depending on the input selected. There can be various combinations of the inputs which result in a corresponding state of the system.

The system will have a specific state and current state which is governed by a set of inputs given from the testers.

Consider the example: There is a system which allows employees to log into the application. Now, a current state of the employee is "Out" and it became "In" once he signs- into the system. Under the "in" state, an employee can view, print, scan documents in the system.

State Charts is an extension of Finite state machine and can be used for complex and real time systems. they are used to describe various behaviors of the system. they have a definite number of states. The behavior of the system is analyzed and represented in the form of events for each state.



For example: Defects are raised in defect management tool with the status as New. Once it is fixed by developers, it must be changed to status Fixed. If a defect is not fixed change status to Re-open. State chart should be designed in such a way that it should call for an event for each state.

Unified Modeling Language (UML) is a standardized general-purpose modeling language. UML includes a set of graphic notation techniques to create visual models of that can describe very complicated behavior of the system.

UML has notations such as: Activities, Actors, Business Process, Components, Programming language.

4. Experimental part:

The aim of the experimental part is to test and compare the performance and the fault coverage of black box and white box testing techniques. In order to achieve this, we asked ourselves:
What benchmark case should we rely on? What language to adopt? What techniques among the black box and white box testing techniques should we compare? What tools should we use to test and compare the fault coverage of each technique? What metrics do we choose to compare the performance?

4.1. Benchmark:

We use an open-source program written in Java. Java is a very popular programming language commonly used in software testing as it provides suitable tools for software testing.
We create the Test suites for this program based on the testing technique that we want to try.
The algorithm of this program is Bubble sort which we took from https://waytolearnx.com/2018/11/tri-a-bulle-en-java.html. A simple yet a good example to evaluate the software testing techniques because it contains loops, if else conditions, functions and classes which are basic parts of any complex program. For this program we are going to use different types of testing and draw our conclusion. This figure presents the algorithm.

```
public void tri_bulle(int[] tab){
    int taille = tab.length;
    int tmp = 0;
    for(int i=0; i < taille; i++){
            for(int j=1; j < (taille-i); j++){
                    if(tab[j-1] > tab[j]){
                            tmp = tab[j-1];
                            tab[j-1] = tab[j];
                            tab[j] = tmp;
                    }
            }
    }
}
```

Bubble sort is a simple sorting algorithm. It is comparison-based algorithm: each pair of adjacent elements is compared, and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst-case complexity are of O(n2) where n is the number of items.
We will focus on **functional testing** and use white box and black box testing techniques in order to compare them in terms of fault coverage.

We will use mutation as a testing technique to measure the fault coverage for the dynamic testing techniques.
Mutation testing, according to the authors of https://fr.slideshare.net/KrunalParmar13/mutation-testing-and-mujava, is based on the assumption that a program will be well tested if a majority of simple faults are detected and removed. It introduces simple faults into the program by creating a set of mutants. Mutants are created from the original program by applying Mutation Operators which describe syntactic changes into the programming language. The objective is to measure the effectiveness of a test set in terms of ability to detect faults. If the test set can distinguish a mutant from the original program the mutant is killed otherwise it survives. The adequacy of a test set is measured by the number of mutants killed/ number of non-equivalent mutants.
The objective is to minimize the number of test sequences and preserve the fault coverage.
We are using Pitest to generate the mutants.
Our methodology is to apply the testing techniques gradually from the least resource consuming to the more resource consuming technique. This way, we hope to test the code in an efficient way.

White box static testing:

We start by examining the code. We process both syntactic and semantic verification of the code to detect early defects in our code. Static testing is less time consuming and therefore we recommend it for grey box testing.

Equivalence class testing:

The second step is to test the code dynamically to check whether the code detects invalid inputs. We will use a black box testing technique and determine the validity of our inputs.

Equivalence classes:

To sort a table by this program, the equivalence classes will be:
(a) Valid Input

- Array with integer numbers. [a range of an integer numbers is from -2,147,483,648 to +2,147,483,647]

(b) Invalid Input

- Array with numbers more than +2,147,483,647 or less than -2,147,483,648
- Array with double numbers
- Array with chars

Code coverage testing technique:

Code coverage testing technique is a white box testing technique. The objective is to determine how much of the source code is being tested by the test suite.

How is code coverage calculated?

Code coverage tools will use one or more criteria to determine how the code was exercised or not during the execution of the test suite.

The common metrics include:

Statement coverage: how many of the statements in the program have been executed.

Branches coverage: how many of the branches of the structures (if statements for instance) have been executed.

Condition coverage: how many of the Boolean sub-expressions have been tested for a true and a false value.

Line coverage: how many of lines of source code have been tested.

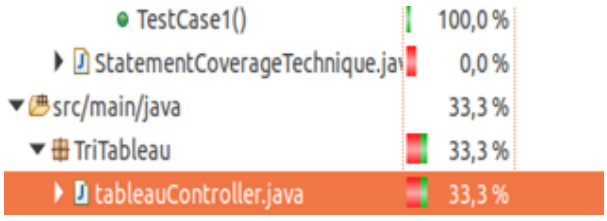
These metrics are related, but distinct.

We will be testing our Bubble sort algorithm using 4 test cases and each one of them will get us a function Coverage and a Branch Coverage percentage.

Test case 1: empty array

```
@Test
public void TestCase1() {
    table1 = new int[0];
    this.Controller.tri_bulle(table1);
    this.Controller.displayTab(table1);
//  assert(estTrie(table1));
 }
```

We can use the coverage tool Emma [12] to determine how much of our code is executed. After running the coverage tool, we get a coverage report displaying coverage metrics. We can see that while our Function Coverage is around 100%, our Branch Coverage is only 33,3%.

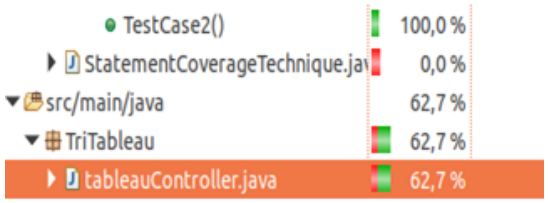


This is because when we run our script, the loops statement and if statement is not executed. If we want to get better coverage, we could simply put a one value array, essentially another test, to make sure that more branches of the loops are used.

Test case 2: Array with one value

```
@Test
public void TestCase2() {
    table2 = new int[1];
    table2[0] = 1;
    this.Controller.tri_bulle(table2);
    this.Controller.displayTab(table2);
    assert(estTrie(table2));
}
```

A second run of our coverage tool will now show that 62,7% of the source code is covered thanks to the new table which means that one of the loops was used.

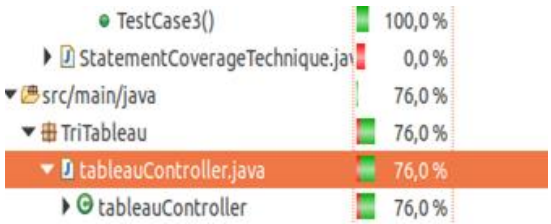


Test case 3: two-values array

A third test with a two-values array will improve the coverage percentage but it will be an already sorted array.

```
@Test
public void TestCase3() {
    table3 = new int[2];
    table3[0] = 1 ;
    table3[1] = 2 ;
    this.Controller.tri_bulle(table3);
    this.Controller.displayTab(table3);
}
```

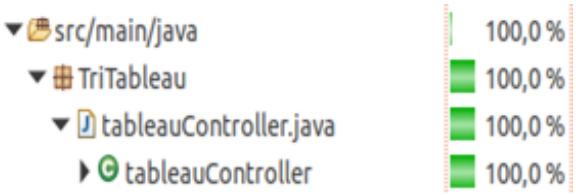
This time we will go through the two loops but without getting into the if statement. This explains the 76% coverage percentage.

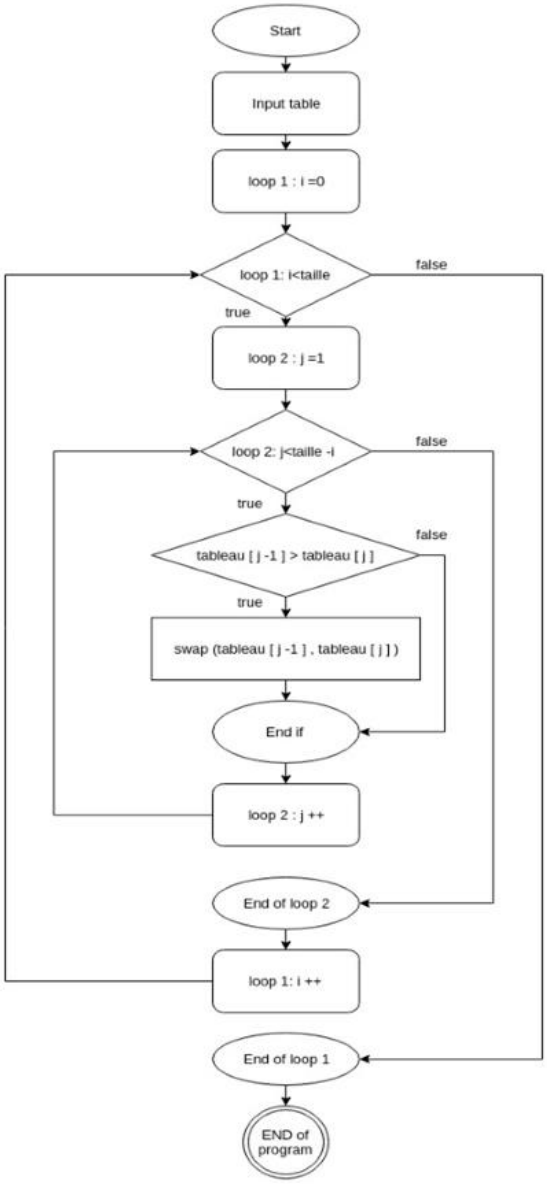


Test case 4: Completely Unsorted array

And the final test will allow us to go to the maximum 100% coverage. A completely unsorted array.

```
@Test
public void TestCase4() {
     table4 = new int[2];
     table4[0] = 2 ;
     table4[1] = 1 ;
     this.Controller.tri_bulle(table4);
     this.Controller.displayTab(table4);
}
```





The last test case is 100% efficient in terms of code coverage. This is an ideal case that proves that with one adequate test case, we can go through the integrality of the code. This technique is not resource consuming yet efficient.

The code coverage technique provides us with a short test case that covers the code entirely. However, is it efficient in terms of fault coverage?

We will do a mutation testing on our last test case in order to test this technique in terms of fault coverage:

```
@Before
public void SetUp() {
     tableauxATrier  = new ArrayList<int[]>();
     this.Controller = new tableauController();
     int entier;

     for(int j=0;j<10000;j++) {
        int val1 = ThreadLocalRandom.current().nextInt(-1000,1000) ;
        int val2 = ThreadLocalRandom.current().nextInt(-1000,1000) ;
         int tableau [] = new int[2] ;
         tableau[0]=val1;
         tableau[1]=val2;
         tableauxATrier.add(tableau);
     }
}

@Test
public void SortRandomNumbers() throws InterruptedException, ExecutionException {
    for(int[] T : this.tableauxATrier) {
        this.Controller.tri_bulle(T);
        assertTrue(isSorted(T));
    }
}
```

First, we generate mutants using Pitest and test them on our unsorted array.

```
================================================================================
- Timings
================================================================================
> scan classpath : < 1 second
> coverage and dependency analysis : < 1 second
> build mutation tests : < 1 second
> run mutation analysis : < 1 second
--------------------------------------------------------------------------------
> Total  : 1 seconds
--------------------------------------------------------------------------------
================================================================================
- Statistics
================================================================================
>> Generated 12 mutations Killed 10 (83%)
>> Ran 12 tests (1 tests per mutation)
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  3.023 s
[INFO] Finished at: 2019-06-01T16:50:55+02:00
[INFO] ------------------------------------------------------------------------
```

Pit Test Coverage Report

Package Summary

TriTableau

Number of Classes	Line Coverage		Mutation Coverage	
1	100%	11/11	83%	10/12

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
tableauController.java	100%	11/11	83%	10/12

In our example, 2 out of the 12 mutants have survived. We examined the codes of the undetected mutants and we found out that those are equivalent mutants.

One in which we made it as if we can swap two equal numbers and the other “i” the first counter can get to the limit of the array so “j” will go from 1 to 0 by +1 step which is impossible so the second loop will be bypassed and the program is still working. So, the true Coverage percentage is: N° killed mutants / (N° mutant – N° equivalence) = 100%.
The fault coverage is 100%. All the mutants are killed, and the test case generated by the white box code coverage technique is efficient.

In grey box testing, we recommend using this technique by finding an adequate test suite in order to cover the maximum of the code provided.

Random Black box testing:

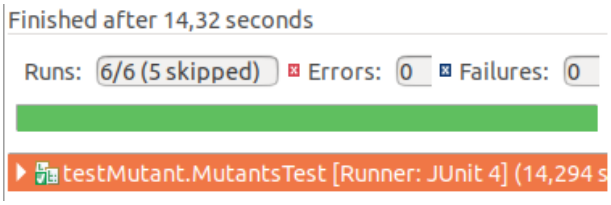
We then continue our tests using black box random testing by generating a test suite with 10000 arrays filled with 1000 random numbers.
We start by executing the test suite on the initial program.

```
@Before
public void SetUp() {
     tableauxATrier  = new ArrayList<int[]>();
     this.Controller = new tableauController();
     int entier;

     for(int j=0;j<10000;j++) {
         limit = ThreadLocalRandom.current().nextInt(0,1000) ;
         int tableau [] = new int[limit] ;
         for(int i = 0;i < limit;i++) {
             entier = ThreadLocalRandom.current().nextInt(0,limit);
             tableau[i] = entier;
         }
         tableauxATrier.add(tableau);
     }
}

@Test
public void SortRandomNumbers() throws InterruptedException, ExecutionException {
    for(int[] T : this.tableauxATrier) {
        this.Controller.tri_bulle(T);
        assertTrue(isSorted(T));
    }
}
```

The function isSorted is a verification function that checks if the array is sorted or not.



After running the black box test suite on our sorting program, all the arrays generated will be sorted. The bubble sorting program is working correctly on our test suite.

5. Performance:

We want to test the fault coverage of the black box testing technique and we will use mutation testing in order to do so. On the other hand, we used the same mutants as for the white box testing technique. The objective is to determine the fault coverage of our test suite.

```
================================================================================
- Mutators
================================================================================
> org.pitest.mutationtest.engine.gregor.mutators.ConditionalsBoundaryMutator
>> Generated 3 Killed 1 (33%)
> KILLED 1 SURVIVED 2 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.IncrementsMutator
>> Generated 2 Killed 2 (100%)
> KILLED 2 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.MathMutator
>> Generated 4 Killed 4 (100%)
> KILLED 4 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.NegateConditionalsMutator
>> Generated 3 Killed 3 (100%)
> KILLED 3 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
================================================================================
- Timings
================================================================================
> scan classpath : < 1 second
> coverage and dependency analysis : 5 seconds
> build mutation tests : < 1 second
> run mutation analysis : 12 seconds
--------------------------------------------------------------------------------
> Total  : 18 seconds
--------------------------------------------------------------------------------
================================================================================
- Timings
================================================================================
> scan classpath : < 1 second
> coverage and dependency analysis : < 1 second
> build mutation tests : < 1 second
> run mutation analysis : < 1 second
--------------------------------------------------------------------------------
> Total  : 1 seconds
--------------------------------------------------------------------------------
================================================================================
- Statistics
================================================================================
>> Generated 12 mutations Killed 10 (83%)
>> Ran 12 tests (1 tests per mutation)
```

Pit Test Coverage Report

Package Summary

TriTableau

Number of Classes	Line Coverage		Mutation Coverage	
1	100%	11/11	83%	10/12

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
tableauController.java	100%	11/11	83%	10/12

We obtain the same results. 2 equivalents mutants remain undetected and all the other mutants are killed. But with a much faster time and less test cases. Due to our knowledge of the system (white-box), we were able to optimize the test suite in term of time and memory; as we were able to minimize the test suites to simply random two-values array.

For the bubble sorting code, the Random black box testing technique is efficient in terms of fault coverage. The test suite generated randomly has a 100% coverage percentage. Random black box testing technique is more time and resource consuming than the techniques that we tested before and therefore we recommend using it after those techniques.

- Both Black box random testing and white box code coverage testing techniques detect the mutants correctly. They are efficient in terms of fault coverage. However, the black box test suite is formed of 10000 arrays with a 1000 element each. while the white box test suite is only formed of a 1 array with 2 elements.

- The white box testing technique in our example seems to be significantly less time and resource consuming. However, it supposes that we can find a test suite that covers 100% of the code which is rarely possible if we do not have access to the entire code. In that case, random black box testing has more chance of detecting errors.

6. Conclusion

6.1. An approach to grey box technique

White Box testing and black box testing are not antipodes. It is always better to combine these two approaches in a way for building better test suites. So that the advantages of both paradigms are combined, and the disadvantages are erased. In a nutshell and based on our experimental results, the processing of testing can be performed in the following steps: First, we recommend using all the provided information about the software by examining the requirements and the parts of the code that are provided. This is white box static testing and it permits the early detection of errors. The inputs and outputs of the code should then be tested using black box equivalence testing technique. The next step is to analyze the parts of the code provided and find a test suite that covers as much of the code as possible: white box code coverage testing technique. The last step is to apply black box random testing on the software and analyze the possible defects.

All these steps depend on the amount of information that we have on our software and on the resources, we must test it. If our software is heavy to test dynamically then we are limited by our resources and we would prefer to use as much static testing techniques as we can and reduce the size of our test suites. The white box dynamic testing techniques such as code coverage testing are in that case more likely to be used.

If we have a very limited access to the source code of the software, then the black box testing is preconized. Random black box testing techniques are usually used in the very grey box or black box cases.

In conclusion, the steps to adopt when testing a software are strongly linked with the greyness of the box (the amount of information and code provided during the tests) and the resources that can be used during the tests.

7. references

[1]https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[2] “Software testing is the process of applying inputs, observing output responses and drawing conclusions.” Dr Natalia Kushik

[3] https://www.researchgate.net/publication/301351247_The_Classification_of_Software_Testing_Techniques

[4] https://www.guru99.com/testing-review.html

[5]https://www.guru99.com/dynamic-testing.ht ml

[6] Different Approaches to black box testing technique for finding errors by Moh. Ehmer Khan

[7]Essays, UK. (November 2018). Black Box And White Box Comparison Computer Science Essay. Retrieved from https://www.ukessays.com/essays/computer-science/black-box-and-white-box-comparison-computer-science-essay.php?vref=1

[8]https://www.academia.edu/27884905/A_White_Box_Testing_Technique_in_Software_Testing_Basis_Path_Testing

[9] https://saucelabs.com/blog/the-challenges-and-benefits-of-model-based-testing

[10]https://fr.slideshare.net/KrunalParmar13/mutation-testing-and-mujava

[12] https://www.eclemma.org/