Baltic J. Modern Computing, Vol. 1 (2013), No. 3-4, 210-227

Inventory of Testing Ideas and Structuring of Testing Terms

Ivans Kuļešovs, Vineta Arnicane, Guntis Arnicans, Juris Borzovs

Faculty of Computing, University of Latvia,
19 Raina Blvd., Riga, LV-1586, Latvia

ivans.kulesovs@gmail.com, vineta.arnicane@lu.lv,
guntis.arnicans@lu.lv, juris.borzovs@lu.lv

Abstract. Study includes software testing terms and ideas inventory, software testing high level review and schematization on meta-level and structuring of lower level elements related to the software testing such as testing oracles, testing levels, software quality characteristics, testing approaches, methods, and techniques. Main testing dichotomies are collected and described. An attempt to lay the scientific basis under the proper use of such terms as testing approach, testing method, and testing technique is performed.

Keywords: software testing ideas; testing meta-level; testing approaches, methods, techniques.

1. Introduction

In year 1975, the first theoretic foundation of testing by Goodenough & Gerhart was published. Since those times, theory and practice of testing has evolved quite significantly through emergence of testing activists (Myers, Beizer, Kaner, Bach, Pettichord, Black etc.) and under the influence of different software development approaches (waterfall, rapid application development, agile, etc.).

Despite the attempts of standardization of testing terms and ideas by different authorities, such as ISTQB and IEEE, there is still a little chaos prevailing in the testing literature, and between the testers themselves on the explicit usage and definition of the terms.

An attempt to systemize the main testing ideas and terms ordering them into definite structure has been performed using the tool that adopts the term graph building algorithm developed by one of the publication co-authors (Arnicans, Romans, & Straujums, 2013; Arnicans & Straujums, 2012). ISTQB Standard glossary of terms used in Software Testing (ISTQB Glossary further in text) (ISTQB, 2012) was used as the main source of terms. During the systematization process testing ideas have been selected and divided into eight classes.

During the research process the scope of this publication was enlarged with meta-level review of software testing and introduction of our own way of testing terms systematization to eliminate the misuse or vague use of testing terms that we find confusing. For example such terms as testing approach, testing method, and testing technique are often treated as they have the same meaning, but it is not so. Even in the texts of one author the usage of the terms for the same statements is not systemized. For example, “A **test strategy** or **test technique** is a systematic **method** used to select and/or generate tests to be included in a test suite.” (Beizer, 1995, p.8-9); “… here I present you with ready-made equivalence class partitioning **methods** (or **test techniques**) …” (Beizer, 1995, p.xiv); “**[T]est execution technique:** The **method** used to perform the actual test execution, either manual or automated” (ISTQB, 2012).

Our systematization concept of testing ideas could be useful for making ordered introduction to software testing for fresh minds of novices in testing, while we recognize that in practice it can be very hard to make sharp-cut edges between some of the ideas.

It is worth mentioning that test management is out of scope of this publication.

2. Inventory of Testing Ideas

Inventory of testing ideas has been performed manually, separately by every co-author. After acknowledgment with each other’s inventory results five main parts of the publication have been crystallized: structuring of the testing ideas into eight classes, enlarging main ideas with those emerged through building terms and ideas graphs using the concept map generating tool output results after feeding ISTQB Glossary (ISTQB, 2012) in to it, description and schematization of software testing on meta-level, description of software testing dichotomies, and laying the scientific basis under proper use of such terms as testing approach, testing method, and testing technique.

Testing ideas were divided into the following classes:

- Fundamental ideas.
- How to detect the correctness of the test result?
- How to detect the completeness of the testing?
- How to test (approach, method, technique)?
- What to test (object)?
- Which quality attribute (characteristic) to test?
- When to test (phase)?
- Unclassified.

Authors have identified three millennial fundamental testing ideas. They are:

- Errare humanum est – To err is human.
- Aliena vitia in oculis habemus, a tergo nostra sunt - The vices of others we have in the eyes, in the rear of our own.
- In propria causa nemo judex - No one can be judge in his own cause.

Other testing ideas can be identified through analyzing testing terms, thus testing terms from ISTQB Glossary (ISTQB, 2012) were collected under each related idea class. The same coloring as shown for the ideas classes is used for the terms in the resulting graphs that were built using the tool mentioned above. The graphs produced by tool can be found at link: `http://science.df.lu.lv/kaab13`. Graphs show the testing terms, linkage between each other, and their relation to the definite idea class. It is worth mentioning that work on the testing ideas inventory is still in progress. Resource mentioned above contains the latest inventory results.

3. Software Testing Review on Meta-level

From practical point of view software testing mainly can be expressed by *testing strategy* and *testing tactics* on the meta-level (i.e. on the higher level of abstraction). Contexts of real testing project and theoretical background and experience of testing team influence the selection of the strategy and/ or tactics and the usage of principles of testing schools in the current testing project or campaign. A software testing review on meta-level is depicted in Fig.1.

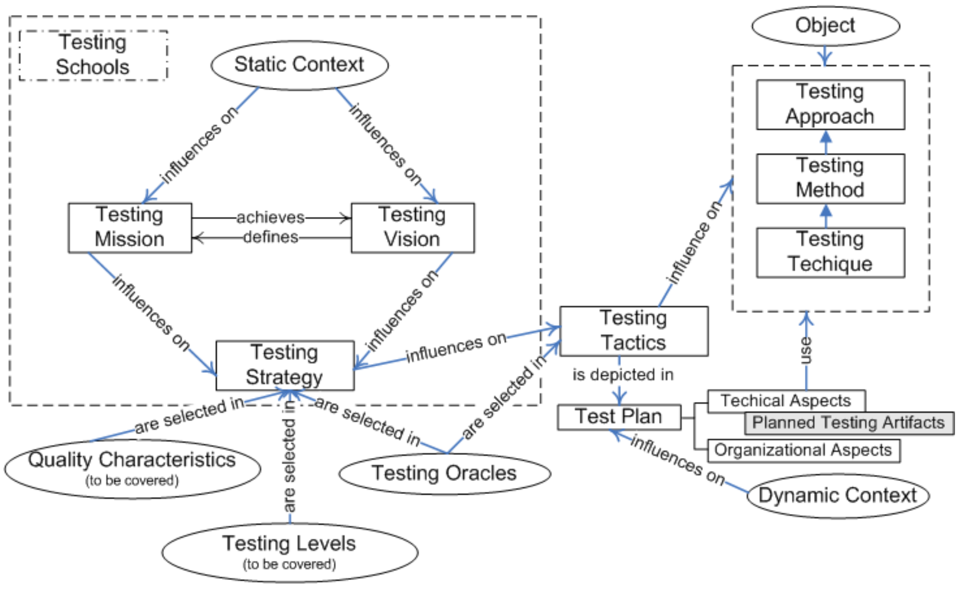

Fig.1. Software testing review on meta-level

Static context influences very much the *testing vision* and *testing mission*. Static context depends on the type of the organization (i.e. governmental, outsourcer, start-up etc.) and on the type of the software produced (enterprise software, commercial software, web page etc.). The options mentioned above are generally static during the whole product lifecycle. *Testing vision* denotes the aims that the testing team wants to achieve by testing. In some cases testing vision can be focused on producing the software with all high and critical software failures discovered and fixed, and 95% of medium severity failures identified. In some cases it can be to receive an acceptance sign-off of the product from the customer. *Testing mission* denotes actions that testing team does in order to achieve the testing vision. For example, the team can use only scripted testing, or it can use the benefits of the exploratory testing as well, to receive an acceptance sign-off of the product from the customer. Or testers prepare automated tests before the development to keep our product always deliverable to the customer as test-driven development suggests. *Testing schools* are theoretical frameworks that define *testing vision* and *testing mission* based on the *static context*.

All aspects of testing schools (it can also be the mix of aspects from different schools) that prevail within the organization and are common for the definite product type influence the testing strategy of the given software project. *Testing strategy* describes a general approach for testing. *Testing strategy* consists of the specification of the roles and responsibilities of each person involved in testing, testing levels, environment requirements, overall testing schedule, testing tools, risks and its mitigations, testing priorities, testing status reporting, etc.

Testing oracles that define testing exit-criteria and those that are used as the source of the derivation of test cases and expected results (i.e. correctness oracles) should be chosen within the testing strategy definition. The selection of *quality characteristics* to be covered by testing process should occur during the definition of testing strategy as well. Test results completeness oracles can be defined when selecting testing tactics, because often there are much more details about expected results amount available during tactics selection process. Testing oracles are in details discussed in the section 6.1.

Dynamic context depends on the project phase and influences the choice of the testing tactics that are appropriate for the given time frame, for the definite object under test, and for the current micro testing goal. Examples of the dynamic context factors are fulfillment of test entry criteria in time, availability of shared testing resources, the stabilization and bug fixing phase of the development etc. *Testing tactics* should be consistent with the testing strategy.

Testing tactics for each object under test are depicted in the *test plan*. Test plan consists of organizational and technical aspects. Testing tactic also influences the choice of the *testing approach* to be used to fulfill the current micro testing goals. Thus, technical aspects of the test plan should include the selection of the appropriate testing approaches, methods, and techniques. Testing artifacts (like test cases, test suites, traceability matrix, test data, etc.) to be produced by the testing process should be mentioned in the test plan as well. It is worth noting that some schools do not require formal and written test plans as a mandatory artifact of testing process.

4. Testing Dichotomies

There are many dichotomies exist in software testing. Some of them clearly belong to definite testing school. Others are opposable because of other reasons, for example project phase. It is worth mentioning that dichotomies mentioned below, despite their difference, make good testing when used together proportionally.

The dichotomy we should start with is ***testing vs. debugging***. The goal of testing is to discover the defect while the goal of debugging is to find why the defect occurs. Some schools see debugging as a job of software developer only, but nowadays it is more common for good test engineer to investigate the root cause of the defect by himself or together with a software developer.

The most known testing dichotomy is ***black-box testing vs. white-box testing***. The difference between them is the point of view on the knowledge of the internal structure of the software that test engineer takes when designing the test cases.

Functional testing vs. non-functional testing is another important testing dichotomy. Functional testing "*verifies a program by checking it against ... design* *document(s) or [functional] specification(s)*" (Kaner, Falck, & Nguyen, 1999, p.52). Non-functional testing checks software against its non-functional requirements where non-functional quality characteristics are addressed. System testing is different from functional testing because it "*validate[s] a program by checking it against the published user or system requirements*" (Kaner et al., 1999, p.52).

Another quite old dichotomy is ***manual testing vs. automated testing***. Return on investment is taken into consideration when testing is automated, as it requires skilful workforce and additional scripting and maintenance effort. Still, only part of the testing can be automated. UI automation is often used for regression testing, while unit and integration tests can be written in advance to development.

These two testing ideas are very different by their nature: ***scripted testing vs. exploratory testing***. Scripted testing can show the thoroughness of the testing to stakeholders, while exploratory testing can find failures that hardly could be discovered when using scripted testing, because it is sometimes even hard to imagine the appropriate test cases before investigating the behavior of the new functionality under test.

Another dichotomy consists of two of the oldest testing ideas: ***verification vs. validation***. Controversy ***contract vs. client happiness*** is closely connected to the testing missions represented above, thus, depending on this, different testing strategies are chosen. Verification evaluates if product meets the requirements that usually are part of the contract while validation check if product satisfy the clients (or other stakeholders) expectations, i.e. makes them happy.

Positive testing vs. negative testing dichotomy both parts are necessary if there is an aim to make testing as complete as possible. Positive testing tends to prove that software behaves in the way it is supposed to. Negative testing shows that software does not do that it is not supposed to.

Testing of design vs. testing of implementation identifies different testing needs depending on the software project phase. Thus, different testing tactics can be used during each phase. Testing of design also uncovers the idea that testing should be started as early as possible.

Static testing vs. dynamic testing dichotomy intersects with previously mentioned dichotomy. Testing of designs is always a static testing, i.e. testing process without executing the software itself. Testing of implementation (except the review of the code) in most cases is a dynamic testing, i.e. testing of the running software.

Hierarchical vs. big bang are different approaches of the integration testing. There are two hierarchical integration testing approaches: bottom-up and top-down. When bottom-up approach is used then testing is started from the components on the lowest level and goes up to the testing of the integration of the next level components. Integration testing between top level components is the first point of the top-down approach. It goes to the lower level components testing afterwards till the lowest level is reached. On the contrary, integration on all levels occurs simultaneously when big bang integration testing approach is used.

The last, but not least software testing dichotomy that we would like to mention is ***traditional testing vs. agile testing***. Agile school has completely different mission then other ones and it discovers the role of software engineer in test as the great automation specialist and the main participant of the test driven development.

5. Testing Schools

Testing society distinguishes five testing schools (Pettichord, 2007). They are:

- Analytic School;
- Standard School;
- Quality School;
- Context-Driven School;
- Agile School.

The schools are frameworks for categorization of test engineers' believes about testing and are their guide on the testing process. Testing schools are not competitive; they can be used in the collaborative mode as well. They all have exemplar techniques or paradigms, but they are not limited to them. Usage of schools can vary within the organization from project to project, but it is often hard to move the whole organization from one school to another.

The *analytic school* assumes that software is a logical artifact. It concentrates on technical aspects, and it is keen on the white-box testing. Analytic school is associated with academia institutions, and it is assumed to be the most suitable for safety-critical and telecom software.

The *standard school* assumes that testing should be very well planned in advance and managed. According to this school, testing main goal is to validate that software meets contractual requirements and/or governmental standards using the most cost-effective model, thus it is mostly applied for governmental and enterprise IT products. Requirements traceability matrix is the most common testing artifact for the school. Software testing can be seen like assembly line through V-model prism. IEEE standards' boards and testing certifications are the most valued institutions by this school.

The *quality school* prefers "Quality Assurance" over "Testing". Thus testing defines and controls the development processes. QA manager or test lead is like a gatekeeper who can decide if software is ready or not. ISO and CMMI are the most valued institutions for followers of this school.

The *context-driven school* concentrates about (skilled) people and their collaboration. The goal of the context-driven testing is to find bugs that can bother any of the stakeholders. What to test right now is defined according to the current situation in the project. Test plans to be constantly adapted based on the test results. Exploratory testing is an exemplar technique of this school. Context-driven testing is mostly applied for the commercial and market-driven software. The Los Altos Workshop on Software Testing held by Cem Kaner and Brian Lawrence are thought to be the main events of this school.

The *agile school* main postulate is that tests must be automated. Testing answers the question if user story is done. Test-driven development is one of the agile testing school paradigms, thus unit tests are demonstrative exemplar of the school.

It is worth mentioning that categorization of beliefs and testing goals into testing schools helps testers to understand and to evaluate each other experience through the prism of the specific organizational context.

6. Testing Strategy

There are many things to be covered in the testing strategy that define the overall approach for testing. Here we only look into the details of its most important aspects that are noticeable on the software testing meta-level.

6.1. Testing Oracles

Testing oracles can be divided into three major groups based on their purpose. Groups of oracles and representatives per each group are shown in Fig.2.

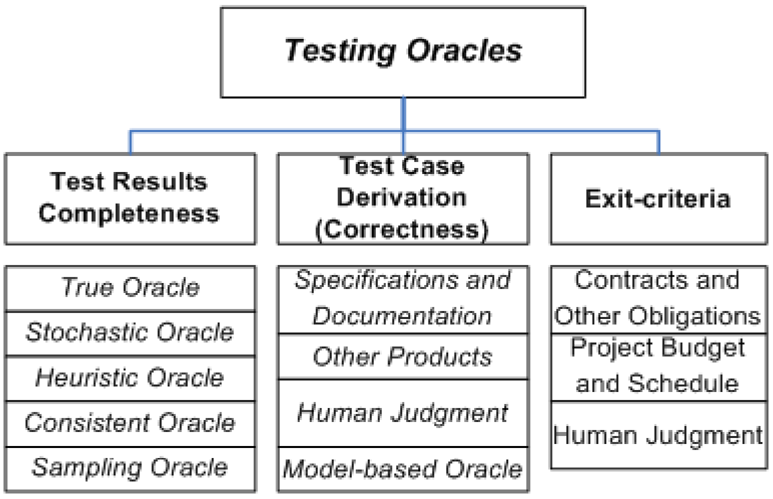

Fig.2. Testing Oracles (partially adapted from Hoffman, 1998)

Test results completeness oracles are differentiated based on the completeness of the set of the expected test results. There are five main types of test result completeness oracles. (Hoffman, 1998) They are:

- True oracles – they have the complete set of expected test results.
- Stochastic oracles – they verify a randomly selected sample.
- Heuristic oracles– they can verify the correctness of some values and the consistency of other values.
- Consistent oracles – they verify current test run results with previous test run results (regression).
- Sampling oracles – they select the specific collection of inputs or results.

They all have their advantages and disadvantages, as well as their cost decreases in a top-down manner, but speed increases in the same manner.

Test case derivation (correctness) oracles are differentiated based on the source test cases and expected results are derived from.

Exit-criteria oracles define when testing can be finished. The most common, but not complete testing exit-criteria are:

- All planned test cases are executed (Contracts and other obligations).
- All high and critical priority bugs are fixed (Contracts and other obligations).
- All planned requirements are met (Contracts and other obligations).
- Scheduled time to finish testing has come (Project budget and schedule oracle).
- Test manger has signed off the release (Human judgment oracle).

It is worth mentioning that multiple oracles of each group are often used together depending on the software project phase.

6.2. Quality Characteristics

There are 8 quality characteristics shown in the new revision of ISO/IEC 9126 standard - ISO/IEC 25010 (ISO 2011). All quality characteristics are depicted in Fig.3a and b.

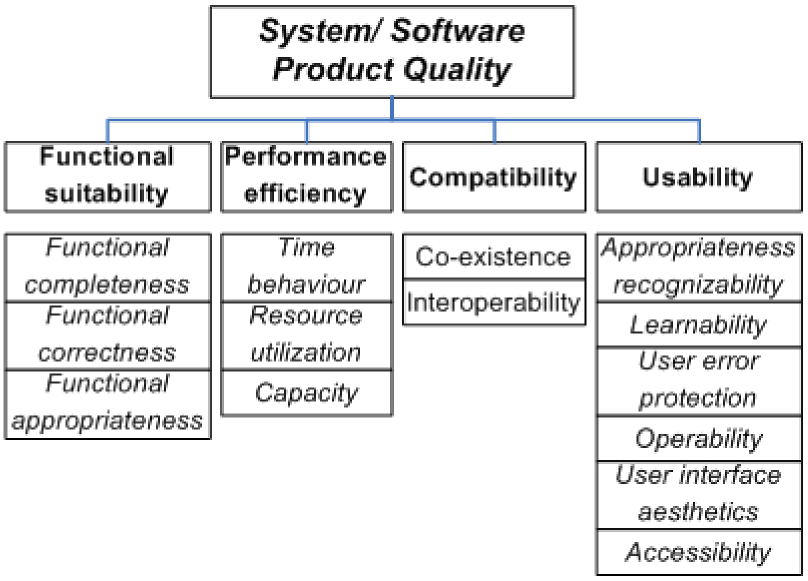

Fig.3a. Product Quality Model (ISO, 2011)

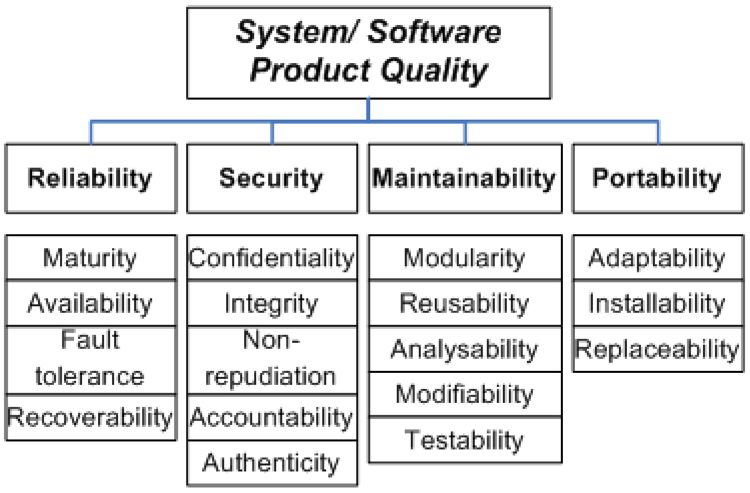

Fig.3b. Product Quality Model (continued) (ISO, 2011)

Functional testing is a testing of functional suitability characteristic. Almost all formal testing methods and techniques are concentrated around functional suitability quality characteristic as well, and especially are related to the functional correctness and functional completeness sub-characteristics.

6.3. Testing Levels

There are four main *testing levels* differentiated in the software development project. Their applicability differs based on the project phase and the scale of the object under test. These levels are: (Black, 2009)

- Unit Testing – testing of single component on the code level; it is usually performed by developers.
- Integration Testing – testing of cooperation of several components; comparison with expected result can be done both on the code level and manually by human; can be performed either by developer, or by tester.
- System Testing – testing of the whole complete system; usually is performed by tester.
- Acceptance Testing - testing of the whole system to verify that it meets some contract obligation and/ or satisfies users' expectation about the software product; usually is performed by the customer.

All levels starting from integration testing can be scaled out till the system of systems testing when product consists of or is dependent on multiple systems.

7. Testing Tactics

Testing tactics can differ depending on the phase of the project and other changeable circumstances of the environment. *Testing tactics* should be consistent with the testing strategy. Thus tactics often are chosen within the static boundaries of the influencer schools. Appropriate testing approaches, methods, and techniques should be selected for micro testing goals fulfillment and should be depicted in the test plan. *Testing artifacts* (like test cases, test suites, traceability matrix, test data, etc.) to be produced by the testing process should be mentioned in the test plan as well. We have structured testing methods and techniques under *black-box* and *white-box* approaches. The borders of grey-box testing approach are quite ambiguous, and methods and techniques under this approach are not formally described yet in the testing theory. They do not have settled definitions in the testing practice as well.

It is worth mentioning that we respect other software testing systematization concepts, for instance, division of all methods and techniques under 4 coverage approaches (Graph Coverage, Logic Coverage, Input Space Partitioning, and Syntax-Based Testing) by Ammann & Offutt (2008), but we still have not found the explicit difference between testing approach, method, and technique in other concepts.

7.1. Testing Artifacts

Software testing usually produces testing artifacts mentioned below:

- Test Data – multiple sets of values to be used as inputs for testing definite functionality often combined into one file.
- Test Script – code that substitutes user activity and/or interaction with software UI.
- Test Case – consists of preconditions, steps, inputs, and expected results to test some part of the functionality.
- Test Scenario – test case with higher level of abstraction that depicts scenarios in which user is considered to use the software.
- Test Suite – the set of test cases or test scenarios for given functionality or testing type (i.e. regression, smoke, or sanity).
- Test Plan – document that depicts testing tactics to test definite software product in the definite testing run; often consists of the test suites to be executed and the testing approach to be used.

Traceability matrix is the example of cross-referring document that can be used to depict the relations between test cases/test scenarios/ test suites (depending on the scale) and requirements.

Test harness is a virtual, to testing related artifact that consists of many aspects to make testing under given conditions and configurations possible. It can consist of the specific IT infrastructure, tools, big samples of test data etc.

Despite the fact that testing artifacts mentioned above to be produced during the whole testing process lifecycle, the high level description of the approach to be used to produce them to be defined in the testing strategy.

7.2. Systematization of Testing Terms: Approach, Method, and Technique

The connection and clear border between *testing approach*, *testing method*, and *testing technique* are not defined in the testing theory. For example, Beizer (1995, p.8-9) defines test technique as a systematic method: “A **test strategy** or **test technique** is a **systematic**

method used to select and/or generate tests to be included in a test suite.” In the same time, he uses test technique and test method as completely equal statements: “… here I present you with ready-made **equivalence class partitioning methods** (or **test techniques**) …” (Beizer, 1995, p.xiv); “**[T]est execution technique:** The **method** used to perform the actual test execution, either manual or automated” (ISTQB, 2012). Other authors, such as Kaner et al. (1999), Pressman (2005), and Sommerville (2007) have a mix of using words technique, method, approach, and strategy in regard to testing as well.

The attempts of making a distinction between approach, method, and technique were already performed by language teaching specialists in 1963, 12 years before the first theoretic foundation of testing by Goodenough & Gerhart was published. In 1963 Anthony provided “much needed coherence to the conception and representation of elements that constitute language teaching:” (as cited in Kumaravadivelu, 2006)

An *approach* is “a set of correlative assumptions dealing with the nature of language and the nature of language teaching and learning. It describes the nature of the subject matter to be taught. It states a point of view, a philosophy, an article faith…”

A *method* is “an overall plan for the orderly presentation of language material, no part of which contradicts, and all of which is based on the selected approach. An approach is axiomatic, a method is procedural”.

A *technique* is described as “a particular trick, stratagem, or contrivance used to accomplish an immediate objective”.

"The arrangement is hierarchical. The organizational key is that techniques carry out a method which is consistent with an approach."

In 1982 Richards & Rogers (as cited in Kumaravadivelu, 2006) performed an attempt to enhance the framework developed by Anthony through dividing language teaching process into approach, design, and procedure. But, despite rather vague definition of terms *approach*, *method*, and *technique*, and not considering in any way of complex connections between them, exactly these terms are in favor of the most current teacher training manuals. (Hall, 2011)

We suggest systemizing testing approach, testing method, and testing technique in the same hierarchical way, using the experience and keeping in mind the mistakes of language teaching specialist. Schematic relation between terms mentioned above is shown in Fig.4.

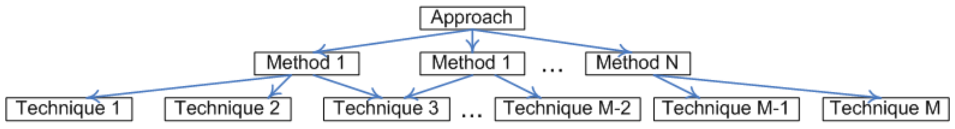

Fig.4. Relation between approach, method, and technique

Testing approach “states a point of view, a philosophy, an article faith” that a test engineer takes when designing test cases.

Testing method is *“an overall plan for the orderly presentation”* of testing techniques.

Testing technique is *“a particular trick, stratagem, or contrivance”* to design the test case.

Testing techniques are united under testing methods based on test case design formality (for black-box testing approach) or based on other common pronounced attributes (for white-box approach).

The *"organizational key"* stays the same as suggested by Anthony – "*techniques carry out a method which is consistent with an approach*".

7.3. Black-box Testing

Black-box is a software testing approach when test engineer designs test cases as if she does not know anything about the internal structure of the software under test.

Black-box testing approach consists of seven testing methods that are differentiated based on the source used for test case design process and based on the level of formality of test case designs. The relation between black-box testing methods and techniques is shown in Fig.5.

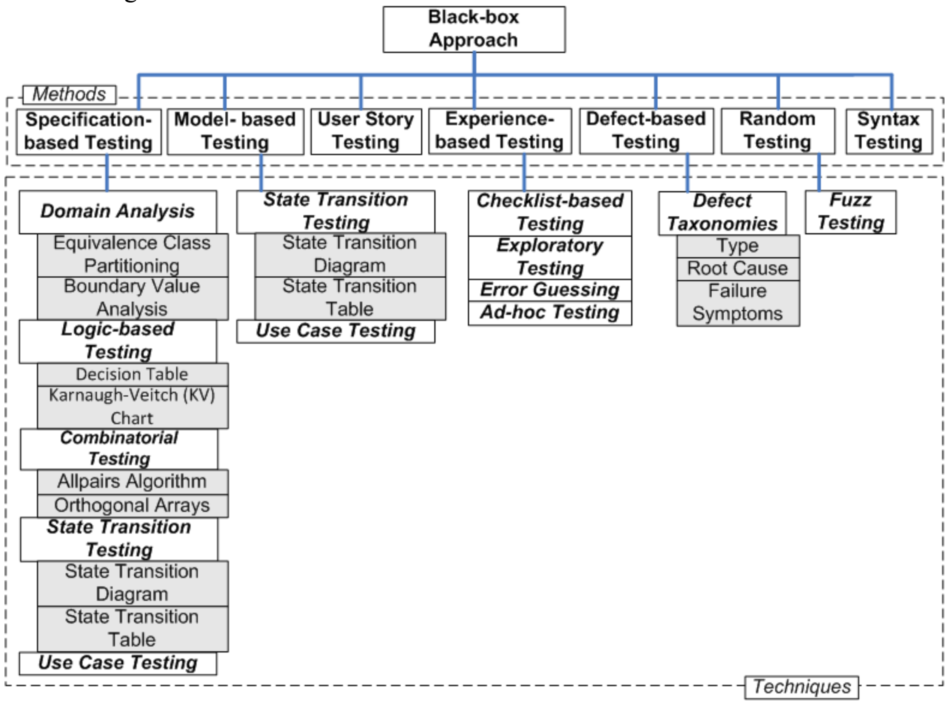

Fig.5. Black-box Approach

Specification-based testing is a testing method which includes all formal test case design techniques. As can be derived from the name of the method, specification (or requirements) documents are used as a source for test case design. Formal test case design techniques or groups of techniques are Domain Analysis, Logic-Based Testing, Combinatorial Testing, State Transition Testing, Use Case Testing, and Syntax Testing.

Domain analysis group consists of two closely connected testing techniques: Equivalence Class Partitioning and Boundary Value Analysis. The first technique defines the group (class) of inputs that produces the same output. The second technique checks the boundary values of the equivalence classes.

Logic-based testing group consists of two testing techniques: Decision Tables and Karnaugh-Veitch (KV) Charts. They all are used when combination of different inputs results into specific output. They are used for checking business logic and user interface. According to Copeland (2003), a decision table consists of conditions, combinations of every condition alternatives that result into single rules, actions, and actions occurrence under every rule. It is worth mentioning that cause-effect graphing can also be used for designing decision tables according to Myers (1979/2004).

KV charts are used to simplify the Boolean algebra expressions. They were introduced by E. Veitch in 1952 and improved by Karnaugh in 1953. They allow decreasing the amount of calculation needed through humans' pattern-recognition capability (Beizer, 1990). From our experience usage of decision tables is more common technique in the field of business application testing, especially nowadays.

State Transition Testing is a group of techniques that are used when some part of the functionality of the system can be represented as “finite-state machine”. Finite state machine is an abstract machine that has finite states, that can be in only one state at once, and whose transitions from one state to another are triggered by some event or condition. There are two common techniques that are used for state transition testing: State Transition Diagrams and State Transition Tables. State transition diagram is a schematic representation of machine’s states and transitions between them. State transition table is more complete and systematic way of representation of the same machine’s states and transitions. Only valid state-transition combinations are depicted in the state transition diagrams, while all possible state-transition combinations are covered in the state transition tables that can be required for testing safety-critical software. (Black, 2009)

Combinatorial Testing is a group of testing techniques that are most often used for testing combinations of configurations or input parameters. The most popular techniques are Orthogonal Arrays and Allpairs Algorithm. Orthogonal array is a two-dimensional array that has an interesting property – “all the pairwise combinations will occur in all the column pairs” (Copeland, 2003, p.66). This part of discrete math was introduced into testing field by Tatsumi in 1987. Allpairs algorithm invented by Bach allows achieving the coverage of testing of all pairs combination with less steps when input parameters have different number of possible values (Kaner, Basch, & Pettichord, 2001).

Use Case Testing is a technique that allows to test system’s functionality that is described as a use case. Use case is a type of quite detailed specification that concentrates on user (or another system) interaction with the system under test to complete some specific task or to deliver some other business value. It often has a main, the most commonly used flow and extensions or some special cases. The test scenario for main flow and every extension or special case should be created when use case testing is performed. Use case can be described using natural language or depicted using different modeling languages, for example UML.

Model-based testing is a testing method which unites testing techniques that use similar software, or software prototype, or software usage models as the basis for test cases design. The main representatives of this method are previously described state transition testing and use case testing (when use case is described using different modeling languages).

User story testing method includes acceptance testing techniques in combination with exploratory testing techniques that are described later. User story is a way of a non-detailed software specification that describes it using the mask "As an <actor> I want (or need) <action> so that <achievement>" (in practice, sometimes <achievement> part is not formally specified). User story must come to the development team together with acceptance criteria to align the constraints of the business value to be delivered. User stories are mostly used when software is developed using such Agile software development practices as Scrum, Kanban, and XP. Acceptance tests are executed to verify if implemented user story meets the acceptance criteria. More thorough testing using exploratory testing techniques is performed after acceptance criteria is met. Sometimes, depending on the complexity of the system, usage of more formal testing techniques also takes place.

Experience-based testing method unites less formal testing techniques, but some of them are still very powerful when are used by professionals. These techniques are Checklist-based Testing, Exploratory Testing, Error Guessing, and Ad-hoc Testing.

Very high level checklist of quality attributes or items that are important for the system under test is used for checklist-based testing. Such list should be constantly improved to cover things that are important to some of stakeholders or are parts of some regulation standard (for example, operating system UI guidelines) while product is evolving during the development process.

Test engineer intuition and experience to evaluate the test results are the basis of the exploratory testing technique. The design of new test cases occurs on the fly using the information discovered from the testing of the software itself. Exploratory testing to be productive must be performed in definite time frames and the scope of testing must be defined in advance. Test charters are often used to make these two "musts" possible and also show the productivity of the testing session to the stakeholders by notifying its results. Such exploratory testing management was developed by Jonathan and James Bach in 2000. They named it session-based testing, but we suppose that exploratory testing without clearly defined objectives and time frames is ad-hoc testing that is the least formal testing technique of the experience-based testing method. Usage of ad-hoc testing technique should be avoided. (Black, 2009)

Error-guessing is a testing technique that uses most common programming errors as test case basis. Examples of such errors are null pointers, division by zero, wrong types of parameters etc. Even if tester does not have knowledge of programming she will often discover such errors while testing the software and will reuse this experience afterwards. That is why this technique is part of experience-based testing method. In most cases error-guessing is used as informal supplementary of formally scripted testing techniques.

Defect-based testing method uses the knowledge about defects taxonomies for test cases design or selection. According to Beizer, there are eight categories to be used for defects classification: Functional, System, Process, Data, Code, Documentation, Standards, and Other. There are also five supplementary categories to be used for defects housekeeping: Duplicate, Not a problem, Bad Unit, Root cause needed, Unknown. Black (2009) uses the same defects classification. From our experience this method can be hardly used independently for test cases design. It can only point out which test cases may lead to more defects discovery based on the historical data if it is available. What is more, such analysis of defects is quite expensive and such bookkeeping options are supported by only few tools by default.

Random testing method uses randomly generated inputs from the definite subset as test data. It can be a powerful method for functional testing when operational profile (input domains) of the system and effective oracle are available. In such cases systems are tested with condition that whole test fails if it fails on at least one of the inputs. But in real situation the options mentioned above are hardly available. Even if uniform distribution can be applied to the input values, it is very hard to substitute the effective oracle for outputs. That is why random testing is mostly used for reliability testing of the complex systems. It can prove that system can work without failures for given amount of time (Hamlet, 1994). When reliability of the system is tested with totally random input values it means that Fuzz Testing technique is applied.

Syntax testing is a static, black box testing method for testing syntactic specification of a system's (or protocol's) input values. "Anti-parser" can be used to compile the grammar to produce "structured garbage". This "structured garbage", that can contain misplaced or missing elements, illegal delimiters, and so on, is used to test how object under test behaves when inputs deviate from the defined syntax. (Beizer, 1990)

7.4. White-box Testing

White-box is a software testing approach when test engineer designs test cases based on the internal structure of the software under test. There are three most known white box testing methods: control flow testing, data flow testing, and mutation testing. The relation between the white-box testing methods and techniques is shown in Fig.6.

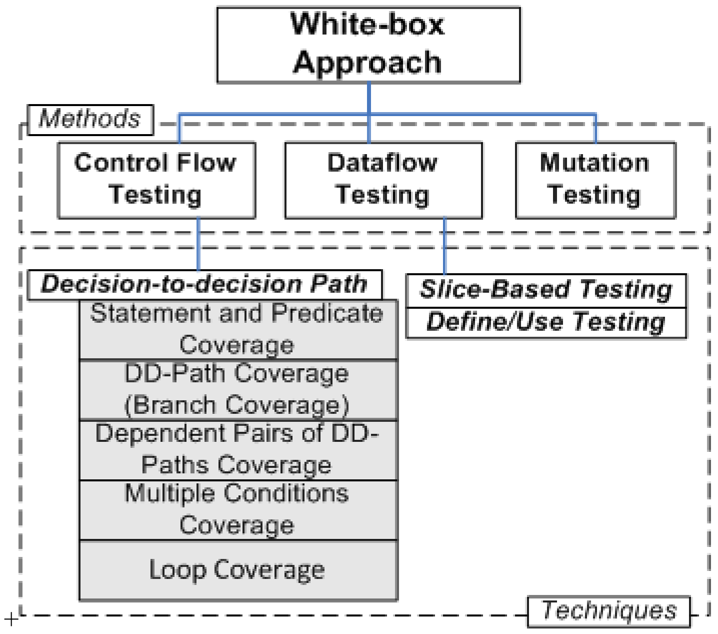
White-box Approach
Methods
Control Flow Testing
Dataflow Testing
Mutation Testing
Decision-to-decision Path
Statement and Predicate Coverage
DD-Path Coverage (Branch Coverage)
Dependent Pairs of DD-Paths Coverage
Multiple Conditions Coverage
Loop Coverage
Slice-Based Testing
Define/Use Testing
Techniques

Fig.6. White-box Approach

Control flow testing concentrates about testing the sequence of the statements in which system under test operates. There are two main programming paradigms that influence the statements' sequence execution. They are conditions and loops. The main technique of control flow testing is called Decision-to-Decision Path Testing (Jorgensen, 2008). Decision-to-Decision path testing technique uses program graph to represent all possible statements (graph nodes) and conditions (graph edges). Coverage of different code aspects can be checked when using this technique.

Dataflow testing method concentrates about the points of program graph where variables receive values and where these variables are used. Thus dependent pairs of the DD-paths coverage of previously mentioned Decision-to-Decision Path Testing technique is most efficient exit criteria for such testing method while the whole lifecycle of the variable is monitored.

Mutation testing method is used to prove that the set of unit tests that pass actually is correct and complete. Mutation (i.e. wrong peace of code) is introduced into the program itself. For example, operators or commands execution order can be changed, or even some code can be removed. If unit tests still pass after mutation introduction then it means that some of the unit tests are wrong or that mutated code is never executed.

Some static testing techniques are used for software code testing. They differ based on the formality and thoroughness of the process. Code review is often used to improve the overall quality of the code and to educate less experienced developers. This process helps to deliver more qualitative and tested code from development to testing right at the moment, but educative aspects help to improve the quality of the code for the future deliveries. Inspections and walkthroughs are used when there is less time available to conduct the static testing process.

8. Conclusions

Inventory and structuring of testing ideas and terms has resulted into discovering of eight classes of the testing ideas. Initiation of such process has helped to understand the need of making the clear definition of such terms as testing approach, testing method, and testing techniques that has been achieved using the solution made by Anthony in the field of language teaching. Structuring of the ideas have also made it possible to schematize the software testing on meta-level, defining the relation between such concepts as testing strategy, testing tactics, testing schools, testing mission, testing vision, different (organizational and project-wide) contexts, testing approach, testing method, testing technique, testing plan, etc.

As a further work we see the need in providing more observable relation between the categorization of the testing ideas and software testing review on meta-level, and between them and software testing dichotomies. Categorization of testing terms between testing ideas classes should be continued as well.

Acknowledgments

I. Kuļešovs wishes to express sincere thanks to his wife Jekaterina for support and patience during the article preparation.

This paper partially has been supported by the European Regional Development Fund Project No. 2DP/2.1.1.3.1/11/APIA/VIAA/010.

References

Arnicans, G., Romans, D., & Straujums, U. (2013). Semi-automatic Generation of a Software Testing Lightweight Ontology from a Glossary Based on the ONTO6 Methodology. In: Caplinskas, Albertas, et al. (eds.), *Frontiers in Artificial Intelligence and Applications. Databases and Information Systems VII: Selected Papers from the Tenth International Baltic Conference, DB&IS 2012*, Vol. 249, IOS Press, 263-276.

Arnicans, G. & Straujums, U. (2012), Transformation of the Software Testing Glossary into a Browsable Concept Map, *International Conference on Engineering Education, Instructional Technology, Assessment, and E-learning (EIAE 12); International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE 12), December 7 - 9, 2012.*

Amman, P. & Offutt, J. (2008). *Introduction to Software Testing*, Cambridge: Cambridge University Pres.

Beizer, B. (1990). *Software Testing Techniques*, 2nd Edition, New York: Van Nostrand Reinhold Co.

Beizer, B. (1995). *Black-Box Testing: Techniques for Functional Testing of Software and Systems,* New York: John Wiley & Sons, Inc.

Black, R. (2009). *Advanced Software Testing – Vol.1,* Santa Barbara, CA: Rock Nook Inc.

Copeland, L. (2003). *A Practitioner's Guide to Software Test Design,* Norwood, MA: Artech House, Inc.

Goodenough, J. & Gerhart, S. (1975). Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering, Volume 1 (2),* 156-173

Hall, G. (2011). *Exploring English Language Teaching: Language in Action.* New York: Routledge.

Hamlet, R. (1994). Random Testing, in Marciniak, J., ed., *Encyclopedia of Software Engineering*, Wiley, Chichester, 970-978.

Hoffman, D. (1998). A Taxonomy for Test Oracles, *in Quality Week*, 1998.

ISO (2011). *ISO/IEC 25010:2011*.

ISTQB (2012). *Standard glossary of terms used in Software Testing*. In: van Veenendaal, E. (Ed.), available at *http://www.istqb.org/downloads/finish/20/101.html*.

Jorgensen, P.C. (2008). *Software Testing: A Craftsman's Approach,* 3rd Edition, Boca Raton, FL: Auerbach Publications.

Kaner, C., Falck, J., & Nguyen, H. (1999). *Testing Computer Software,* 2nd Edition, John Wiley & Sons, Inc.

Kaner, C., Basch, J. & Pettichord, B. (2001). *Lessons Learned in Software Testing: A Context-Driven Approach,* New York: John Wiley & Sons, Inc.

Kumaravadivelu, B. (2006). *UNDERSTANDING LANGUAGE TEACHING: From Method to Postmethod,* Mahwah, NJ: Lawrence Erlbaum Associates, Inc.

Myers, G. (1979/ 2004), *The Art of Software Testing,* 2nd Edition, Hoboken, New Jersey: John Wiley & Sons, Inc.

Pettichord, B. (2007). *Schools of Software Testing,* available at *http://www.prismnet.com/~wazmo/papers/four_schools.pdf.*

Pressman, R. (2005). *Software Engineering: A Practitioner's Approach,* 6th Edition, Singapore: McGraw-Hill.

Sommerville, J. (2007). *Software Engineering,* 8th Edition, Harlow, Essex: Pearson Education Limited.

Tatsumi, K. (1987). Test Case Design Support System, *Proceedings of International Conference on Quality Control (ICQC),* Tokyo, pp. 615-620.

Authors' information

Ivans Kuļešovs is a PhD student in Computer Science at the University of Latvia and Test Manager in C.T.Co Ltd., a software development company. His research interests include software testing in general and mobile applications testing in particular, as well as enterprise mobility platforms.

Kuļešovs received his master degree with distinction in Computer Science from University of Latvia and MBA degree from Blekinge Institute of Technology, Sweden.

Vineta Arnicane is a Senior Researcher in the Faculty of Computing at the University of Latvia. Her research interests include software engineering, software testing, and artificial intelligence. Arnicane received her PhD in computer science from the University of Latvia.

Guntis Arnicans is a Professor and Director of Bachelor program "Computer science" in the Faculty of Computing at the University of Latvia. His research interests include software engineering, software testing, and artificial intelligence, with a focus on creating concept map and ontology for software testing domain. Arnicans received a PhD in computer science from the University of Latvia. He is a member of IEEE and ACM.

Juris Borzovs is currently Professor and Dean of the Faculty of Computing at the University of Latvia. His research interests include software engineering, software quality, software testing, and IT terminology. Borzovs received his candidate of science degree from the Institute of Mathematics of Belarusian Academy of Science, doctor of science degree and doctor habilitatus degree from the University of Latvia. He is a member of several organizations that focus on information technology.

Received August 15, 2013, revised November 13, 2013, accepted November 13, 2013