International Journal of Engineering Research & Management Technology ISSN: 2348-4039

November-2016 Volume 3, Issue-6

Email: editor@ijermt.org www.ijermt.org

SURVEY OF SOFTWARE TESTING TECHNIQUES

Ruby Singh
Faculty of Information Technology
SRM University
India

Chiranjit Dutta
Faculty of Information Technology
SRM University
India

Ranjeet Singh
Faculty of Information Technology
SRM University, India

ABSTRACT: In this paper main testing methods and techniques are shortly described. General classification is outlined: two testing methods – black box testing and white box testing, and their frequently used techniques:

- ***Black Box techniques:*** Equivalent Partitioning, Boundary Value Analysis, Cause-Effect Graphing Techniques, and Comparison Testing;
- ***White Box techniques:*** Basis Path Testing, Loop Testing, and Control Structure Testing. Also, the classification of the IEEE Computer Society is illustrated.

1. DEFINITION AND THE GOAL OF TESTING

PROCESS of creating a program consists of the following phases (see [8]): 1. defining a problem; 2. designing a program; 3. building a program; 4. analyzing performances of a program, and 5. final arranging of a product. According to this classification, software testing is a component of the third phase, and means checking if a program for specified inputs gives correctly and expected results.

Software testing (Figure 1) is an important component of software quality assurance, and many software organizations are spending up to 40% of their resources on testing. For life-critical software (e.g., flight control) testing can be highly expensive. Because of that, many studies about **risk analysis** have been made. This term means the probability that a software project will experience undesirable events, such as schedule delays, cost overruns, or outright cancellation (see [9]), and more about this in [10].

There are a many definitions of **software testing**, but one can shortly define that as: [1]

A process of executing a program with the goal of finding errors (see [3]). So, testing means that one inspects behavior of a program on a finite set of **test cases** (a set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement, see [11]) for which valued inputs always exist. In practice, the whole set of test cases is considered as infinite, therefore theoretically there are too many test cases even for the simplest programs. In this case, testing could require months and months to execute. So, how to select the most proper set of test cases? In practice, various techniques are used for that, and some of them are correlated with risk analysis, while others with test engineering expertise.

Testing is an activity performed for evaluating software quality and for improving it. Hence, the goal of testing is systematical detection of different classes of errors (**error** can be defined as a human action that produces an incorrect result, see [12]) in a minimum amount of time and with a minimum amount of effort. We distinguish (see [2]):

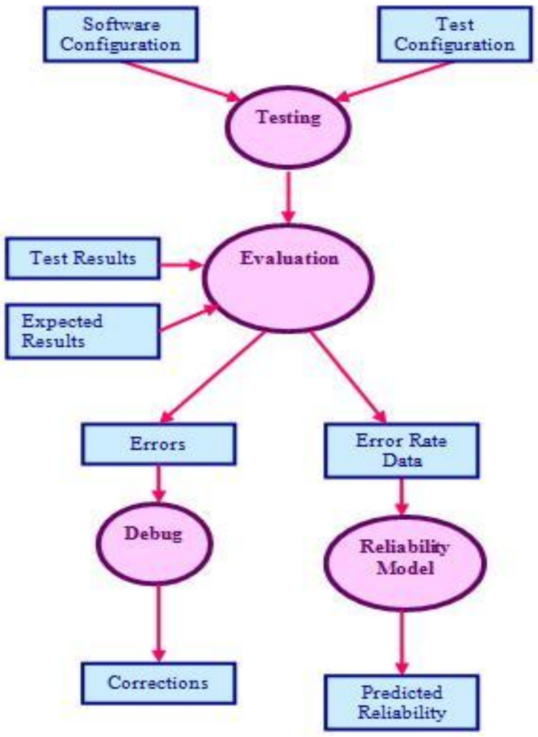


Figure 1: Test Information Flow

- ***Good test case:*** have a good chance of finding an yet undiscovered error; and
- ***Successful test cases:*** uncovers a new error.

Anyway, a *good test case* is one which: validate whether code implementation follows intended design, to validate implemented security functionality, and to uncover exploitable vulnerabilities (see [15]).

***Black box testing** is testing software based on output requirements and without any*
- Has a high probability of finding an *knowledge of the internal structure or coding in* error; Is not redundant;
- Should be "best of breed";
- Should not be too simple or too complex.

2. TESTING METHODS

Test cases are developed using various test techniques to achieve more effective testing. By this, software completeness is provided and conditions of testing which get the greatest probability of finding errors are chosen. So, testers do not guess which test cases to chose, and test techniques enable them to design testing conditions in a systematic way. Also, if one combines all sorts of existing test techniques, one will obtain better results rather if one uses just one test technique.

Software can be tested in two ways, in another words, one can distinguish two different methods:

1. Black box testing, and
2. White box testing.

White box testing is highly effective in detecting and resolving problems, because bugs (***bug*** *or* ***fault*** *is a manifestation of an error in a software*, see [12]) can often be found before they cause trouble. We can shortly define this method as *testing software with the knowledge of the internal structure and coding inside the program* (see [13]). White box testing is also called white box analysis, clear box testing or clear box analysis. It is a strategy for software ***debugging*** (*it is the process of locating and fixing bugs in computer program code or the engineering of a hardware device*, see [14]) in which the tester has excellent knowledge of how the program components interact. This method can be used for Web services applications, and is rarely practical for debugging in large systems and networks (see [14]). Besides, in [15], white box testing is considered as a ***security testing*** (*the process to determine that an information system protects data and maintains functionality as intended*, see [6]) method that can be used to *the program* (see [16]). In another words, a black box is any device whose workings are not understood by or accessible to its user. For example, in telecommunications, it is a resistor connected to a phone line that makes it impossible for the telephone company's equipment to detect when a call has been answered. In data mining, a black box is an algorithm that doesn't provide an explanation of how it works. In film–making, a black box is a dedicated hardware device: equipment that is specifically used for a particular function, but in the financial world, it is a computerized trading system that doesn't make its rules easily available.

In recent years, the third testing method has been also considered – ***gray box testing***. *It is defined as testing software while already having some knowledge of its underlying code or logic*

(see [17]) . It is based on the internal data structures and algorithms for designing the test cases more than black box testing but less than white box testing. This method is important when conducting integration testing between two modules of code written by two different developers, where only interfaces are exposed for test. Also, this method can include reverse engineering to determine boundary values. Gray box testing is non-intrusive and unbiased because it doesn't require that the tester have access to the source code.

The main characteristics and comparison between white box testing and black box testing are follows.

2.1. Black Box Testing Versus White Box Testing

***Black Box Testing*:**

- Performing the tests which exercise all functional requirements of a program;
- Finding the following errors:

1. Incorrect or missing functions;
2. Interface errors;
3. Errors in data structures or external database access;
4. Performance errors;
5. Initialization and termination errors.

❖ ***Advantages of this method:***

- The number of test cases are reduced to achieve reasonable testing;
- The test cases can show presence or absence of classes of errors.

***White Box Testing*:**

- Considering the internal logical arrangement of software;
- The test cases exercise certain sets of conditions and loops;

❖ ***Advantages of this method:***

- All independent paths in a module will be exercised at least once;
- All logical decisions will be exercised;
- All loops at their boundaries will be executed;
- Internal data structures will be exercised to maintain their validity.

3. GENERAL CLASSIFICATION OF TEST TECHNIQUES

In this paper, the most important test techniques are shortly described, as it is shown in Figure 2.

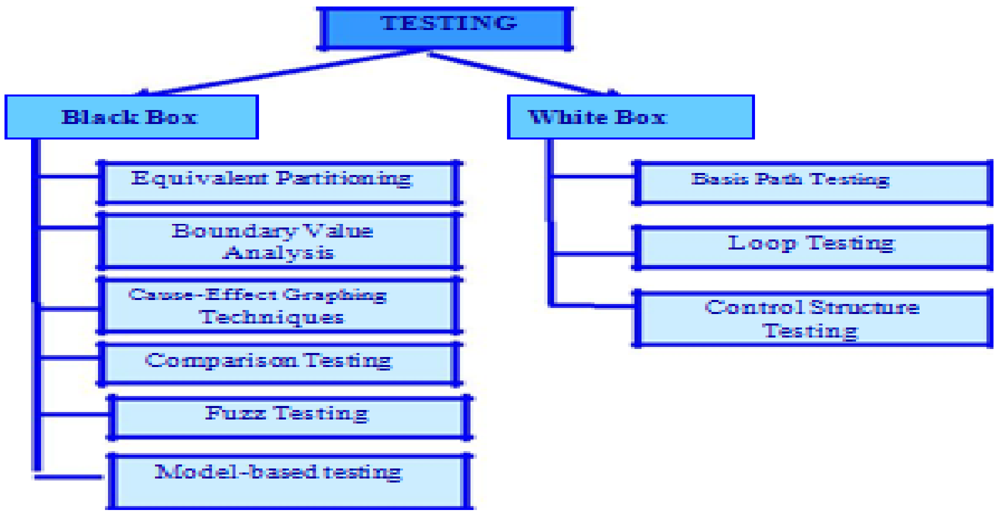


Figure 2: General Classification of Test Techniques

3.1. Equivalence Partitioning

Summary: *equivalence class*

This technique divides the input domain of a program onto equivalence classes.

Equivalence classes: set of valid or invalid states for input conditions, and can be defined in the following way:

1. An input condition specifies a range → one valid and two invalid equivalence classes are defined;
2. An input condition needs a specific value → one valid and two invalid equivalence classes are defined;
3. An input condition specifies a member of a set→ one valid and one invalid equivalence class are defined;
4. An input condition is Boolean→ one valid and one invalid equivalence class are defined.

Well, using this technique, one can get test cases which identify the classes of errors.

3.2. Boundary Value Analysis

Summary: *complement equivalence partitioning*

This technique is like the technique Equivalence Partitioning, except that for creating the test cases beside input domain use output domain. One can form the test cases in the following way:

1. An input condition specifies a range bounded by values a and→ test cases should be made with values just above and below a and b respectively.

2. An input condition specifies various values→ test cases should be produced to exercise the minimum and maximum numbers.
3. Rules 1 and 2 apply to output conditions;

If internal program data structures have prescribed boundaries, produce test cases to exercise that data structure at its boundary.

3.3. Cause-Effect Graphing Techniques

Summary: *translate*

One uses this technique when one wants to translate a policy or procedure specified in a natural language into software's language.

This technique means:

Input conditions and actions are listed for a module ⇒ an identifier is allocated for each one of them ⇒ cause-effect graph is created

⇒ this graph is changed into a decision table

⇒ The rules of this table are modified to test cases.

3.4. Comparison Testing

Summary: *independent versions of an application*

In situations when reliability of software is critical, redundant software is produced. In that case one uses this technique.

This technique means:

Software engineering teams produce independent versions of an application→ each version can be tested with the same test data → so the same output can be ensured.

Residual black box test techniques are executing on the separate versions.

3.5. Fuzz Testing

Summary: *random input*

Fuzz testing is often called fuzzing, robustness testing or negative testing. It is developed by Barton Miller at the University of Wisconsin in 1989. This technique feeds random input to application. The main characteristic of fuzz testing, according to the [26] are:

- the input is random;
- the reliability criteria: if the application crashes or hangs, the test is failed;
- fuzz testing can be automated to a high degree.

A tool called fuzz tester which indicates causes of founded vulnerability, works best for problems that can cause a program to crash such as buffer overflow, cross -site scripting, denial of service attacks, format bug and SQL injection. Fuzzing is less effective for spyware, some viruses, worms, Trojans, and keyloggers. However, fuzzers are most effective when are used together with extensive black box testing techniques.

3.6. Model-based testing

Model-based testing is automatic generation of efficient test procedures/vectors using models of system requirements and specified functionality (see [27]).

In this method, test cases are derived in whole or in part from a model that describes some aspects of the system under test. These test cases are known as the abstract test suite, and for their selection different techniques have been used:

- generation by theorem proving;
- generation by constraint logic programming;
- generation by model checking;
- generation by symbolic execution;
- generation by using an event-flow model;
- generation by using a Markov chains model.

Model-based testing has a lot of benefits (according [28]):

- forces detailed understanding of the system behavior;
- early bug detection;
- test suite grows with the product;
- manage the model instead of the cases;
- can generate endless tests;
- resistant to pesticide paradox;
- find crashing and non-crashing bugs;
- automation is cheaper and more effective;
- one implementation per model, then all cases free;
- gain automated exploratory testing;
- testers can address bigger test issues.

3.6. Basis Path Testing

Summary: *basis set, independent path, flow graph, cyclomatic complexity, graph matrix, link weight*

If one uses this technique, one can evaluate logical complexity of procedural design. After that, one can employ this measure for description basic set of execution paths.

For obtaining the basis set and for presentation control flow in the program, one uses ***flow graphs*** (Figure 3 and Figure 4). Main components of that graphs are:

- ***Node:*** it represents one or more procedural statements. Node which contains a condition is called *predicate node*.
- ***Edges between nodes:*** represent flow of control. Each node must be bounded by at least one edge, even if it does not contain any useful information.
  Region – an area bounded by nodes and edges

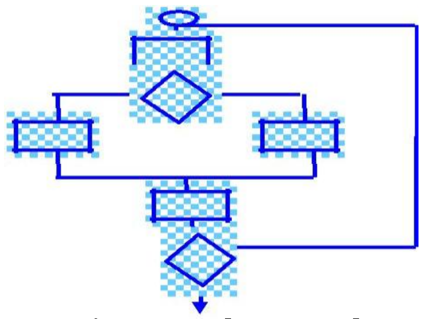


Figure 3: Flow Graph

Cyclomatic Complexity is software metric. The value evaluated for cyclomatic complexity defines the number of independent paths in the basis set of a program.

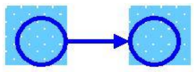
Independent path is any path through a program that introduces at least one new set of processing statements.

For the given graph G, cyclomatic complexity V(G) is equal to:

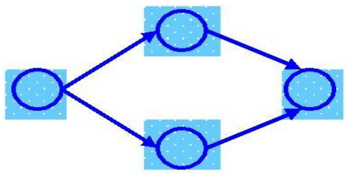
1. The number of regions in the flow graph;
2. **V(G) = E - N + 2**, where E is the number of edges, and N is the number of nodes;

V(G) = P + 1, where P is the number of predicate nodes.

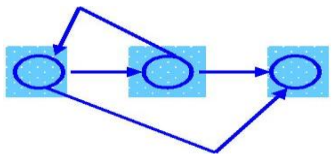
So, the core of this technique is: one draws the flow graph according to the design or code like the basis ⇒ one determines its cyclomatic complexity; cyclomatic complexity can be determined without a flow graph → in that case one computes the number of conditional statements in the code ⇒ after that, one determines a basis set of the linearly independent paths; the predicate nodes are useful when necessary paths must be determined ⇒ at the end, one prepares test cases by which each path in the basis set will be executed. Each test case will be executed and compared to the expected results.



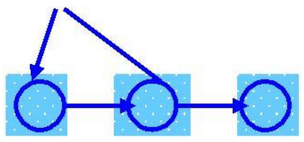
Sequence



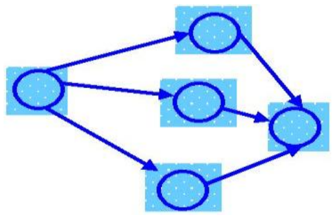
IF



While



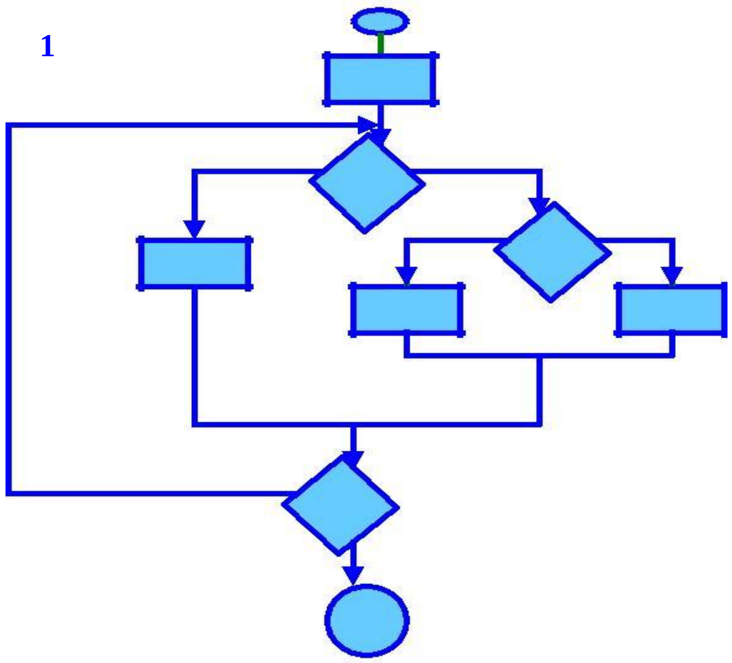
Repeat



Case

Figure 4: Different Versions of Flow Graphs

Example1. (Cyclomatic complexity)



The cyclomatic complexity for the upper graph (figure 5) is:

- V(G) = the number of predicate nodes +1 = 3+1 =4, or
- V(G)= the number of simple decisions +1 = 4.

Well, V(G) = 4, so there are four independent paths:

The path 1: 1, 2, 3, 6, 7, 8 The path 2: 1, 2, 3, 5, 7, 8; The path 3: 1, 2, 4, 7, 8;
The path 4: 1,2,4,7,2,4,…,7,8.
Now, test cases should be designed to exercise these paths.

Example2. (Cyclomatic complexity)

Cyclomatic complexity for graph which is represented in Figure 6 is:

V(G) = E – N + 2 = 17 – 13 + 2 = 6.

So, the basis set of independent paths is: 1-2-10-11-13;
1-2-10-12-13; 1-2-3-10-11-13; 1-2-3-4-5-8-9-2; 1-2-3-4-5-6-8-9-2; 1-2-3-4-5-6-7-8-9-2.

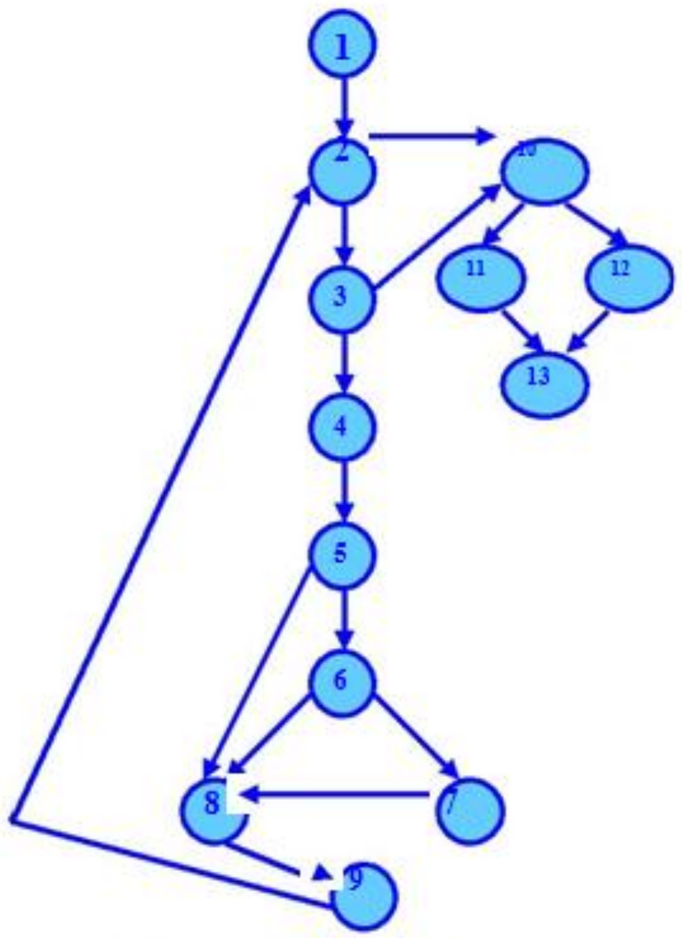


Figure 6: Graph for Example 2.

Example3.

Here are presented corresponding graph matrix and connection matrix for graph which is depicted in Figure 7.

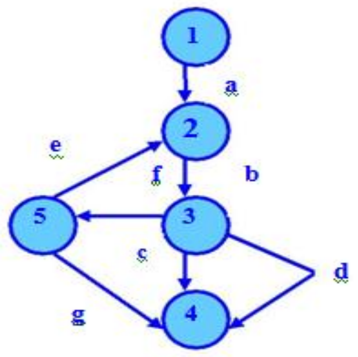


Figure 7: Graph for Example 3.

Table 1: Graph Matrix

Node	1	2	3	4	5
1		a			
2			b		
3				d, c	f
4					
5		e		g	

Table 2: Connection Matrix

Node	1	2	3	4	5	connections
1		1				1-1=0
2			1			1-1=0
3				1, 1	1	3-1=2
4						0
5		1		1		2-1=1

Cyclomatic complexity is 2+1= 3

3.8. Loop Testing

There are four types of loops:

1. Simple loops;
2. Concatenated loops;
3. Nested loops;
4. Unstructured loops.

3.8.1. Simple Loops

It is possible to execute the following tests:

1. Skip the loop entirely;
2. Only one pass through the loop;
3. Two passes through the loop;
4. m passes through the loop where m<n;
5. n-1, n, n+1 passes through the loop, where n is the maximum number of allowable passes through the loop.

A typical simple loop is depicted in Figure 8.

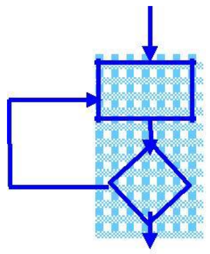


Figure 8: Simple Loop

3.8.2. Nested Loops

If one uses this type of loops, it can be possible that the number of probable tests increases as the level of nesting grows. So, one can have an impractical number of tests. To correct this, it is recommended to use the following approach:

- Start at the innermost loop, and set all other loops to minimum values;
- Conduct simple loop tests for the innermost loop and holding the outer loop at their minimum iteration parameter value;
- Work outward, performing tests for the next loop;
- Continue until all loops have been tested.

A typical nested loop is depicted in Figure 9.

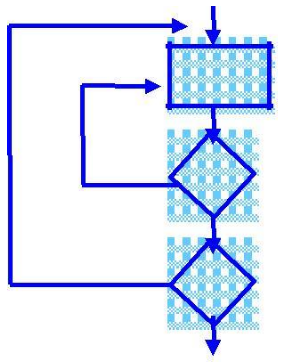


Figure 9: Nested Loop

3.8.3. Concatenated Loops

These loops are tested using simple loop tests if each loop is independent from the other. In contrary, nested loops tests are used.

A typical concatenated loop is presented in Figure 10.

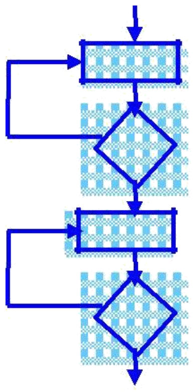


Figure 10: Concatenated Loop

3.8.4. Unstructured Loops

This type of loop should be redesigned.

A typical unstructured loop is depicted in Figure 11.

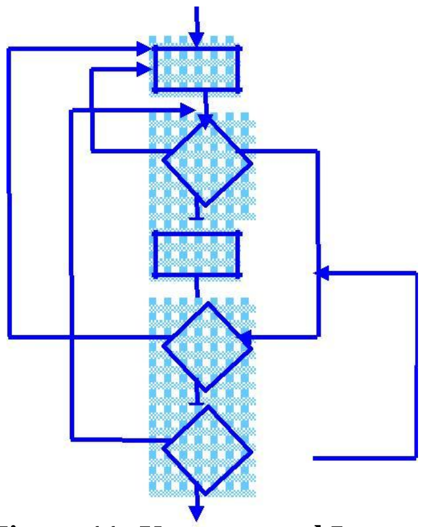


Figure 11: Unstructured Loop

3.9. Control Structure Testing

Two main components of classification of Control Structure Testing (Figure 12) are described below.

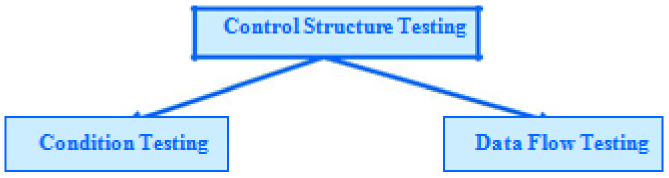


Figure 12: Classification of Control Structure Testing

3.9.1. Condition Testing

By this technique, each logical condition in a program is tested.

A ***relational expression*** takes a form:

E_1 < relational - operator >E_2 ,

Where E1 and E2 are arithmetic expressions, and relational operator is one of the following: $<$, $=$, $\neq$,$\leq$, $>$, or $\geq$.

A ***simple condition*** is a Boolean variable or a relational expression, possibly with one NOT operator.

A ***compound condition*** is made up of two or more simple conditions, Boolean operators, and parentheses.

This technique determines not only errors in the conditions of a program but also errors in the whole program.

3.9.2. Data Flow Testing

By this technique, one can choose test paths of a program based on the locations of definitions and uses of variables in a program.

Unique ***statement number*** is allocated for each statement in a program. For statement with S as its statement number, one can define:

DEF(S) = {X| statement S contains a definition of X}

USE(S) = {X| statement S contains a use of X}.

The definition of a variable X at statement S is live at statement S' if there exists a path from statement S to S' which does not contain any condition of X.

A definition-use chain (or DU chain) of variable X is of the type [X, S, S'] where S and S' are statement numbers, X is in DEF(S), USE(S'), and the definition of X in statement S is live at statement S'.

One basic strategy of this technique is that each DU chain be covered at least once.

4. TEST TECHNIQUES ACCORDING TO THE PROJECT OF THE IEEE COMPUTER SOCIETY, 2004.

The IEEE Computer Society is established to promote the advancements of theory and practice in the field of software engineering.

This Society completed IEEE Standard 730 for **software quality assurance** (it is any systematic process of checking to see whether a product or service being developed is meeting specified requirements, see [18]) in the year 1979. This was the first standard of this Society. The purpose of IEEE Standard 730 was to provide uniform, minimum acceptable requirements for preparation and content of software quality assurance plans. Another, new standards are developed in later years.

These standards are meaningful not only for promotion software requirements, software design and software construction, but also for software testing, software maintenance, software configuration management and software engineering management.

So, for improving software testing and for decreasing risk on all fields, there is classification of test techniques according to this Society, which is listed below.

- ***Based on the software engineer's intuition and experience:***
  1. **Ad hoc testing** – Test cases are developed basing on the software engineer's skills, intuition, and experience with similar programs;
  2. **Exploratory testing** – This testing is defined like simultaneous learning, which means that test are dynamically designed, executed, and modified.

- ***Specification-based techniques:***

1. **Equivalence partitioning**;
2. **Boundary-value analysis**;
3. **Decision table:** Decision tables represent logical relationships between inputs and outputs (conditions and actions), so test cases represent every possible combination of inputs and outputs;
4. **Finite-state machine-based:** Test cases are developed to cover states and transitions on it;
5. **Testing from formal specifications:** The formal specifications (the specifications in a formal language) provide automatic derivation of functional test cases and a reference output for checking test results;
6. **Random testing:** Random points are picked within the input domain which must be known, so test cases are based on random.

- ***Code-based techniques*:**

1. **Control-flow-based criteria:** Determine how to test logical expressions (decisions) in computer programs (see [19]). Decisions are considered as logical functions of elementary logical predicates (conditions) and combinations of conditions' values are used as data for testing of decisions. The definition of every control-flow criteria includes a statement coverage requirement as a component part: every statement in the program has been executed at least once. Control–flow criteria are considered as program-based and useful for white-box testing. For control-flow criteria, the objects of investigation have been relatively simple: Random Coverage, Decision Coverage (*every decision in the program has taken all possible outcomes at least once*), Condition Coverage (*every condition in each decision has taken all possible outcomes at least once*), Decision/Condition Coverage (*every decision in the program has taken all possible outcomes at least once and every condition in each decision has taken all possible outcomes at least once*) , etc.
2. **Data flow-based criteria**
3. **Reference models for code-based testing** – This means that the control structure of a program is graphically represented using a flow graph.

- ***Fault-based techniques:***

1. **Error guessing** – Test cases are developed by software engineers trying to find out the most frequently faults in a given program. The history of faults discovered in earlier projects and the software engineer's expertise are helpful in this situations;
2. **Mutation testing** - *A mutant* is a modified version of the program under test. It is differing from the program by a syntactic change. Every test case exercises both the original and all generated mutants. Test cases are generating until enough mutants have been killed or test cases are developed to kill surviving mutants.

- ***Usage-based techniques*:**

1. **Operational profile** – From the observed test results, someone can infer the future reliability of the software;
2. **Software Reliability Engineered Testing**.

- ***Techniques based on the nature of the application:***

1. **Object-oriented testing** – By this test technique we can find where the element under test does

not perform as specified. Besides, the goal of this technique is to select, structure and organize the tests to find the errors as early as possible (see [25]).

2. **Component-based testing** – Is based on the idea of creating test cases from highly reusable test components. *A test component* is a reusable and context-independent test unit, providing test services through its contract-based interfaces. More about this test technique on: http://www.componentbasedtesting.org/ site/.
3. **Web-based testing** – Is a computer-based test delivered via the internet and written in the "language" of the internet, HTML and possibly enhanced by scripts. The test is located as a website on the tester's server where it can be accessed by the test-taker's computer, the client. The client's browser software (e.g. Netscape Navigator, MS Internet Explorer) displays the test, the test-taker completes it, and if so desired sends his/her answers back to the server, from which the tester can download and score them (see [20]).
4. **GUI testing –** Is the process of testing a product that uses a graphical user interface, to ensure it meets its written specifications (see [6]).
5. **Testing of concurrent programs**;
6. **Protocol conformance testing** – ***A protocol*** *describes the rules with which computer systems have to comply in their communication with other computer systems in distributed systems* (see [23]). ***Protocol conformance testing*** *is a way to check conformance of protocol implementations with their corresponding protocol standards, and an important technology to assure successful interconnection and interoperability between different manufacturers* (see [24]). Protocol conformance testing is mostly based on the standard ISO 9646: "Conformance Testing Methodology and Framework" [ISO 91]. However, this conventional method of standardization used for protocol conformance test, sometimes gives wrong test result because the test is based on static test sequences.
7. **Testing of real-time systems** – More than one third of typical project resources are spent on testing embedded and real-time systems. Real-time and embedded systems require that a special attention must be given to timing during testing. According to the [21], real-time testing is defined as *evaluation of a system (or its components) at its normal operating frequency, speed or timing*. But, it is actually a conformance testing, which goal is to check whether the behavior of the system under test is correct (conforming) to that of its specification (see [22]). Test cases can be generated offline or online. In the first case, the complete test scenarios and verdict are computed a-priori and before execution. The offline test generation is often based on a coverage criterion of the model, on a test purpose or a fault model. Online testing combines test generation and execution.
8. **Testing of safety-critical systems**.

- ***Selecting and combining techniques*:**

1. **Functional and structural**;
2. **Deterministic vs. random** - Test cases can be selected in a deterministic way or randomly drawn from some distribution of inputs, such as is in reliability testing.

5. CONCLUSION

Software testing is a component of **software quality control** (**SQC**). *SQC means control the quality of software engineering products, which is conducting using tests of the software system* (see [6]). These tests can be: *unit tests* (this testing checks each coded module for the presence of bugs), *integration tests* (interconnects sets of previously tested modules to ensure that the sets behave as well as they did as independently tested modules), or *system tests* (checks that the entire software system embedded in its actual hardware environment behaves according to the requirements document). SQC also includes formal check of individual parts of code, and the review of requirements documents.

SQC is different from **software quality assurance** (**SQA**), *which means control the software engineering processes and methods that are used to ensure quality* (see [6]). Control conduct by inspecting quality management system. One or more standards can be used for that. It is usually *ISO 9000*. SQA relates to the whole software development process, which includes the following events: software design, coding, source code control, code reviews, change management, configuration management, and release management.

Finally, *SQC is a control of products, and SQA is a control of processes*.

Eventual bugs and defects reduce application functionality, do not look vocational, and disturb company's reputation. Thence, radically testing is very important to conduct. At that way, the defects can be discovered and repaired. Even if customers are dissatisfied with a product, they will never recommend that product, so product's cost and its popularity at the market will decrease.

Besides, *customer testing* is also very important to conduct. Through this process one can find out if application's functions and characteristics correspond to customers, and what should be changed in application to accommodate it according to customer's requests.

Large losses can be avoided if timely testing and discovering bugs in initial phases of testing are conducting. Deficits are minor if the bugs are discovered by testing within the company, where developers can correct errors rather than if the bugs are discovered in the phase of customer testing, or when the application is started "live" in some other company or system for which the application is created. In that case, the losses can be enormous.

Therefore software testing is greatly important, and test techniques too, because they have the aim to improve and make easier this process.

There is considerable controversy between software testing writers and consultants about what is important in software testing and what constitutes responsible software testing.

So, some of the major controversies include:

- ***What constitutes responsible software testing?*** – Members of the "context-driven" school of testing believe that the "best practices" of software testing don't exist, but that testing is a collection of skills which enable testers to chose or improve test practices proper for each unique situation. Others suppose that this outlook directly contradicts standards such as *IEEE 829* test documentation standard, and organizations such as *Food and Drug Administration* who promote them.
- ***Agile vs. traditional*** – Agile testing is popular in commercial circles and military software providers. Some researchers think that testers should work under conditions of uncertainly and constant change, but others think that they should aim to at process "maturity".
- ***Exploratory vs. scripted*** – Some researchers believe that tests should be created at time when they are executed, but others believe that they should be designed beforehand.
- ***Manual vs. automated*** – Propagators of agile development recommend complete automation of all test cases. Others believe that test automation is pretty expensive.
- ***Software design vs. software implementation*** – The question is: Should testing be carried out only at the end or throughout the whole process?
- ***Who watches the watchmen*** – Any form of observation is an interaction, so the act of testing can affect an object of testing.

REFERENCES

1. http://www.his.sunderland.ac.uk/~cs0mel/comm83wk5. doc, February 08, 2009.
2. Stacey, D. A., "Software Testing Techniques"
3. Guide to the Software Engineering Body of Knowledge, Swebok – A project of the IEEE Computer Society Professional Practices Committee, 2004.
4. "Software Engineering: A Practitioner's Approach, 6/e; Chapter 14: Software Testing Techniques", R.S. Pressman & Associates, Inc., 2005.
5. Myers, Glenford J., IBM Systems Research Institute, Lecturer in Computer Science, Polytechnic Institute of New York, "The Art of Software Testing", Copyright 1979. by John Wiley & Sons, Inc.
6. Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/
7. http://www2.umassd.edu/CISW3/coursepages/pages/CI S311/outline.html
8. Parezanovic, Nedeljko, "Racunarstvo i informatika", Zavod za udzbenike i nastavna sredstva – Beograd, 1996.
9. Wei–Tek, Tsai, "Risk – based testing", Arizona State University, Tempe, AZ 85287
10. Redmill, Felix, "Theory and Practice of Risk-based Testing", Software Testing, Verification and Reliability, Vol. 15, No. 1, March 2005.
11. IEEE, "IEEE Standard Glossary of Software Engineering Terminology" (IEEE Std 610.12-1990), Los Alamitos, CA: IEEE Computer Society Press, 1990.
12. http://www.testingstandards.co.uk/living_glossary.htm# Testing, February 08, 2009.
13. http://www.pcmag.com/encyclopedia_term/0,2542,t=wh ite+box+testing&i=54432,00.asp, February 08, 2009.
14. http://searchsoftwarequality.techtarget.com/sDefinition/ 0,,sid92_gci1242903,00.html, February 08, 2009.
15. Janardhanudu, Girish, "White Box Testing", https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best- practices/white-box/259-BSI.html, February 08, 2009.
16. http://www.pcmag.com/encyclopedia_term/0,2542,t=bla ck+box+testing&i=38733,00.asp, February 08, 2009.
17. http://www.pcmag.com/encyclopedia_term/0,2542,t=gra y+box+testing&i=57517,00.asp, February 08, 2009.
18. http://searchsoftwarequality.techtarget.com/sDefinition/ 0,,sid92_gci816126,00.html, February 08, 2009.
19. Vilkomir, A, Kapoor, K & Bowen, JP, "Tolerance of Control-flow testing Criteria", Proceedings 27th Annual International Computer Software and applications Conference, 3-6 November 2003, 182-187, or http://ro.uow.edu.au/infopapers/88
20. http://www2.hawaii.edu/~roever/wbt.htm, February 08, 2009.
21. http://www.businessdictionary.com/definition/real-time- testing.html, February, 2009.
22. Mikucionis, Marius, Larsen, Kim, Nielsen, Brian, "Online On-the-Fly Testing of Real-time systems", http://www.brics.dk/RS/03/49/BRICS-RS-03-49.pdf, February, 2009.
23. Tretmans, Jan, "An Overview of OSI Conformance Testing", http://www.cs.aau.dk/~kgl/TOV03/iso9646.pdf
24. http://www.ifp.uiuc.edu/~hning2/protocol.htm, February 2009.
25. http://it.toolbox.com/blogs/enterprise-solutions/better- object-oriented-testing-21288, February 2009.
26. http://pages.cs.wisc.edu/~bart/fuzz/fuzz.html, February, 2009.
27. http://www.goldpractices.com/practices/mbt/index.php, February, 2009.
28. http://blogs.msdn.com/nihitk/pages/144664.aspx, February, 2009.