International Journal of Enhanced Research in Science, Technology & Engineering
ISSN: 2319-7463, Vol. 5 Issue 4, April-2016

Automated Code-Checker for Python Programs

Niharika Jayanthi[1], Lipisha Chaudhary[2], Priti Govalkar[3], Kavita Wale[4]

[1,2,3,4] Department of Computer Engineering, Fr. Conceicao Rodrigues Institute of Technology, Vashi, Navi Mumbai

Abstract: The most common setback faced by beginners in programming is the need to check whether the code will run robustly, given any form of input. Even software written by professionals needs to undergo stress testing to ensure its smooth field performance. This project aims to provide a mechanism to detect the logical errorsin a Python program and notify the user about them. The user gives code written in Python as an input. It is initially checked for syntax errors. Code Checker will then determine the inputs being taken by the code submitted by ther user. Once this information is gathered, the code is checked again for common logical errors– deadlocks, division by zero, infinite loop conditions, uninitialised variable references and these errors are presented to the user in a report. This will help a beginner programmer to write efficient Python programs by learning in an iterative fashion.

Keywords: **Automated, dynamic analysis, logical errors, notification, test cases**

I. INTRODUCTION

Software testing is the activity of checking a software system for errors. Any application needs to be checked for whether it satisfies the user requirements. Testing is an important phase in software development cycle as any software defects (bugs) can plague the developer after deployment. Unlike physical objects, defects in software systems occur in design. Design of any application must account for all possible types of input from user, else the system may malfunction.

Testing is broadly deployed in every stage of the software development cycle. Once a bug is found, the code can be changed in order to accommodate that design failure. However, this change can cause new bugs to crop up. This is known as Pesticide Paradox- Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual.[1]

There are several levels of software testing which are unit testing, component testing, integration testing, component integration testing, system integration testing, system testing, acceptance testing, alpha testing and beta testing. Integration testing can be performed in one of the four types which are Big Bang integration testing, Top Down testing, Bottom Up testing and Functional Incremental testing. Unit testing of code can either be performed before the code is executed or during the execution of the program. Following are two types of analysis to check for bugs in any given code-

A. Static Analysis

In static analysis, the software program is evaluated for errors or bugs without actually executing the program. This is, therefore, also known as Source Code Analysis. The process provides an understanding of the code structure, and can help to ensure that the code adheres to industry standards. Automated tools can assist programmers and developers in carrying out static analysis. Static analysis toolsare generally used by developers as part of the development and component testing process. The key aspect is that the code (or other artefact) is not executed or run but the tool itself is executed, and the source code we are interested in is the input data to the tool.There are various techniques available to analyze a source code statically.

a) Data Flow Analysis

Data flow analysis is used to collect run-time (dynamic) information about data in software while it is in a static state [2].

b) Control Flow Graphs

An abstract graph representation of software by use of nodes that represent basic blocks. A node in a graph represents a block; directed edges are used to represent jumps (paths) from one block to another. If a node only has an exit edge, this is known as an ‘entry’block, if a node onlyhas an entry edge, this is known as an ‘exit’ block[2].

c) Taint Analysis

Taint Analysis attempts to identify variables that have been 'tainted' with user controllable input and traces them to possible vulnerable functions also known as a 'sink'. If the tainted variable gets passed to a sink without first being sanitized it is flagged as a vulnerability.

d) Lexical Analysis

Lexical Analysis converts source code syntax into ‘tokens’ of information in an attempt to abstract the source code and make it easier to manipulate[2]. The principal advantage of static analysis is the fact that it can reveal errors that do not manifest themselves until a disaster occurs weeks, months or years after release. Other than software code, static analysis can also be carried out on things like, static analysis of requirements or static analysis of websites (for example, to assess for proper use of accessibility tags or the following of HTML standards).

B. Dynamic Analysis

Under dynamic analysis, errors are detected by executing the source code to be tested. To ensure efficient results, the program must be executed enough times to produce interesting results. Dynamic analysis typically involves instrumenting a program to examine or record certain aspects of its run-time state. This instrumentation can be tuned to collect precisely the information needed to address a particular problem [3]. In dynamic analysis, the correlation between any changes in input and the corresponding output can be easily determined as they are linked by program execution. So, while the attention of analysis in static analysis was on the code, in dynamic analysis it is on the input provided.

Dynamic analysis is flexible with respect to what to scan for. It can be performed for any application, even if there is no access to the source code of the application. Any false negatives given during static analysis can be identified during dynamic analysis. Code Checker application performs unit testing of Python 2.7x programs using dynamic analysis. It takes the program to be tested from the user as input and presents the results on the graphical user interface.

II. PROPOSED SYSTEM

Code Checker using Automated Theorem Prover (ATP) is a system that will take name of the project, lines of code (LOC), the code itself and many other inputs from the user. It will assume that the given source program is free of any syntax errors. These inputs are transferred to the back-end process. This process uses a testing tool to check for errors in the code. Depending on the lines of code in the user’s source program, the code may have to be divided into parts. The way a code is parted will depend on whether all the necessary instructions to do testing are present in that part. One can simply ensure efficient testing by dividing code as per its modules. In this project, we shall focus on testing modules of a code. This is known as Unit Testing.

Once the user’s source program has been effectively divided into parts (assuming LOC has a high value), each part will now be verified with a testing tool. This testing tool will check for various errors such as possible infinite loops, divide by

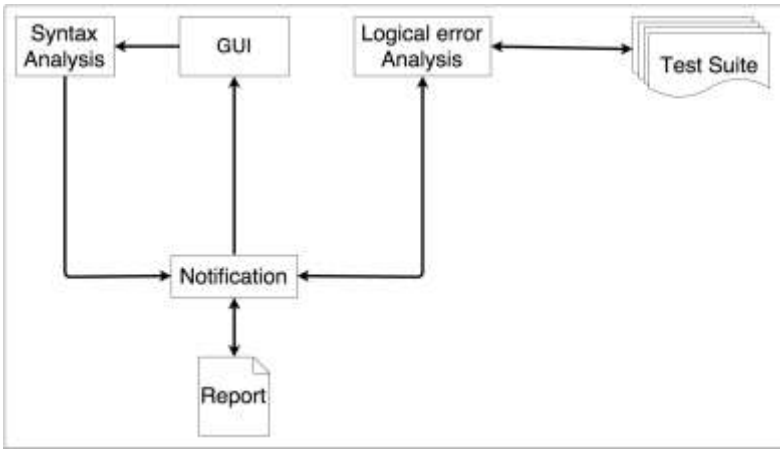

Fig. (1) Block diagram of the proposed system

zero error, unreachable code blocks, infinite recursion and other such logical errors. If any error is found to be present in a source program part, that error details will be stored in an error report file.

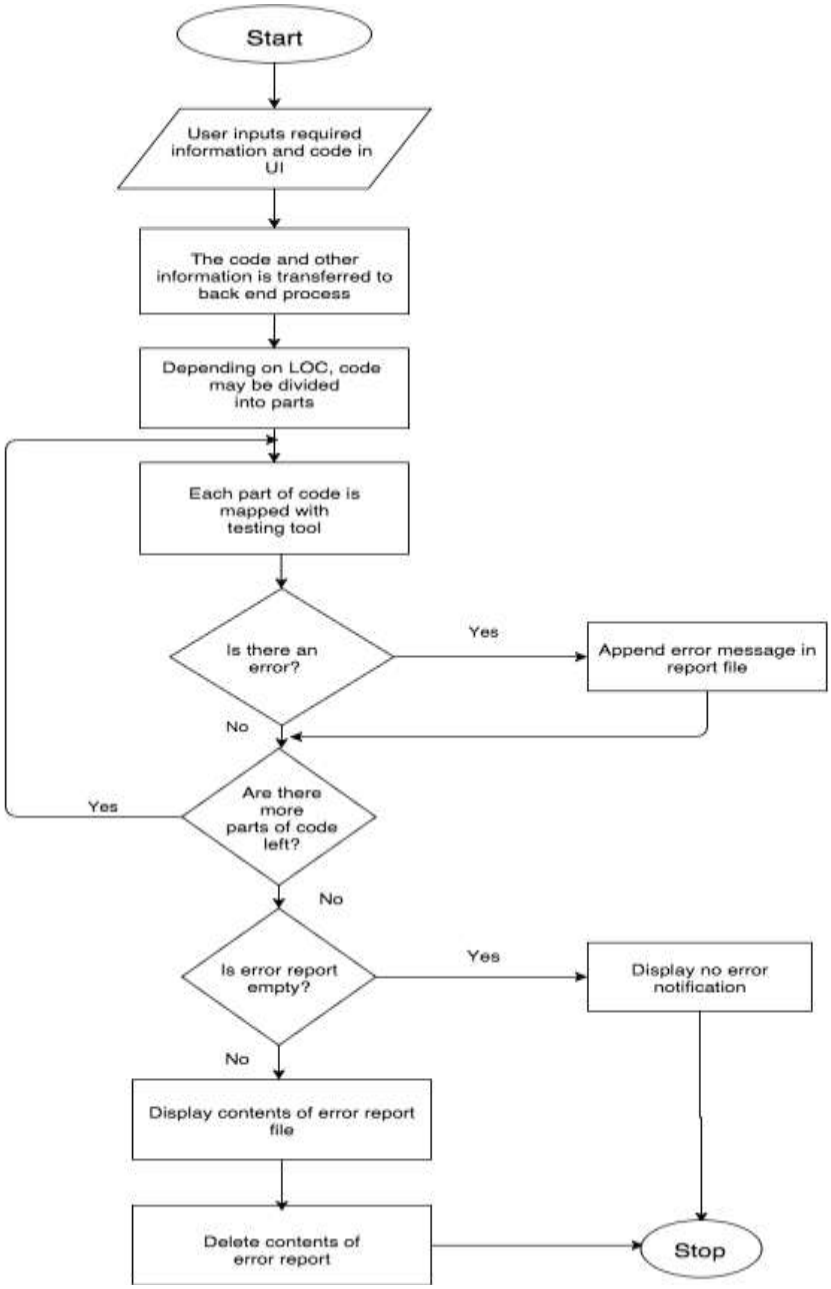

Fig. (2) Flow of control in Code Checker application

Once results have been displayed, the input source program will be deleted and the contents of the error report will be cleared. The details provided about the error will include the line number of the instruction where error was found to be present, the type of error encountered, the test cases with which the error has occurred and a brief notification message for that error type. After all the source program parts have been successfully tested, we will have to display the results of testing. We first check the report for any errors present. If there are no errors, then we shall display a "No errors found! Good job!" notification. However, if the report is not empty, then the contents of the report will be displayed to the user through the user interface.

III. TESTING

The main purpose of software testing is to find bugs and defects, perform verification and validation and in overall improve the quality of product. Ideally, testing should be efficient, quick and cheap to run and maintain, without worrying if the testing is manual or automated. Manual testing is performed by a human sitting in front of a computer carefully executing the test steps. Automated Testing means using automation tool to execute your test case suite. The automation software can also enter test data into the System Under Test, compare expected and actual results and generate detailed test reports. Using a test automation tool it's possible to record this test suite and re-play it as required. Once the test suite is automated, no human intervention is required. Thus the goal of Automation is to reduce number of test cases to be run manually and not eliminate manual testing all together.

Need for Automated Testing:

Automated testing is important due to following reasons:

a. Manual Testing of all work flows, all fields , all negative scenarios is time and cost consuming
b. It is difficult to test for multi lingual sites manually
c. Automation increases speed of test execution
d. Automation helps increase Test Coverage
e. Manual Testing can become boring and hence error prone.

The Benefits of Automation Testing:

Automated testing offers a revolutionary alternative to manual testing methods. This enables the automation of testing tasks and provides broad testing coverage and functionality across the entire software testing lifecycle.

If implemented correctly, automated testing offers the following advantages over manual testing:

a. Reduces costs
b. Optimization of speed
c. Better usage of available time and resources
d. Easily repeatable
e. Eliminates human errors
f. Improves quality.
g. New test cases are generated continuously and can be added to existing automation in parallel to the development of the software itself.

A well-known fact is that the later a bug is found, the more expensive it is to fix. The utilization of test automation can help find bugs earlier.

Automated testing metrics:

The measuring of different aspects of software testing can aid in making assessments if the set goals are being or have been met. A metric is a standard of some meaningful measurement that can be used to analyze past, present and future performance.

The three main categories of software testing metrics are:

a. Coverage
b. Progress
c. Quality

A good automated testing metric has the following characteristics:

a. Objective
b. Measurable
c. Meaningful
d. Relevant information is easily collected
e. Helpful to identify areas of test automation improvement
f. Simple.

Approaches for automated Testing:

Automated tests can be categorized into five main approaches:

a. Record and playback
b. Data-driven testing
c. Keyword-driven testing
d. Test scripting
e. Model-driven testing

VI. TEST CASES

A test case, in software engineering, is a set of conditions under which a tester will determine whether an application, software system or one of its features is working as it was originally established for it to do. The mechanism for determining whether a software program or system has passed or failed such a test is known as a test oracle. In some settings, an oracle could be a requirement or use case, while in others it could be a heuristic. It may take many test cases to determine that a software program or system is considered sufficiently scrutinized to be released. Test cases are often referred to as test scripts, particularly when written - when they are usually collected into test suites[4].

In order to fully test that all the requirements of an application are met, there must be at least two test cases for each requirement: one positive test and one negative test. A negative test case is a test that is designed to test erroneous input and is expected to fail. For example, when testing a calculator application, a test case where a number is divided with zero. The opposite is a positive test case that is expected to pass normally. The usage of negative test cases is referred to as destructive testing. Similarly, non-destructive testing refers to the usage of positive test cases. A characteristic of good software is to gracefully handle both positive and negative tests.

The four attributes of a good test case are:

a. To be effective to detect defects,
b. To test more than just one thing,
c. To be economical to perform, analyze and debug and
d. To be flexible to maintain as software evolves.

For checking of the actual code submitted by the user, test cases are used. While running these test cases the Code Checker application will detect error, if any, by executing each of the test cases present in the test suite. The result of which will be stored in a test log which the user can refer later for making any modification in their code.

A) Test Suites

In software development, a test suite, less commonly known as a 'validation suite', is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviors. A test suite often contains detailed instructions or goals for each collection of test cases and information on the system configuration to be used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests[5].

B) Test Oracle

In computing, software testers and software engineers can use an oracle as a mechanism for determining whether a test has passed or failed. The use of oracles involves comparing the output(s) of the system under test, for a given test-case input, to the output(s) that the oracle determines that product should have [6].

In manual testing the person performing the test is typically the test oracle. In automated software testing the test oracle is implemented with software. There are different types of oracles based on their characteristics. The five oracle types are[7]-

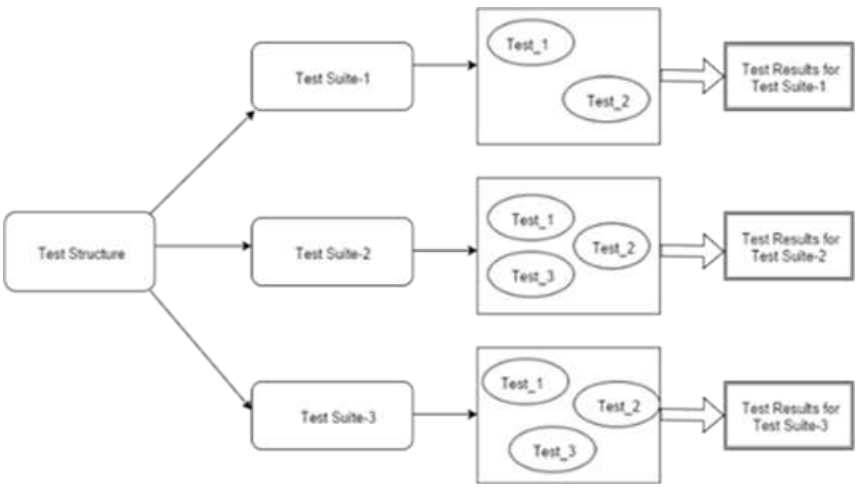

Fig. (3) Test Suites

a) A True Oracle

A true oracle independently re-implements the same functionality as the Software/System Under Test. All of the input is then given to both SUT and the oracle compared against each other.

b) Stochastic

A stochastic oracle randomly selects a sample for verification. The randomly chosen values are easy to implement, but systematic and specific errors will be easily missed.

c) Heuristic

Provides approximate results or exact results for a set of a few test inputs.

d) Sampling

The sampling based oracle uses a selected set of values. The difference between a stochastic oracle and sampling oracle is that the sampling oracle does not choose values randomly. Instead the values which are often chosen are minimum, maximum boundary values and midpoint values.

e) Consistent

A consistent oracle uses previously saved test results for decision making.

C) Test Log

When test cases are executed on a code some errors or failures occur. These errors or failures are recorded in a document called test log. In this application when the user inputs his/her code, relevant test cases from the test suites are run on the input code some output is generated based on the values obtained from the test cases. This output can be an error or failure message, which can be stored in the test logs. The test logs stores the output of the code and also some other important information about the code, the execution time, date etc [7].

VI. IMPLEMENTATION

A) Graphical User Interface

The GUI of Code Checker application is very simple to use and navigate. The basic GUI consists of means for taking the file as an input in an effortless fashion. This file is the one which the application takes as an input and performs unit testing on it to check for logical errors. There are two ways in which the file can be inputted one is by the conventional way of browsing the files and the second way is by drag and drop feature. Due to this features the interface is clean and fuss-free.

Fig. (4) Browse files and Drag & Drop feature

After the user inputs the file which needs to be checked, the application then starts the processing (checking) the source code file. While the checking of the file takes place a progress bar is displayed, to mark the time remaining till the checking of the file is complete.

Fig. (5) Progress Bar

Once the checking is done the progress bar displays a message "Done" with a message dialog box notifying the user that checking of their file is finished successfully also that the error report is also generated.

Fig. (6) Notification Dialog Box

B) Logical Error Analysis

i) Deadlocks

A deadlock refers to a state where two or more processes form a cycle while waiting for each other to release a resource. Self-deadlocks or recursive deadlocks are some of the common types of deadlocks that occur when a process tries to obtain a resource it is already holding. The solution for this kind of deadlock is to avoid calling functions outside the module when you don't know whether they will call back into the module without re-establishing invariants and dropping all module locks before making the call. Of course, after the call completes and the locks are reacquired, the state must be verified to be sure the intended operation is still valid.[8]

The next common type of deadlock is when process A waits for resource from process B whereas process B waits for resource from process A. These type of deadlocks can be analysed using both static analysis and dynamic analysis. By generating models such as Wait-For graphs or Resource Allocation graphs, it is easy to detect deadlock cycles and recover from them.

Code Checker application uses static program analysis to detect deadlocks in Python programs given as input by the user. This analysis works for both Python 2.x and Python 3.x versions. So it can be ubiquitously used as per the user's preference.

Detection of possible deadlocks in the program is done by checking for call cycles. A call cycle refers to the situation when function A calls function B, function B calls function C and so on until function N and function N calls function A, thus, creating a cycle. The first step is to divide the input Python program into several units where each unit comprises of the code in one function and one unit consists of the main code outside of any function. The main code is analysed first to check for the function calls present in it. Once a function call is encountered, analysis of that unit is paused and the unit containing the called function gets analysed next. If any function calls are encountered in this second unit, then analysis of this unit gets paused and the called function's unit is analysed. This

takes place until the main unit is successfully analysed completely. If any function call is encountered whose unit has been paused in analysis, then we have a new deadlock possibility. In the end, a report is generated which the user can access. This will contain the date and time of analysis, name of the input program that was analysed and whether any possible deadlocks have been found. If found, then it is represented via a Wait-For graph.

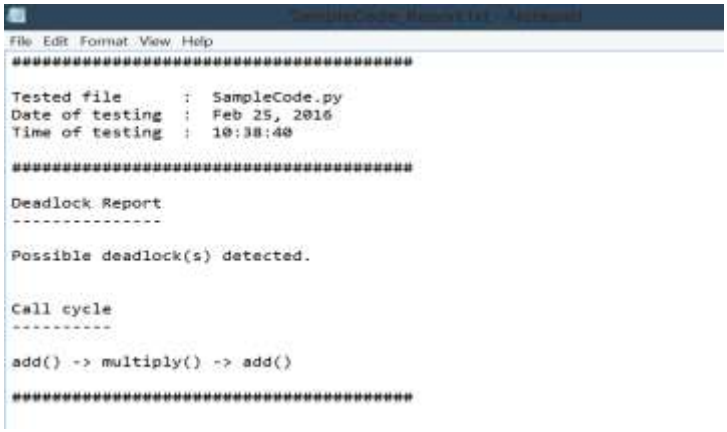

Fig. (7) Report generated after deadlock analysis

ii) Unit Testing

For unit testing breaking of program into units is essential. Therefore the Code Checker first breaks the user submitted code into functions where each function is considered as a single unit. Each unit is then independently checked for logical errors by the application. This testing is done by automatically generating the test cases that run on each unit. The system then gives the output to the user by specifying the number of tests that ran on the code and also the time required to run the tests. If the user submitted code is logically correct than the system will notify the user with an "OK" message else, will display the logical errors that encountered in the program. Consider an example of Arithmetic program which the user submits to the Code Checker through GUI. The code consist of functions like add, subtract, multiple and division. The system first breaks the entire code into independent functions where each function is a unit. Currently test cases are provided manually. This test cases run on each unit. The system output shows the total number of tests ran and the time taken to run them. If the code is logically correct then an "OK" message is displayed.

Fig. (8) Message displayed for successful verification

Suppose in test cases of division, a number is divided by zero then the system will notify the user by displaying logical error i.e. "Division by Zero Error". Thus, the system helps the user who could a novice to improve his/her code by displaying the warning notifications on the interface and allowing him/her to modify their code accordingly.

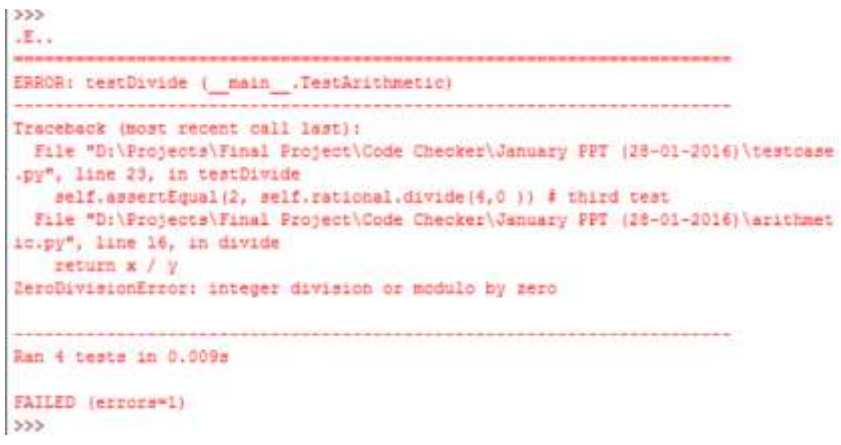

Fig. (9) "Division by zero" error message

C) Syntax Analysis

Since dynamic analysis involves execution of the program under test, it is imperative that the program should be syntactically correct. If the program contains syntax errors, testing will fail as program will not get executed. However, since Code Checker for Python programs is an application that is suitable for beginners, it cannot be assumed that input program will not contain any syntax errors. In such a case, syntax analysis of the input Python program must be performed before proceeding to logical analysis. Code Checker application will check for different types of syntax errors such as indentation errors and missing colons.

```
x = int(raw_input())
s = 0
for i in range(x)
    s += i
    print i

print "Sum upto", x, "whole numbers is", s
```

Fig. (10) Python code with syntax error(missing colon)

```
>>>
Compiling syn.py ...
  File "syn.py", line 3
    for i in range(x)
                    ^
SyntaxError: invalid syntax
```

Fig. (11) Syntax Error Notification

CONCLUSION

Testing and debugging the program have always been an inseparable part of the development process, but automated softwares are less used to address the challenge of testing complex programs. In this paper we have presented an overview of automated program verification, which can be used to check the correctness of the code using a combination of static and dynamic program analysis. We have also reviewed the proposed system, Code Checker, in order to verify Python code that is given as an input by the user. The system thus checks for syntax eroors and logical errors by automatically running the test cases on it. Thus, the system helps the user or beginner to improve their code by displaying the warning notifications on the interface. Also, automating this process reduces the amount of time developers spend performing tests and enables more exhaustive testing by finding the problems earlier.

REFERENCES

[1]. https://users.ece.cmu.edu/~koopman/des_s99/sw_testing/
[2]. https://www.owasp.org/index.php/Static_Code_Analysis# Data_Flow_Analysis
[3]. http://research.microsoft.com/en-us/us/um/tball/people/fse-concept.pdf
[4]. https://en.wikipedia.org/wiki/Test_case
[5]. https://en.wikipedia.org/wiki/Test_suite
[6]. https://en.wikipedia.org/wiki/Oracle_(software_testing)
[7]. “Designing and automating dynamic testing of software Nightly builds” – University of Oulu
[8]. https://docs.oracle.com/cd/E19455-01/8065257 /6je9h0347/index.html