A Study of Software Testing Techniques and Their Use in Enterprises

[1]Komal Alwani, [2]Dr. Pushpneel Verma

[1]Research Scholar, Deptt. Of Computer Application, Bhagwant University, Ajmer, Rajasthan

[2]Associate Professor, Bhagwant University, Ajmer, Rajasthan

Abstract

This paper presents the software testing techniques, including static and dynamic testing as a code analysis, test design based methods to create test cases, software testing levels that are analyzed as a stage of software development. Test execution types, including manual and automated testing, are analyzed as well. Further, the practical use of software testing techniques in enterprises is examined to identify how enterprises adopt those software testing techniques and what benefits and limitations they are facing while using any of software testing techniques.

Keywords: Software Quality, Static Analysis, Benefits, Software Development etc.

---

1. INTRODUCTION

1.1 Testing techniques as code analysis

The activities for software quality assessment can be divided into two broad categories, such as static analysis and dynamic analysis (Naik & Tripathy, 2008). Static analysis and dynamic analysis are related with each other from code's perspective, as the first one describes the testing without executing the code, while other uses that analyzed code for execution (it evaluates the dynamic behavior). Some researchers suggests to create a hybrid analysis that combines both approaches for better effectiveness (Ernst, 2003). It is noticed that both should be performed repeatedly and alternated. To understand better each of those code analysis techniques, the main principles and their types will be introduced in the following subchapters.

1.1.1. Static testing

Static testing (static analysis) is performed before the code is executed or completed. It has been already introduced in subchapter 1.1 as a technique for verification process. The following types of static testing are distinguished by Graham et al. (2008) and Myers et al. (2011) and analyzed below:

Code or Design Inspection - the most formal review and aimed at detecting all faults, violations of development standards, and other problems in design and code. According to Fagan (2001), all required documents, including detailed design in specific areas like paths, logic of code, should be prepared and presented for inspection meeting. During inspection process the code is inspected in order to found defects that are handed to the author for fixing.

Review (informal, peer, technical, management) - in practice, technical reviews vary from quite informal to very formal (Graham et al., 2008). Review are performed by the experts (such as architects, designers, key users). During review actual work is compared with established standards to determine whether the product is ready to proceed with the next phase of SDLC.

Walk-through - a non formal process when a programmer leads team members and other interested parties through a segment of documentation or code, and the participants ask questions and make comments about possible errors, violation of development standards, and other problems (Graham et al., 2008; “IEEE Standard Glossary of Software Engineering Terminology,” 1990). Although, DeVolder et al. (2008) and Hass, (2008) define an additional technique - Audit which is the most formal static testing technique. Audits are performed by external auditors with the purpose of providing "an independent evaluation of an activity’s compliance to applicable process descriptions, contracts, regulations, and/or standards" (Hass, 2008, p. 301). The author discovers the main disadvantages of audits - they are quite expensive and the least effective static testing type; however, audits are usually performed because

they are mandatory in some context. The main similarities and differences in the most commonly used techniques are illustrated in a table below (see Table 1.1).

Table 1.1: General principles for Static Testing techniques

	Walk-through	Technical Review	Management Review	Inspection
Primary purpose	Finding defects	Finding defects	Finding defects	Finding defects
Secondary purpose	Sharing knowledge	Make decisions	Monitor and control process	Process improvement
Preparation	Usually none	Familiarization	Familiarization	Formal preparation
Usage of basis	Rarely	Maybe	Maybe	Always
Leadership of the meeting	Author	As appropriate	As appropriate	Trained moderator
Recommended group size	2-7	3 or more	3 or more	3-6
Formal procedure	Usually not	Sometimes	Sometimes	Always
Volume of material	Relatively low	Moderate to high	Moderate to high	Relatively low
Collection of metrics	Usually not	Sometimes	Sometimes	Always
Output	Sometimes an informal report	More or less formal report	More or less formal report	Defect list, measurements, and formal report

Source: (Hass, 2008)

Regarding the common features of techniques, the main purpose of static testing in general can be defined – defects prevention in early stage of SDLC and improvement of software quality (correctness of code, compliance of requirements) with the aid of team members involved. Whilst, the researchers (Nidhra & Dondeti, 2012; Saglietti et al., 2008) presents more detailed definition static testing purpose: to check whether the code meets functional requirements, design, coding standards; to identify whether and all functionalities are covered; to uncover incorrect programming assumptions; to find logical and random typographical errors in the program code. Emanuelsson & Nilsson (2008) distinguishes the main runtime problems (errors) that are detected by static testing: Improper resource management - resource leaks of dynamically allocated memory, files, sockets that are no longer used; Illegal operations of arithmetic functions, illegal values, arrays addressing, null pointers referencing etc.; Dead code and data - code and data that is not reachable or not used; Incomplete code - missing initialized variables, functions with unspecified return values and incomplete branching statements.

Some of such errors can be detected by tools instead of manual testing (Hass, 2008). In spite of the variety of static analysis tools available on the market (e.g. "PolySpace", "C Verifier", "SonarQube"), or as open source systems (e.g. ARCHER, BOON, SPLINT, UNO) ("SonarQube," 2016; Zitser et al., 2004), some struggling issues can be faced while choosing the right tool or considering the need of it: the functionality of tool depends on the specified programming language which is designed for; more complex system requires deeper analysis compared with a simple one; limited enterprises resources restrict the choice of desired tool. Some of tools are standard development tools, such as compilers or linkers, while others are aimed for code analysis that monitor and track the following issues (Graham et al., 2008; Hass, 2008): the flow of code instructions; the data flow accessed and modified by code; compliance to standards that consists of a set of programming rules and other conventions; calculation of code metrics that analyze the depth of nesting, cyclamate number and number of lines of code. After discussion of static testing features, the value for all SDLC is identified: static testing reduces the chances of failures in later phases of SDLC; it prevents from runtime problems (errors) that are detected mainly by static software testing technique (Emanuelsson & Nilsson, 2008); a vast of complex rules in the coding standards can be verified by tools instead of a time-consuming manual review. It is noticed that missed defects during static testing could be detected at the latest phases of SDLC; thus, it affects the cost of whole software development process.

1.1.2. Dynamic testing

Dynamic testing (or dynamic analysis) compared with static testing executes the software actually. It is defined as the process of evaluating a system or a component based upon its behavior during execution in order to expose possible program failures (Hass, 2008). It is done by tools that helps to gather run-time information about the

behavior and state of software, thus, Graham et al. (2008) explains that they are 'analysis' rather than 'testing' tools. The main features of dynamic analysis tools are listed below:

- to report on the state of software during its execution (Naik & Tripathy, 2008);
- to monitor the allocation, use and reallocation of memory (Naik & Tripathy, 2008);
- to identify unassigned pointers (Hass, 2008; Naik & Tripathy, 2008);
- to detect memory leaks (Naik & Tripathy, 2008; Graham et al., 2008; Hass, 2008);
- to identify pointer arithmetic errors, e.g. null pointers (Graham et al., 2008; Hass, 2008; Naik & Tripathy, 2008);
- to identify time dependencies (Graham et al., 2008; Naik & Tripathy, 2008);
- coverage analysis (Hass, 2008) - these tools provide objective measurement for some white-box test coverage metrics (e.g. statement coverage or branch coverage; both will be presented in the further subchapter);
- performance analysis (Hass, 2008) - it measures the performance of a product under the controlled circumstances before the product is released; Some tools (called as memory debuggers) used for detecting memory leaks and uses of dead storage are as follows: "Purify" and "LCLint" (Ernst, 2003). Whereas, other tools are more powerful and include more dynamic analysis features mentioned above - "VB Watch" (Aivosto, 2016) or "IBM Rational AppScan" ("IBM - Software - IBM Security AppScan," 2016).

Dynamic testing executes the software and validates the output with the expected outcome and it can be either black or white box testing (Graham et al., 2008). Since this technique is performed during validation process, the testable levels (test levels will be analyzed more detailed in the subchapter 2.3) are distinguished:

Unit Testing: individual units or modules are tested by the developers. It involves testing of source code by developers as well.

Integration Testing: individual modules are grouped together and tested by the developers. The purpose is to determine that modules are working as expected once they are integrated together.

System Testing: checking whether the system or application meets the BRS and SRS by testing the whole system. To summarize both static and dynamic testing, the main features are identified. Static testing reduces the chances of failures in later phases of SDLC; focus on prevention of defects during verification process. However, it is time consuming activity. Whereas, dynamic testing executes the software and validates the output with the expected outcome and it can be either black or white box testing. The main focus is on finding defects. Both techniques can be performed by tools, however, there are some limitations. Automated tools of static analysis do not support all programming languages, while tools for dynamic analysis provide a false sense of security that everything is being addressed.

2. TEST DESIGN BASED TECHNIQUES

Traditionally software testing techniques can be broadly divided into white-box testing and black-box testing (Liu & Kuan Tan, 2009), however there are few more test design techniques that are used rarer than white-box and black-box techniques. They are as follows: experience-based (Graham et al., 2008; Hambling & Morgan, 2011; Hass, 2008; Myers et al., 2011) and error guessing (Myers et al., 2011) or called as defect-based (Hass, 2008). Sometimes the gray-box technique is separated as the different approach even if it based on both white-box and black-box techniques (Mohd Ehmer Khan & Khan, 2012). All these approaches focus on the sources of information for test design. There are many advantages of using techniques to design test cases. They provide good insights for finding possible faults - this is the most essential objective for all software development. Indeed, white-box testing and black-box testing techniques can be perform by static and dynamic analysis in order to find defects. White-box testing and black-box testing are considered corresponding to each other. Some researchers underline that it is essential to cover both specification and code actions in order to test software more efficiency (Jorgensen, 2016; Liu & Kuan Tan, 2009). Whereas, Hass (2008) see test design based techniques are as a very precise and systematic analysis of BRS or SRS which makes testing more effective and corrective. Designing test cases by these techniques also shows the experience of testers, whereas, other testers are able to learn from provided test cases by executing them. One of the most important thing for black-box and white-box testing is to achieve a full coverage of what is required to cover: it could be requirements, or statements, or paths - it depends on selected technique and test objectives. It is noted that some difficulties could be faced with even when the full coverage is obtained: faults could remain undetected because of non conformance of the code and users expectations. To overcome this or mitigate this risk as much as possible, the validation of the requirements should be performed narrowly before starting the dynamic testing. As we discussed in a previous chapter - first focus should be made on the first phase of SDLC. More detailed white-box testing and black-box testing will be analyzed further in this subchapter.

Grey box is seen as the combination of white-box and black-box techniques (Mohd Ehmer Khan & Khan, 2012; Sawat, Bari, Chawan, & P. M., 2012). In grey box testing the tester must have knowledge of internal data structures and algorithm of application, for the purpose of designing test cases (M. E. Khan, 2011a). In spite of combination of two techniques, the grey box testing won't be discussed detailed; the main focus is on mostly used techniques.
According to Hambling & Morgan (2011) experience-based techniques are based on the users' and the testers' knowledge and skills to determine the most important areas of a system to be chosen to test. Experience-based techniques go together with specification-based and structure-based techniques, and are also used when there is no specification from which to derive specification-based test cases, or an inadequate or out of dated specification is used, or there is no time to run the full structured set of tests. It is recommended to use experience-based techniques even when specifications are available. Structured tests could be augmented with some additional steps in order to find defects similar to those which are founded by experience in other similar systems. Some types of experience-based techniques are as follows (Graham et al., 2008; Hass, 2008): error guessing, checklist-based. Error guessing depends on experience of tester as good testers know where the defects are most likely to be. Second type is uses checklists to guide testing where the checklist is basically a high-level list, or a reminder list, of areas to be tested. Finally, the main focus of exploratory testing is on exploring software with intent to understand its behavior. The main feature of these types that they are based by tester's experience. The may be used before the other techniques to uncover "weak" areas, but experience-based techniques must never be the only technique to be used.
Taking into consideration defect-based technique, it is defined as less systematic than the previously discussed techniques, since it is usually not possible to make exhaustive collections of expected defects. Whereas, experience-based testing techniques are based on the tester's experience with testing, development, similar applications, the same application in previous releases, and the domain itself (Graham et al., 2008). Furthermore, the main test design techniques can be classified in smaller techniques, while the wider categorization group, static and dynamic testing, covers all previously mentioned techniques. The tree categorization of mostly used software testing techniques is presented below (see Figure 1.1). These classified techniques will be discussed further in this chapter.

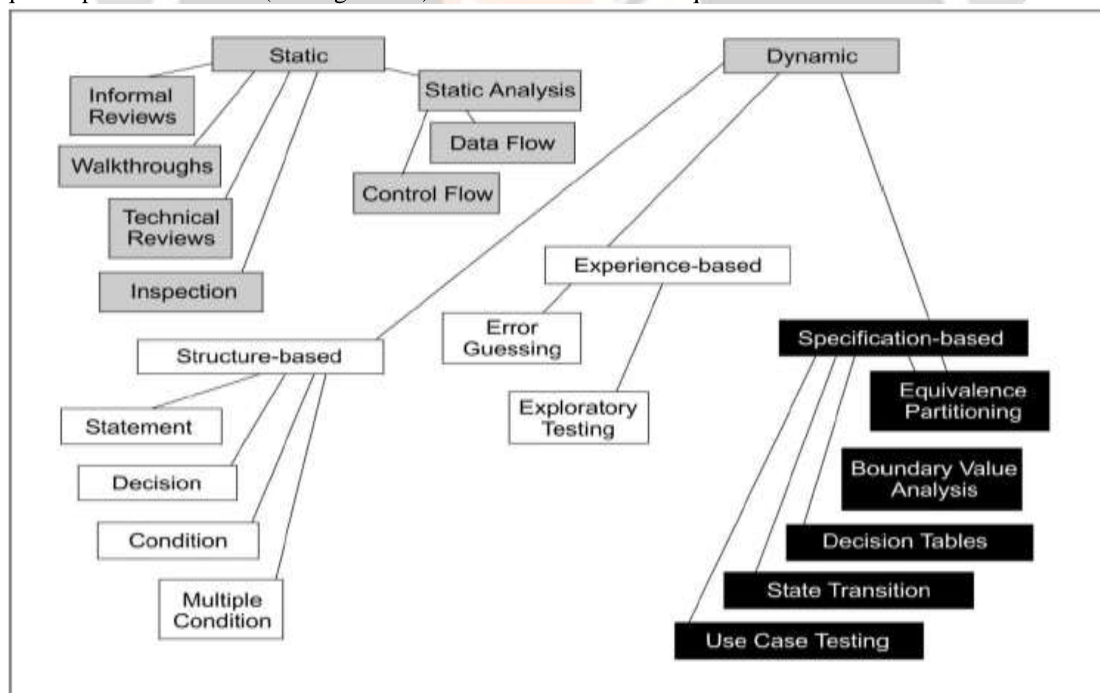

Figure 1.1: The tree structure of the testing techniques
Source: (Hambling & Morgan, 2011)

2.1. Structure-based (white-box) techniques

White-box testing techniques are called structural testing techniques or as Myers et al. (2011) noticed - logic-driven techniques. Structural testing is defined as testing that takes into account the internal mechanism of a system or component ("IEEE Standard Glossary of Software Engineering Terminology," 1990). Indeed, its techniques are based on deriving test cases directly from the internal structure of a component or system with intent to explore system or component structures at several levels. Traditionally the internal structure has been interpreted as the

structure of the code (Liu & Kuan Tan, 2009). According to Hass (2008) and Naik & Tripathy (2008), in structural testing (white-box), the main focus is on the testing of code and they are primarily used for component testing and low-level integration testing. The researchers also notes the use in system (Graham et al., 2008; Sawat et al., 2012) and acceptance testing (Graham et al., 2008) with the different structures (e.g. the coverage of menu options could be the structural element in system or acceptance testing). Acceptance testing is defined as "formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system ("IEEE Standard Glossary of Software Engineering Terminology," 1990). Acceptance testing will be discussed more detailed in second subchapter. Developing the previous example of code coverage, the following code coverage criteria of structural test case design techniques are enumerated further (Hass, 2008):

1. Statement testing - test cases are designed to execute statements that are defined as a no comment or nonwhite space entity in a programming language,
2. Decision testing/branch testing - testing of decision outcomes. Mostly a decision has two outcomes, such as "True" or "False", but it might have more outcomes, for example, in "case of ..." statements.
3. Condition testing - testing of conditional expressions (e.g. AND, OR, a < b etc.).
4. Multiple condition testing - combinations of condition outcomes are tested in order to get fully multiple combination coverage.
5. Condition determination testing - testing of branch condition outcomes that independently affect a decision outcome.
6. LCSAJ (loop testing) - testing of loop iterations that start at a specific point in the code and end with a jump (or at the end of the component).
7. Path testing - testing of a sequence of executable statements in a component from an entry point to an exit point to get full coverage of paths.
8. Inter-component testing - this testing technique is used in integration testing where the test objects are interfaces (interfaces exist between interacting components and systems).

These different techniques exercise every visible path of the source code to minimize errors and create an error-free environment. The main view of white-box testing is to get knowledge on which line of the code is being executed and being able to identify what the correct output should be. Galin (2004) and Graham et al. (2008) identify the main advantages: structure-based techniques can be used at all levels of testing starting from unit (component) and ending at acceptance testing, direct statement-by-statement checking of code ensures software correctness as expressed in the processing paths, including whether the algorithms were correctly defined and coded. The research by M. E. Khan (2011b) presents few more benefits: white-box testing techniques reveals error in hidden code by removing extra lines of code and maximum coverage is attained during test scenario writing. Whereas some disadvantages are seen as well: it is very expensive testing techniques as they require a skilled tester to perform such testing; many paths remain untested because of difficulties to discover hidden errors in a complex system; some of the codes omitted in the code could be missed out (M. E. Khan, 2011b). In addition, there is no ability to test software performance in terms of reliability, load durability, and other testing classes related to operation, revision and transition factors (Graham et al., 2008).

To conclude this subchapter, the main features of white-box techniques are distinguished. First of all, white-box techniques are based on deriving test cases directly from the internal structure of a component or system and the main purpose is to explore system or component structures at several levels. Furthermore, in spite of the fact that they are primarily used for component testing and low-level integration testing, system and acceptance levels are tested by white-box techniques as well. Finally, white-box testing techniques are seen as very expensive testing techniques as they require a skilled tester to perform such testing, whilst, it reveals error in hidden code by removing extra lines of code and maximum coverage is attained during test scenario writing.

2.2. Specification-based (black-box) techniques

Black-box techniques, known as specification-based techniques, are also called as functional testing (Liu & Kuan Tan, 2009) or input/output driven testing techniques (Graham et al., 2008; Sawat et al., 2012) because they view the software as a black-box with inputs and outputs generated in response to selected inputs and execution conditions. According IEEE Standard Glossary of Software Engineering Terminology (1990) functional testing is defined as "testing conducted to evaluate the compliance of a system or component with specified functional requirements". Thus, the main focus of functional techniques is on validating the software whether it meets requirements. These techniques design test cases based on the information from the requirements specification, including both functional and non-functional (e.g. performance, usability, portability, maintainability, etc.) aspects. Software tester is concentrating on what the software does according the specified requirements instead of analyzing how the system

works. According to Hass (2008), these test case design techniques can be used in all stages and levels of testing, especially, they are useful in high-level tests, such as acceptance testing and system testing, where the test cases are designed from the requirements. Test cases can be supplied with structural or white-box test in order to get full coverage. The functional test case design techniques are enumerated by Hass (2008):

1. Equivalence partitioning and boundary value analysis - equivalence partitioning can reduce the number of test cases, as it divides the input data of a software unit into partition of data from which test cases can be derived. While boundary value analysis focuses more on testing at boundaries, or where the extreme boundary values are chosen (Graham et al., 2008; M. E. Khan, 2011a)
2. Domain analysis - it can be used to identify efficient and effective test cases when multiple variables can should be tested together (as multidimensional partitions or domain).
3. Decision tables - this technique is applied to specific situations or inputs where there are different combinations of inputs that result in different actions as well (Graham et al., 2008).
4. Cause-effect graph - testing begins by creating a graph and establishing the relation between the effect and its causes (M. E. Khan, 2011a).
5. State transition testing - it is used where some aspect of the system can be defined as a ‘finite state machine’. A system where different output is get for the same input, depending on what has happened before, is a finite state system (Graham et al., 2008). It is useful for navigation of graphical user interface (M. E. Khan, 2011a).
6. Classification tree method - partitioning of different classes are made by identifying test relevant aspects (classifications) and their corresponding values (classes).
7. Pair wise testing - test cases are designed to execute possible combinations of each pair of input parameters (M. E. Khan, 2011a).
8. Use case testing - testing the main flow and alternative flow (if it is needed) step by step as it is specified in the description of use case.
9. Syntax testing.

Regarding the testing techniques enumerated above it is assumed that black-box testing techniques have the biggest collection of testing methods that mainly focus on compliance of requirements and user needs (Graham et al., 2008; Myers et al., 2011; Nidhra & Dondeti, 2012; Sawat et al., 2012). Thus, these techniques are the most used while validating the software by BRS and SRS.

Research that was made by M. E. Khan, (2011a) has represented the main advantages of black box testing: efficient for large code segment, users perspective are clearly separated from developers perspective (programmer and tester are independent of each other). However, there are some limitations as well: test coverage is limited as the access to source code is not available; it is difficult to associate defect identification in distributed applications. Moreover, many software paths remain untested because of absence of control of line coverage (Galin, 2004). As test cases are created according to specified requirements (from business perspective), some part of the code lines could not be covered by test cases, as a result, black box tests may not execute particular code lines that are not covered by test cases.

To summarize the main features of black-box testing techniques some conclusions are made. Firstly, these techniques design test cases based on the requirements specification, including both functional and non-functional aspects, with intent to validate whether the software meets requirements. Further, these techniques can be used in all stages and levels of testing and they are seen as efficient for large code segments. Moreover, the independent work of programmer and tester enables efficient testing from user's perspective. However, some software paths could still remain untested as the functionality (derived from business requirements) covered by test cases does not include code coverage.

After discussion of box testing approaches, the main differences between them (including grey-box) can be distinguished (see Table 1.2).

Table 1.2: The comparison between three box approaches techniques

S. No.	Black Box Testing	Grey Box Testing	White Box Testing
1.	Analyses fundamental aspects only i.e. no proved edge of internal working	Partial knowledge of internal working	Full knowledge of internal working
2.	Granularity is low	Granularity is medium	Granularity is high
3.	Performed by end users and also by tester and developers (user acceptance testing)	Performed by end users and also by tester and developers (user acceptance testing)	It is performed by developers and testers
4.	Testing is based on external exceptions – internal behaviour of the program is ignored	Test design is based on high level database diagrams, data flow diagrams, internal states, knowledge of algorithm and architecture	Internal are fully known
5.	It is least exhaustive and time consuming	It is somewhere in between	Potentially most exhaustive and time consuming
6.	It can test only by trial and error method	Data domains and internal boundaries can be tested and over flow, if known	Test better: data domains and internal boundaries
7.	Not suited for algorithm testing	Not suited for algorithm testing	It is suited for algorithm testing (suited for all)

Source: (Mohd Ehmer Khan & Khan, 2012)

To sum up all analyzed design based testing techniques, they can be broadly divided into white-box testing and black-box testing. Other techniques, such as, experience-based and defect-based, are used rarely. All these approaches focus on the sources of information for test design. White-box testing and black-box testing techniques can be perform by static and dynamic analysis in order to find defects. The black-box techniques design test cases based on the requirements specification, including both functional and non-functional aspects, with intent to validate whether the software meets requirements. While, white-box techniques are based on deriving test cases directly from the internal structure of a component or system with intent to explore system or component structures at several levels. Grey box testing is seen as the combination of white-box and black-box techniques.

3. SOFTWARE TESTING LEVELS AND CORRESPONDING TESTING TYPES

There are generally four recognized levels of testing that need to be completed before a software can be delivered for users (Naik & Tripathy, 2008; Sawat et al., 2012): unit testing, integration testing, system testing and acceptance. However, some authors tend to include more testing types to categorization by levels: Alpha testing and Beta testing (Graham et al., 2008; Mailewa, Herath, & Herath, 2015), Installation testing (Myers et al., 2011), component (module) testing (Mailewa et al., 2015; Myers et al., 2011), regression testing (Naik & Tripathy, 2008). In our opinion, some techniques, such as Alpha testing, Beta testing and regression testing are different types of testing and they are not related with previous levels which describe levels from code's perspective. In other words, some part of code is merged with another part until the system is fully integrated with all small units (components). Therefore, those techniques will be discussed later as testing types.

Software tests are frequently grouped by software development process, or by the level of specificity of the test. Each phase of SDLC goes through the testing. Thus, main testing levels mentioned before are enumerated and described more detailed below:

1. Unit testing: "testing of individual hardware or software units or groups of related units" ("IEEE Standard Glossary of Software Engineering Terminology," 1990). Graham et al. (2008) and Myers et al. (2011) identify the main purpose: to find discrepancies between the program's units (modules) and their interface specifications, and to determine whether the application functions is designed correctly and meet the user specifications. One of the biggest benefits of this testing phase is that it can be run every time a piece of code is changed, allowing issues to be resolved at that moment. However, more attention to maintenance of such tests should be paid as from an every minor code change in a component, to the general refactoring can affect whole system and the tests will likely require revision (Di Tommaso & Roche, 2011).

2. Integration testing: a level of the software testing process where individual units are combined and tested as a group in order to test the behavior and functionality of both the modules after integration. There are few types of

integration testing (Hass, 2008): Big bang integration testing, Top down, Bottom up, Functional incremental. The main purpose of this level of testing is to reveal faults in the interaction between integrated units and to construct a reasonably stable system for system level testing (Naik & Tripathy, 2008).
3. System testing: according to Hass (2008) and Naik & Tripathy (2008) testing is performed on a complete, integrated system to evaluate the system's compliance with its specified requirements and to check that it meets quality standards. It includes a wide scope of testing techniques, for instance, functionality testing, security testing, load testing, stress testing, performance testing etc. System testing level is seen as a critical phase of SDLC because of the need to meet a tight schedule, to detect most of all faults, and verify that fixed defects are working properly without causing new faults.
4. Acceptance testing: acceptance testing focus on customer side and the main goal is to ensure that the requirements of the specification are met and the software satisfies the customer 's requirement (Hass, 2008).
In order to complete testing and detect the majority of defects (the exhausting full testing is impossible as we discussed before), Myers et al. (2011) suggests to use the model of test levels corresponding phases of SDLC (see Annex 2, page 71). This approach focuses on distinction of each testing process toward distinction of each development process by verifying each step separately. It means that each development step should be followed by appropriate testing technique which would discover the particular class of errors. The main advantage of this structure - it helps to avoid useless redundant testing and prevents from overlooking large classes of defects.
Whereas, Mailewa et al. (2015), Myers et al. (2011) and Sawat et al. (2012) support the idea of categorization component (or module) testing as well. The main purpose is the same as for previous testing levels - to find defects and to verify their proper functionality that satisfies BRS and SRS. Component testing may be performed in isolated system part which do not depend on development life cycle model chosen for that particular application (Sawat et al., 2012)
Further, the research by Nidhra & Dondeti (2012) identifies more testing techniques related with testing levels that were defined as a part of level testing by some researchers (Graham et al., 2008; Mailewa et al., 2015; Naik & Tripathy, 2008). The testing techniques and their purpose are as follows:
Regression Testing - "Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements" ("IEEE Standard Glossary of Software Engineering Terminology," 1990). In other words, the main aim is to ensure that the reliability of each software release and testing after changes has been made. Moreover, after retesting fixed defects tester should verify whether new defects into the system were not appeared (Jorgensen, 2016).
Alpha Testing - this technique is defined more like a strategy instead of testing method according to Graham et al. (2008). Alpha testing is usually done at the developer's site by a group that is independent of the design team in order to observe the users and note identified problems (Graham et al., 2008; Nidhra & Dondeti, 2012).
Beta Testing - comparing with Alpha Testing, this technique more focuses on the user's perspective and practices. It is done at the customer's site with no developer in site. The main purpose is to discover any flaws or issues with user's help (Graham et al., 2008; Nidhra & Dondeti, 2012).
Functional Testing is performed for a completed software- this testing is to verify that all functionality are implemented by BRS and SRS and the software works as expected. This technique was already discussed as black-box testing technique for designing test cases. There are some functional testing types, namely, usability, smoke, automated, acceptance, regression etc. Although this categorization has been made by Mailewa et al. (2015). On the other hand, some researchers include acceptance testing and regression testing into test level categorization as it was mentioned before. Moreover, according Graham et al. (2008) usability should be classified as non-functional testing as it tests the software without prepared requirements and checks whether the software is built in user-friendly form by following criteria: learnability, efficiency, satisfaction, memorability etc. The other technique, smoke testing is defined as a type of functional testing as it most often uses prepared test cases and verifies the conformance between system and requirements. The main difference compared with other functional techniques, it ensures that the major and the most critical functionalities (not full coverage) of the application are working properly. Some of previously examined techniques can be automated and used as automation testing tools, but this approach will be discussed later.
These testing types are based on white-box (structural) or black-box (functional) techniques, however the third category can be subtracted as well - non-functional testing. Indeed, this category is not a part of test design based techniques as it not requires test cases. Non-functional testing focus more on aspects of the software that may not be related to a specific function or user action. Non-functional testing includes the various types; the main activities are as follows (Graham et al., 2008):
Usability testing - as it was observed before.
Maintainability testing - with refers to quality factor "maintainability".

Portability testing - with refers to quality factor "portability".
Compliance testing - it verifies, whether the software meets the defined IT standards by the company.
Performance testing - "Testing conducted to evaluate the compliance of a system or component with specified performance requirements" (“IEEE Standard Glossary of Software Engineering Terminology,” 1990)
Security testing - this testing is about to ensure the security mechanisms in the software, such as user data, user authority, privacy (Myers et al., 2011).
Stress testing - "Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements" (“IEEE Standard Glossary of Software Engineering Terminology,” 1990).
Internationalization testing and Localization testing - these techniques tests the issues related with different languages used in a software. They verify whether the various languages and regions are adapted in a system and translations are made correctly (Graham et al., 2008). The correspondence between non-functional testing and test levels are similar like functional testing - both of them can be performed at all levels (Graham et al., 2008).
This chapter presented testing levels and distinguished the main four levels, such as unit testing, integration testing, system testing and acceptance. Further, the two main categories of techniques - functional and non-functional - have been examined and then listed some testing types under each main category. As software goes through testing at each phase of SDLC, hence each of testing techniques can be applied at each level. It is applicable for both, functional testing and non-functional testing. Finally, the table is provided to show the differences between the main techniques.

4. AUTOMATED TESTING

All techniques of testing discussed in previous subchapters can be defined as manual testing because of human involvement in test execution with a purpose to ensure that software's behavior is as expected, while automated testing does the same thing, except the fact that some manual testing activities are automated by tools. In more specific terms, “test automation is the use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes with predicted outcomes” (Huizinga & Kolawa, 2007). Indeed, manual testing is widely used comparing with automated testing. According to Mailewa et al. (2015) manual testing is applied more for smaller projects or for companies with limited financial resources; whereas, other enterprises see the benefits of automating some tests instead of running them manually. Mulder & Whyte (2013) states that software test automation could help to reduce testing costs and time dedicated for testing in software development. However, automated tests may not be useful if it is not applied in the right time, right context and with the appropriate approach. Implementation and maintenance of automated tests are expensive for company, thus more attention should be paid on when and what to automate (Garousi & Mäntylä, 2016). As Mulder & Whyte (2013) explains, wrong decisions made in selecting areas that should be automated, can lead to disappointments and major expenditures of software development, including human efforts and the cost of engagement automation tools, and automated testing sometimes is seen as a high-risk activity (Persson & Yilmazturk, 2004). The automation tools cover a wide range of activities and are applicable for use in all phases of the systems development life cycle. They could automate varies areas, some of them are as follows (Perry, 2006):
Executable specs This tool enables automatic execution of requirements specification. However, specification should be written in a such way that it can be compiled into a testable program.
Test data generator The main objective of this tool is to generate test data automatically for test purposes. It is useful foe large amounts of test transactions.
Tracing- A representation of the paths followed by computer programs as they process data or the paths followed in a database to locate one or more pieces of data used to produce a logical record for processing.
Although, besides these tools there are tools (e.g. "Selenium", "HP Quick Test Professional", "TestComplete", "LoadRunner") that helps to execute specified test automatically without human intervention. They mainly automate some testing techniques that we discussed before, for instance (Cem Kaner, 2014):
Function equivalence testing - generating random input data and comparing the behavior of the function under test with a reference program.
Random regression testing - system reuse already passed tests and then executes them in a random way automatically. Automated regression tests are useful in order to check whether the previous functionality are still working on every daily build version after changes (Graham et al., 2008). Daily build can be generated by automated tools as well (e.g. by Jenkins which creates a job to deploy an application every day with the newest changes in a code).
Hybrid performance and functional testing - running the system under load and monitoring system responsiveness (performance testing) as well as behavioral correctness.

The survey conducted by (D. M. Rafi et al., 2016) showed the main benefits and limitations of test automation from practitioner's perspective. Practitioners explained that automation saves their efforts in test executions, and according them, tests can be reused as well as repeated again. The other advantage is seen when several regressions testing rounds are needed and regression test coverage is improved as well by automated tests (D. M. Rafi et al., 2016; Naik & Tripathy, 2008). Naik & Tripathy (2008) adds one more advantage - increased test effectiveness. Regarding the limitations, the high initial cost for automation and its tools are highlighted (D. M. Rafi et al., 2016) - Mulder & Whyte (2013) also agrees with these disadvantages. Furthermore, training the staff is considerable question as well. Most of practitioners argue that that current test automation tools offer a poor fit for their needs or they need more training on specific tool. Despite the limitations enterprises still think about full automated testing which helps to reduce human efforts (Garousi & Mäntylä, 2016). On the other hand, the full coverage of test by automated testing is impossible in practice due to budget and time constraints according to research conducted by Garousi, Coşkunçay, Betin-Can, & Demirörs (2014) in Turkey. This view is supported by survey made by D. M. Rafi et al. (2016) as well.
In order to achieve successful use of test automation, the enterprise should asses their capabilities to use such tools by analyzing following issues (Naik & Tripathy, 2008; Persson & Yilmazturk, 2004): the system should be stable and functionalities are well defined, test cases that need to be automated should be prepared correctly, adequate budget should be allocated for testing tools, test automation strategy should be defined clearly etc. Without enterprise assessment, automation process could be done in a wrong way which leads to failure of automation engagement. Furthermore, to understand better test automation purpose, the differences between manual testing and automation testing are distinguished and illustrated in a table (see Table 3.3) below.

Table 1.3: The difference between Manual Testing and Automation Testing

Manual Testing	Automation Testing
Time consuming and tedious: Since test cases are executed by human resources it is slow and tedious.	Fast execution: Automation runs test cases significantly faster than human testers.
Huge investment in human resources: Test cases need to be executed manually so more testers are required in manual testing.	Less investment in human resources: Test cases are executed by using automation tool(s) so lesser number of testers are required in automation testing.
Less reliable: Manual testing is less reliable as tests may not be performed with precision each time because of human errors.	More reliable: Automation tests perform precisely same operation each time they are run.
Non-programmable: No programming can be done to write sophisticated tests which fetch hidden information.	Programmable: Testers can program sophisticated tests to bring out hidden information.
The results are late: programmers cannot see the result until the tester note down, type, print and copy the results.	Quick results: the programmers can see the result real-time in a networked environment.
Might catch more of the errors occurring due to human nature as the testers are human beings themselves.	Might skip errors occurring due to human nature as software cannot 'think' as human beings.
Can be cheaper for small software than automated testing.	Can be costly for smaller projects but cost-effective for larger projects.
Much better at visual testing.	Weak at visual testing. Software do not think like humans and thus it cannot decide whether a GUI is attractive, distractive or dull.

Source: (Mailewa et al., 2015)
To summarize automated testing, the main features are defined. The main difference between manual and automated testing is that manual testing uses human intervention in test execution with a purpose to ensure that software's behavior is as expected, while automated testing does the same thing, except the fact that some manual testing activities or techniques are automated by tools. The main benefits are as follows: automation saves their efforts in test execution, tests can be reused as well as repeated again, the coverage of automated regression tests is improved. However, high initial cost for automation and its tools are highlighted.

5. USE OF SOFTWARE TESTING TECHNIQUES IN ENTERPRISES

Enterprises use a variety of software testing techniques that are tending to improve software quality. Moreover, they should help testers on designing precise test cases and executing them more effective. However, we think some techniques are depreciated, while others are applied often, because users are used to use them. In fact, there is no

consensus on which technique is the most effective and appropriate to use; it depends on context. On the other hand, some factors could influence the decisions about which technique to choose. The majority of factors are presented by Vegas et al. (2002) in a table (see Annex 4, page 73) and some of them are listed below by Graham et al. (2008):
Models used in developing the system – appropriate technique can be chosen by models that are used to develop the current system. For example, state transition testing is an appropriate technique to use for a state transition diagram included in specification.
Similar type of defects – knowledge of the similar kind of defects (found in previous levels of testing or previous version of software) prompts to apply the same technique as the defect was detected (e.g. regression testing).
Risk assessment – the greater the risk (e.g. safety-critical systems), the more formal testing. technique should be used.
Customer and contractual requirements – sometimes customer specifies the particular testing techniques to use (most commonly statement or branch coverage).
Type of system used – for example, "a financial application involving many calculations would benefit from boundary value analysis".
Regulatory requirements – some industries should use specified testing by techniques regulatory standards. For example, "the aircraft industry requires the use of equivalence partitioning, boundary value analysis and state transition testing."
Time and budget of the project – limited time and budget make to apply the techniques that are known the best for detecting more defects.
Indeed, time and budget of the project are very important issues which affects all SDLC. In fact, it is stated that testing activities are more costly comparing with other activities of SDLC. In 1979, it is seen that approximately 50 percent of the elapsed time and more than 50 percent of the total cost of project management budget is allocated for testing (Myers et al., 2011). While a later survey performed in 1994, shows the decrease of cost spent for testing in a whole software development process: about 24% of the overall software budget and 32% of the total cost of project management budget (Perry, 2006). According to World Quality Report (Hans van Waayenburg & Raffi Margaliot 2016), the steady growth of quality assurance and testing budgets is seen since 2012. In addition, on average the enterprises are now spending 31% of its information technology budget on testing, compared with 35% in 2015, 26% in 2014, 23% in 2013, and 18% in 2012. In order to reduce the cost, more attention should be paid on the first stage of SDLC. This aspect was presented by McCall et al. (1977) as a guidelines in how to objectively specify the desired amount of quality at the system requirements specification phase. The research introduces into software quality factors and QA activities as it was already discussed in the paper.
Although, there are more studies conducted as a guide for testing to determine the best testing practices. For instance, the one by Bertolino (2007), Glass, Collard, Bertolino, Bach, & Kaner (2006), Vegas, Juristo, & Basili (2002) and Juristo, Moreno, & Strigel (2006). These research studies presented very significant amount of knowledge on good testing practices. The researchers determined the importance of the elicitation of main testing goals, management of testing processes, identification of test criteria on selection of appropriate testing technique. In fact, such knowledge can help to manage and improve software testing practices effectively and efficiently. Additionally, similar studies are prepared by Ng, Murnane, Reed, Grant, & Chen, (2004), Causevic, Sundmark, & Punnekkat (2010) and Lee, Kang, & Lee (2012), except the fact that the authors are using qualitative and quantitative methods instead of theoretical data gathering methods. The survey (Ng et al., 2004) was conducted to study the software testing practices in Australia. The research identified the major testing activities performed in enterprises: designing test cases, documenting test results, re-using the same test cases after changes were made to the software. Almost all surveyed enterprises agreed that formal tests were performed to ensure the developed software meets its requirements and specifications and they suggested to use more user acceptance testing. Regarding the defects statistics, it was found that between 40 to 59 % of such faults were related to specification defects; thus such amount of defects increases the cost of bug-fixing. Moreover, if those bugs were detected in later phases of SDLC, the more significant increase of cost is seen as defects become faults. In such case, more attention should be paid in the first stage of SDLC during validation of requirements specification as we discussed in the first chapter. Further, the most critical barrier to adopt specific testing technique was reported as a lack of expertise, while the adoption of automated tools is seen as costly to use. Despite these facts, a bit more than half of surveyed enterprises stated that they have automated some of their testing activities. While the regular staff training on automated testing and other issues related with software testing was provided only in some enterprises. Most of enterprises agreed that the main reason is cost for such training.
Another published research study (Causevic et al., 2010) presents results of an industrial survey on contemporary aspects of software testing. Their study gives crucial information about discrepancies observed between the current practices and the perceptions of respondents which could prove beneficial in shaping future research on software

testing; however, we believe that the explanations for these observed discrepancies were provided based on researchers assumptions, or in some cases the explanations were not defined clearly. The later survey (Lee et al., 2012) investigates the state of software testing practices in terms of software testing methods and tools with a view to identify: current practices, perceived weaknesses and needs for additional capabilities of software testing methods and tools. The research showed that a half of test is executed manually, while a bit less is automated by tools. Comparing test levels by their use, almost the same percentage is devoted for integration testing and system testing, whereas, unit test and acceptance test are not very popular to use.

Furthermore, some researchers noted that while using an appropriate testing technique, test cases creation and prioritization (Elbaum et al., 2002; Rothermel et al., 1999; Srivastava, 2008) are also considered as a crucial part of software testing. Chang, Liao, Chapman, & Chen (2000) provided a novel approach to generate test scenarios based on formal specification and usage profile. In fact, this approach was developed later, and the new framework of formal notation for requirement specification has been presented (Baig & Khan, 2011). The suggested framework should provide a complete software testing technique which is expected to be accurate, structured technique to test software at each step of software development process contrary to existing practice. Although, the research will give statistical results only after completion of the entire three modules of the study as the researcher presented the first, theoretical, part.

In spite of limited resources and rush to finish projects on time project managers are likely to reduce the testing activities (Galin, 2004). In fact, this can bring bad side effects on software quality, therefore to achieve benefit of software testing under limited resources, it becomes necessary to identify the best software testing practices and create a mapping between various existing software methods and tools.

6. CONCLUSION

The main conclusions of the use of software testing techniques in enterprise are made. It is essential to identify the main testing goals, test criteria while selecting the appropriate testing technique; thus such knowledge can help to manage and improve software testing practices effectively and efficiently. Some factors are enumerated that could influence the decisions about which technique is better to choose. The main factors are customer and contractual requirements, time and budget of the project, type of system used and tester's experience. The case studies of testing techniques are generalized. The main features of case studies are identified: almost a half of all faults were related to specification defects; thus more attention should be paid on the first stage of SDLC. Further, a half of test is executed manually, while a bit less is automated by tools. Finally, the need of training related with software testing is agreed by all surveyed enterprises, however, the regular staff training was provided only in some enterprises.

This chapter presents the theoretical framework of research methodology within a specific enterprise. We will introduce to the methodology of empirical study that examines the effectiveness of software testing techniques a specific enterprise. The research methodology, strategy, questions, and data collection methods will be presented as well as their justification and appropriateness to achieve the goal of our research. Further, we provide the validity of research, including the selection criteria of experts, limitations, and ethics. And finally, the main characteristics of selected experts are illustrated in a table.

References

[1]. Manpreet Kaur and Rupinder Singh (2014), “A Review of Software Testing Techniques”, International Journal of Electronic and Electrical Engineering. ISSN 0974-2174, Volume 7, Number 5 (2014), pp. 463-474 © International Research Publication House http://www.irphouse.com

[2]. A. P. Mathur, “Foundation of Software Testing”, Pearson/Addison Wesley, 2008.

[3]. IEEE Standard 829-1998, “IEEE Standard for Software Test Documentation” pp.1-52, IEEE Computer Society, 1998.

[4]. D. Gelperin and B. Hetzel, “The Growth of Software Testing”, Communications of the ACM, Volume 31 Issue 6, June 1988, pp. 687- 695[history of st]

[5]. D. Richardson, O. O’Malley and C. Tittle, “Approaches to specification-based testing”, ACM SIGSOFT Software Engineering Notes, Volume 14 , Issue 9, 1989, pp. 86 – 96[Approaches to specification-based testing]

[6]. S. Rapps and E. J. Weyuker, “Selecting Software Test Data Using Data Flow Information,” IEEE Transactions on Software Engineering¸ April 1985, pp. 367-375

[7]. Harrold Mary Jean, and Gregg Rothermel. "Performing data flow testing on classes." ACM SIGSOFT Software Engineering Notes. Vol. 19. No. 5. ACM, 1994.[acm.pdf]

[8]. Claessen Koen, and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs." Acm sigplan notices 46.4 (2011): 53-64.

[9]. Vilkomir Sergiy A., Kalpesh Kapoor, and Jonathan P. Bowen. "Tolerance of control-flow testing criteria." Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International. IEEE, 2003.[control ieee]

[10]. Ntafos Simeon C. "On comparisons of random, partition, and proportional partition testing." Software Engineering, IEEE Transactions on 27.10 (2001): 949-960.[comparison random]

[11]. Madeyski Lech et al. "Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation." (2013): 1-1.[LR4]

[12]. Jia Yue, and Mark Harman. "An analysis and survey of the development of mutation testing." Software Engineering, IEEE Transactions on 37.5 (2011): 649-678.

[13]. Graves Todd L. et al. "An empirical study of regression test selection techniques." Proceedings of the 20th international conference on Software engineering. IEEE Computer Society, 1998.

[14]. Juristo Natalia, Ana M. Moreno, and Sira Vegas. "Reviewing 25 years of testing technique experiments." Empirical Software Engineering 9.1-2 (2004): 7-44.

[15]. J. A. Whittaker, “What is Software Testing? And Why Is It So Hard?” IEEE Software, January 2000, pp. 70-79[hard software testing]

[16]. Duran, Joe W., and Simeon C. Ntafos. "An evaluation of random testing."Software Engineering, IEEE Transactions on 4 (1984): 438-444.

[17]. Hamlet Dick, and Ross Taylor. "Partition testing does not inspire confidence (program testing)." IEEE Transactions on Software Engineering 16.12 (1990): 1402-1411.

[18]. Li Nan, Upsorn Praphamontripong, and Jeff Offutt. "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage." Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on. IEEE, 2009.[testcritcomp.pdf]

[19]. Hamlet, Richard. "Random testing." Encyclopedia of software Engineering(1994).

[20]. Luo, L. "Software Testing Techniques: Technology Maturation and Research Strategy." Class Report for (2001).

[21]. Jovanavic, Irena. "Software testing methods and techniques." IM Jovanovic is with the Inzenjering, Mat 26 (2008).

[22]. Bluemke, Ilona, and Karol Kulesza. "A Comparison of Dataflow and Mutation Testing of Java Methods." Dependable Computer Systems. Springer Berlin Heidelberg, 2011. 17-30.

[23]. Yoo, Shin, and Mark Harman. "Regression testing minimization, selection and prioritization: a survey." Software Testing, Verification and Reliability 22.2 (2012): 67-120.

IJARIIE