ISSN No. 0976-5697

Volume 5, No. 7, September-October 2014

International Journal of Advanced Research in Computer Science

REVIEW ARTICLE

Available Online at www.ijarcs.info

Validation Tools in Software Testing Process: A Comparative Study

Alireza Jomeiri
Department of Engineering
Marand Branch, Islamic Azad University
Marand, Iran

Abstract: Software testing provides a means to reduce errors, cut maintenance and overall software costs. Testing has become most important parameter in the case of software development lifecycle (SDLC). Software testing tools enables developers and testers to easily validate the entire process of testing in software development. It is to examine & modify source code. Effective Testing produces high quality software. In this paper, we evaluate popular validation tools in software testing process. We compare these tools with respect to the variety of testing types such as unit testing, integration testing, functional testing, system testing, performance testing, stress testing and acceptance testing. In order to facilitate testing tools description and provide a support for test engineers in selecting correct set of instruments according to their tasks, one can use a tools classifier. This means that by providing necessary information regarding the system under test (SUT), required testing type to perform and other details, a test engineer can get an output of possible testing tools that match concrete criteria. At present time some classifiers are available. The classifier used in this paper can be employed in appropriate choice of testing tool or set of tools for a software project. On the one hand it can be helpful for orientation in the wide subject field of software testing, reducing the amount of time required for specialists to find a proper solution. On the other hand it can be used as a quick introduction to a fast-developing area of testing and currently available testing tools for non-experts in this field.

Keywords: SDLC ; Software Testing ; Validation Tools ; Classifier.

I. INTRODUCTION

The current section provides a presentation of validation activities and classification of supportive software tools, as the justification for the second part of software testing definition.

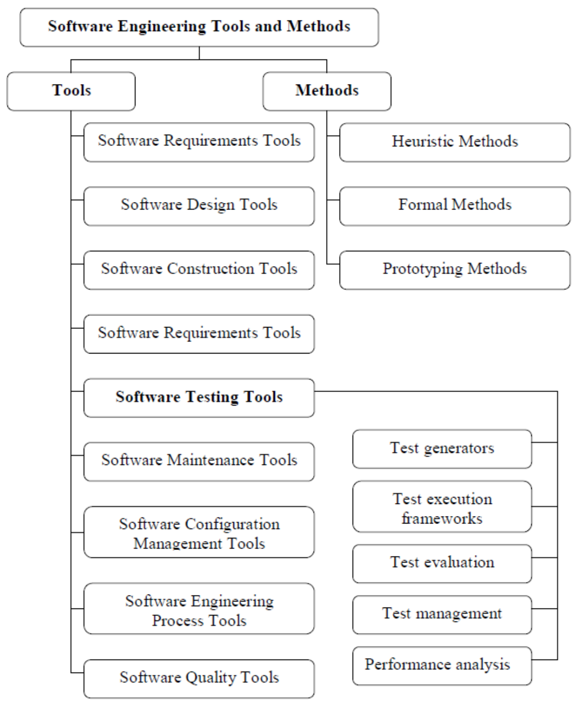
Software testing tools, as a part of software engineering tools (Figure 1), are computer-based tools for assisting software lifecycle processes. Software testing tools allow periodic and defined actions to be automated, reducing the repeated load on the software engineer and allowing concentrating on creative aspects of the process. Both testing tools and methods make software testing more systematic.

Tools are often designed to support one or more software testing methods and are varying in scope from supporting individual tasks to covering the complete testing cycle. As it has been mentioned above, validation checks conformance of any artifacts, which have been created or used during development or maintenance, with user or customer needs and requirements. These requirements can be documented, and correspondingly testing tools can be used for automation of testing activities. These tools are summarized in Table 1.

Table I. Test automation tools classification according to [1]

Tool type	***Description***
Test generators	Assist in test cases development.
Test execution frameworks	Provide execution of test cases in a controlled environment where the behavior of tested artifact can be observed.
Test evaluation	Support the evaluation of test execution results and determine whether or not it conforms to the expected results.
Test management	Support for all of the testing process's aspects.
Performance analysis	Quantitative measuring and analyzing of software performance in order to assess performance behavior rather than correctness.

Figure 1. Software Engineering Tools and Methods [1]


Software Engineering Tools and Methods
Tools
Methods
Software Requirements Tools
Software Design Tools
Software Construction Tools
Software Requirements Tools
Software Testing Tools
Software Maintenance Tools
Software Configuration Management Tools
Software Engineering Process Tools
Software Quality Tools
Heuristic Methods
Formal Methods
Prototyping Methods
Test generators
Test execution frameworks
Test evaluation
Test management
Performance analysis

The last item of tool classification – performance analysis, illustrates to some extent the insufficiency of the classification available in SWEBOK. It fails taking into consideration, for example, functional testing tools, security testing tools, user interface testing tools, stress testing tools and others, which correspond to the objectives of testing as described in SWEBOK IEEE Guide to Software Engineering

Body of Knowledge section 2.2 of Chapter 5 "Software Testing". Each of the mentioned tool type can be a subtype of possible particularized or special testing tools.

In this paper, the classification is adopted from [2], where the validation activity is divided into unit testing, integration testing, functional testing, system testing, and acceptance testing.

II. TESTING TOOLS CLASSIFIER

In order to facilitate testing tools description and provide a support for test engineers in selecting correct set of instruments according to their tasks, one can use a tools classifier. This means that by providing necessary information regarding the system under test (SUT), required testing type to perform and other details, a test engineer can get an output of possible testing tools that match concrete criteria.

Automated test model

At present time some classifiers are available. A classifier derived from [3] is based on a model of automated test (Figure 2). This classifier is supposed for test automation tools. Test automation can be defined as an activity when software tester just executes a test and analyses the results [4].

The automated test model used in this classifier is general enough, so a test engineer can utilize it for modelling various tests, which require automation. This classifier is convenient, as the belonging of some tool to a particular group is easy to evaluate, and it provides an unambiguous classification results. By using the classifier, software tester can obtain a tool or a list of tools that is most suitable for concrete tasks.

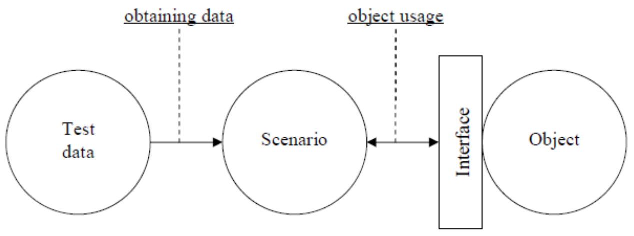


Figure 2. Automated test model schema [3].

In this model testing software is divided into testing scenario and test data. Scenario can be treated like a program, which includes usage of an object under test, response correctness checking and other activities, required for evaluating an object. Test data is used in the scenario for running specific test cases. Test data can be divided into input data, expected output data and auxiliary data.

An object can be a code fragment, a unit or a complete system. Scenario is interacting with an object via object's interface. For example, calling object's operations or checking output correctness. Scenario is obtaining data from some source, but cannot modify it, since the data is defined separately or is generated during a test case.

Classifier's criteria

The resulting classifier uses four criteria:

1. scenario's data acquiring method (marked D);
2. scenario construction type (S);
3. interaction with an object method (M);
4. object's interface type (I).

Each criterion's possible values are depicted at Figure 3.

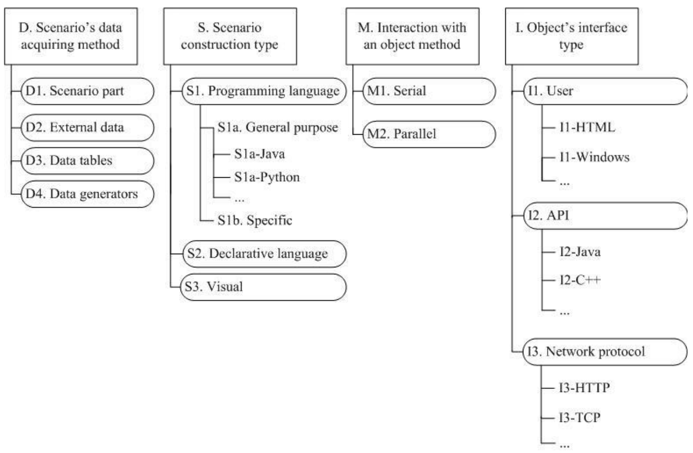


Figure 3. test automation tools classifier [3].

Scenario's data acquiring method can be:

- data as scenario part (D1) – scenario contains constant values and runs with the same data set each time;

- external data (D2) – data can be changed without modifying scenario;

- data tables (D3) – data obtained externally, there is a possibility of executing the scenario with a different data set;

- data generators (D4) – tool can automatically generate testing data using a template.

Scenario construction type can be:

- using a programming language (S1) – these can be either general purpose (S1a) or specific tool languages (S1b), S1a can be further classified by languages (for example Java, Python or Multiple in case of using different languages);

- using a declarative language (S2) – unlike the first class, declarative language simplifies writing primitive scenarios, but makes impossible creating complex scenarios;

- using visual tools (S3) – scenario is constructed with visual interface, no text description available.

Interaction with an object method can be:

- serial execution (M1) – tool executes scenario step by step in a single copy;

- parallel execution (M2) – tool can execute several copies of a scenario in parallel, imitating object's multiple clients.

This criterion can be used for segregating testing tools into two wide categories: functional testing tools and stress testing tools.

Object's interface type can be:

- user interface level (I1) – tool imitates real user behavior, interacting with visible objects (windows, buttons, fields);

- API level (I2) – tool imitates system's unit, which uses an object on functions call level, this is applicable to unit testing tools;

- network protocol level (I3) – in this case tool is imitating a client part of a system, interacting with an object via network protocols

In the appendix there is a table with classification results for test automation tools mentioned in the paper (Appendix I) according to this classifier.

III. UNIT TESTING

Unit testing is fundamental to the way that people develop software [5]. It refers to testing of separate system's units. In object-oriented systems, units typically are classes and methods. These may also be a collection of procedures or functions.

Unit testing tools are represented with a set of xUnit tools which are programming language dependent (JUnit for Java programming language, NUnit supposed for .NET, CppUnit and CUnit for C/C++ correspondingly, and others). These tools imitate one of the system's modules, which use an object under test on the level of functions calling [6]. It corresponds to an API level (I2), previously mentioned in the classifier description. Unit testing is usually performed by developers and can be easily automated, providing the base for further application regression testing – checking whether applying small changes and errors correction does not violate system stability. This is how unit testing during development phase is connected with a regression testing, which is performed at maintenance phase after applying changes with new version release.

Classification of approaches

In order to classify unit testing software, several types of tools have been reported in the literature. These are test drivers and test stubs, dynamic testing tools and automatic test cases generators [7]. They are categorized in Table 2.

Test driver is a piece of software that controls the unit under test. Drivers usually invoke or contain the tested unit. Therefore units under test subordinate to their respective drivers. A stub is a piece of software that imitates the characteristics and behavior of a necessary piece of software that subordinates to the unit and is required for unit to operate.

Table II. Unit testing tools classification

Unit testing approach	***Data acquiring method***	***Interface type***	***Description***	**Tools**
Manual program execution	Test case contains constant values and runs with the same data set each time	API level	The whole program is being run. Proper parameter values are derived by manual calculation in order to invoke the required unit. The main disadvantage of this approach is that it is very time consuming, considering that a unit is tested several times with different test data, requires writing client code.	Automated Testing Framework
Automated test driver	Test case contains constant values and runs with the same data set each time	API level	Sometimes also called test harness. An advantage of the driver is providing a way of saving test cases for regression testing. The unit is required to be taken out of its operational environment. As a result certain values and procedures that are called in the unit become undefined. A test driver automatically constructs the declaration for the undeclared variables. But this approach requires software stubs (or mock objects), which are procedures for replacing undefined procedures called in a unit during a test. Constructing stubs becomes main time-consuming activity during the testing.	CUnit, CppUnit, JUnit.
Direct test access	Tool can automatically generate testing data using a template	API level	The tools can provide the same functionality as automated test drivers but without the need of constructing stubs. It allows the direct control of the unit under test without taking the unit out of its operational environment.	API Sanity Autotest

Unit testing frameworks are now available for many languages. Some but not all of these are based on xUnit, free and open-source software, which was originally implemented for Smalltalk as SUnit.

TTCN-3

One of the new possibilities in unit testing was introduced with a Testing and Test Control Notation version 3. TTCN-3 new test domains have emerged – it can be applied at an earlier stages (during unit testing), but it requires a mapping of the language under test into TTCN-3 to exist [8].

Furthermore, mapping must provide the same operational semantics as mapped language. In [8], a sample C/C++ to TTCN-3 mapping is proposed (Figure 4).

Primarily TTCN was used for conformance testing in communicating systems sphere. With the new version TTCN-3, usage can be expanded to new testing types and new testing domains (Figure 5). Tools supporting TTCN-3 are provided from various software companies: OpenTTCN, Telelogic, Testing Technologies, IBM/Rational and others. The programming language is also used internally in such corporations as Nokia, Motorola and Ericsson.

Advantages of TTCN-3 usage are:

- TTCN-3 procedure-based communication allows direct interfacing to software modules.

- One testing language is used for testing systems under test (SUTs) in different programming languages. No need to write new test suites and test cases. Test artifacts re-usage allows reducing testing time and costs.

- TTCN-3 techniques can be combined with traditional approaches in unit testing.

- TTCN-3 can be edited and represented in multiple formats (core text format, tabular format, graphical format).

• Inheritance in C++	• Using TTCN-3 **import**

**class Base {};
class Derived :
public Base {};
class Derived2 :
public Base {};
class SubClass :
public Derived,
Derived2 {};**
	**module CppBase {
	type record CppBase_t
	{} }
	module CppDerived {
	import from CppBase all;
	type record CppDerived_t {
	CppPtr m_this optional,
	CppBase_t m_base
	}
	}
	module CppDerived2 { */* as above */* }
	module SubClass {
	import from CppDerived all;
	import from CppDerived2 all;
	type record CppDerived2_t {
	CppPtr m_this optional,
	CppDerived_t m_derived,
	CppDerived2_t m_derived2,
	}
	}**

Figure 4. Mapping C++ to TTCN-3 – Inheritance

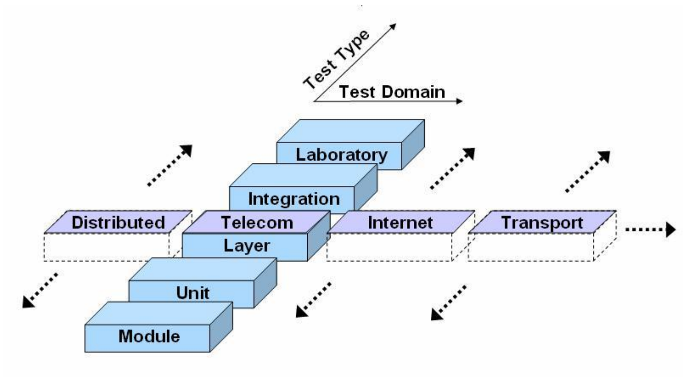


Figure 5. TTCN Usage (ETSI Official TTCN-3 Web Site)

IV. INTEGRATION TESTING

Integration testing is vital to ensure the correctness of integrated system. It is often the most expensive and time consuming part of testing. This testing activity can be divided into two categories:

- incremental: expanding the set of integrated modules progressively;

- non-incremental: software modules are randomly tested and combined.

Integration testing tools are designed for assisting in verification of components interaction. It is important to notice that only a result of the interaction matters, not the details or sequence of interaction. That is the reason why code refactoring process does not affect integration test cases. At the same time, with introducing new modules and functionality it is very easy to add interaction errors to a software product. That is the reason why regression testing is an essential part of integration testing [9].

There is a lack of studied and defined techniques or tools, which are specifically designed for integration testing. Test engineers are often forced to performing integration testing in ad-hoc (without planning and documenting) and ineffective ways that often leads to less reliable test results and errors left in interfacing between components [10].

Top-down integration

Top-down integration is an incremental approach to integration testing. Referring to [9] it is performed in five steps:

1. The main control module is selected as a test driver and all components, which are directly depending on the main module, are substituted with stubs.
2. Subordinate stubs are replaced one by one with actual components. The order of substitution is determined by the selected approach (in depth or in width).
3. Tests are executed after the each component is integrated. At this step testing tools, including automatic input data generation tools, test drivers and results recording tools are used. In [11] it is shown how Rational Rose is used for test generation.
4. After each set of tests is completed, the following stub is replaced by the real component.
5. Regression testing is conducted, in order to ensure that no new errors were produced by the integration.

The process is repeated from step 2 until the whole program structure is constructed.

In this approach the stubs tools are used (the same as in unit testing). This fact explains why software testing tools, initially designed for unit testing, are also used in integration testing. The examples of tools used in this approach are the above mentioned Rational Rose, xUnit frameworks and Cantata++. But in contrast to unit testing, the uncertainty of top-level modules behavior occurs, when most of lower levels are substituted with stubs. In order to resolve this uncertainty, the tester may adopt bottom-up integration approach.

Bottom-up Integration

Define abbreviations and acronyms the first time they are used in the text, even after they have been defined in the abstract. Abbreviations such as IEEE, SI, MKS, CGS, sc, dc, and rms do not have to be defined. Do not use abbreviations in the title or heads unless they are unavoidable.

Bottom-up integration starts from construction and testing components at the lowest level of program. In this approach no stub tools are used, because all the required processing information for a component is already available from the previous steps.

Bottom-up strategy exposes the following structure [9]:

1. Components combined into *clusters* (or *builds*), which are designated for a specific sub function.
2. A test driver is written to control test cases input and output.
3. The cluster is tested.
4. Then the driver is removed and cluster is integrated into the upper level.

From this perspective, tools that are used for integration testing again correspond to those for unit testing activity (test drivers). This could be almost considered an extension of unit testing. With bottom-up integration approach such tools as Cantata++ or VectorCAST/C++ can be used, which have been designed for both unit and integration testing.

Regression testing

Each time after new module is implemented and added into integration testing software behavior changes. With the changed structure of the software, new side effects might appear. In the context of integration testing, regression testing means execution of some tests subset that has already been conducted, after application’s code has been modified, in order to verify that it still functions correctly [9].

This activity can be carried out manually by executing some tests from all test cases or using automated capture/playback tools, which allow testers record test cases and repeat them for following results comparison. Regression testing often starts when there is anything to integrate and test at all. Test cases for regression should be conducted as often as possible. For example, after the new software build is produced, regression testing helps to identify and fix those code modifications that damage application functioning, stabilizing the build (so-called baseline).

Obviously, as it claimed in [1], the compromise should be made, considering the assurance by regression testing every time the change is submitted and the resources required to perform testing. As the application’s development process continues, the regression test suite grows in order to cover new or rewritten code. It may contain thousands of test cases, so that automation of regression testing becomes necessary. Regression test software structure is depicted at Figure 6.

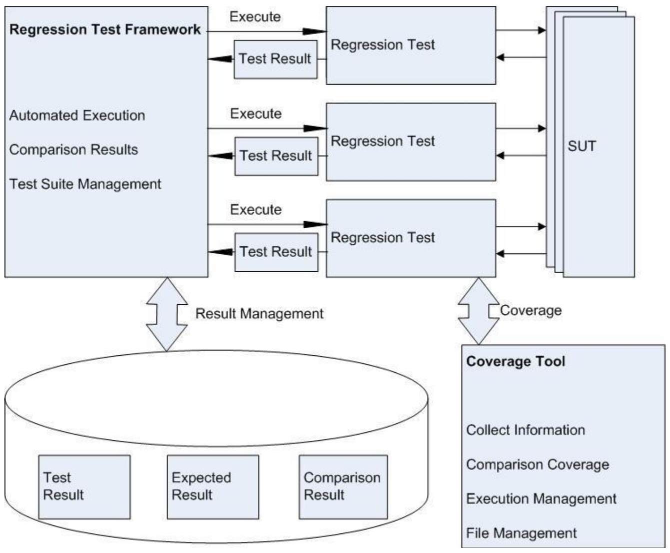


Figure 6. Architecture of regression testing software

Regression test tool consist of:

- Regression tests automation, which allows re-running tests as developers add new functionality. These can be composed of scripted or low-level functional tests or load tests that have been used earlier to verify desired application's behavior.
- Checkpoints management for comparison of the application characteristics and outputs against defined baselines. Checkpoints are used to stabilize application build.
- Regression test management for selecting test cases to run and execution order, because execution of all available test cases at every step is not effective.
- Regression test analyzing to detect which recent code modifications have broken functionality and fix them quickly.

Detected errors can be automatically reported to a bug tracking system after the test run.

The examples of regression testing tools are Selenium, SilkTest, Rational Functional Tester and QEngine.

V. FUNCTIONAL TESTING

Functional testing focuses on aspects surrounding the correct implementation of functional requirements. This is commonly referred to as black-box testing, meaning that it does not require knowledge of the underlying implementation.

Functional testing ensures that every function produces expected outcome, as it described in [12] for functionality quality characteristic.

Functional architecture

According to [13] a functional testing tool must provide resources, which are summarized in Table 3.

Table III. Functional testing tools resources

Resource type	***Description***
Tests definition	Constructed by recording an interaction with the SUT. The record produces a test script, which can be written in a common programming language or in a specific language. For handling data-driven test a tool must provide data access functionality, which selects data sources for the test. For managing test result analysis, control points are defined.
Tests execution	Test cases are automatically reproducing recorded user interaction. Data-driven tests are performed using data access that was set at tests definition phase.
Results reporting	On test completion, the results are compared with the reference state, which is based on the control points that were set at tests definition phase.

Facilitating previously mentioned capabilities, functional test tool relies on a repository, which stores the following elements:

- Function library. It is the list of all available application functions for defining test scripts.
- Object library. The list of recognized objects, which depends on the development environment and the platform where application is installed.
- Test scripts. These are records output, which can be further edited. Used for reproducing tests.
- Test results, which can be further analyzed with functional or other tools

The common structure of such application is depicted at Figure 7 [13].

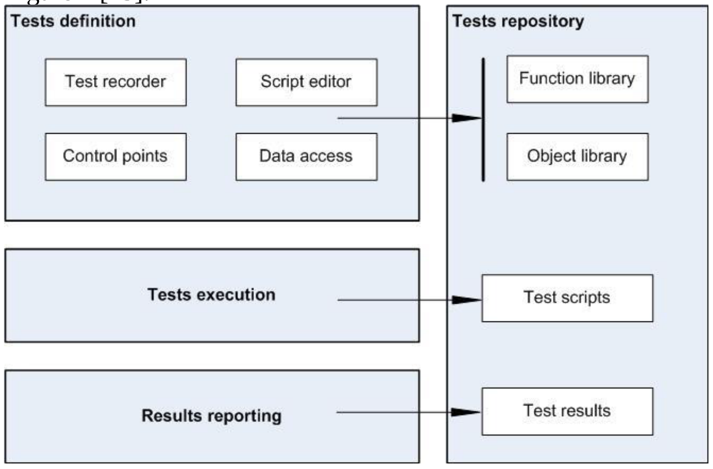


Figure 7. Functional test automation tool

Tools segmentation

Functional test tools should not be confused with test management tools, test evaluation tools and stress testing tools. In contrast to test management tools, functional tools provide the recording of tests. While test management tools are providing the capabilities for integration with other testing types tool (including functional tools), in order to manage test plans.

Functional test tools are focused on "black box" tests, while test evaluation tools are designed for "white box" technique. In contrast to test evaluation, functional test tools do not inspect the application source code. Finally, functional test tools can be distinguished from stress tools in perspective that they are not measuring the response time and the ability of the application to work under the various workloads.

One of the most used examples of functional test software includes Rational Robot (IBM Corporation) and SilkTest (Borland). Rational Robot is designed for ecommerce, ERP and client/server applications testing. It uses SQABasic for scripts recording. SilkTest uses Java and special purposed 4Test language for scripting. It is optimized both for traditional and Agile development environment, supporting faster iterative system delivery through a code-and-test cycles.

Most of analyzed in the study application testing tools are designed especially for functional testing. This can be explained with ISO 9126 Standard (2001), which considers functional quality characteristic as one of the most valuable.

VI. System testing

System testing tools performs end-to-end functional tests across software units, ensuring that all functions combine for the desired business result. The main problem in this testing is “finger-pointing”: when an error is uncovered, it is hard to localize the responsible system element [9]. System testing is a series of various tests with the main purpose of fully exercising the system.

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system in [14] this activity is split into recovery, security, stress and performance testing.

Security testing

Security testing relies on human expertise much more than an ordinary testing, so full automation of the security test process is less achievable than with other testing types [15]. Nevertheless, there is a significant number of black box test tools designed for testing application security issues. According to [15], these tools are aimed at testing:

- input checking and validation;
- session management;
- buffer overflow vulnerabilities;
- injection flaws.

Among the existing tools, there are subsets focused on specific security areas: database security, network security and web application security.

Database security test tools designed for identifying vulnerabilities, which can be results of an incorrect database configuration or poor implementation of the business logic accessing the database (SQL injection attacks). Database scanning tools are usually embedded into network security or web application security.

Network security tools generally allow network scanning and identifying vulnerabilities that give access to insecure services. These tools can also be referred to as penetration testing tools.

Web application security tools detect security issues for applications, which can be accessed via Internet. These tools are identifying abnormal behavior within applications available over specific ports, and can be used for Web Services based application technologies.

The technologies used in security testing tools can be divided, based on its functionality. The results are summarized in Table 4.

One of the most used examples of security testing tools include HP WebInspect, IBM Rational AppScan and Nikto, which were designed for automating Web application security testing.

Table IV. Security testing tools functionality

Functionality type	*Description*
Fuzzy injection	Injection of random data at various software interfaces.
Exploratory testing	Testing which is conducted without any specific expectation about the results.
Syntax testing	Generating a range of both legal and illegal inputs, usually considering some knowledge of underlying protocols and data formats used by the software.
Monitoring program behavior	Check how program responds to test inputs.

Performance and stress testing

Performance tests are often coupled with stress testing [9]. Stress testing is conducted to evaluate a system at the maximum design load or beyond the specified limits, while performance testing aimed at verifying that the software meets the specified performance requirements [1].

This testing activity is difficult, if possible at all, to perform manually due to a need of imitating a certain workload. The main principle of operation of performance and stress testing tools is simulation of real user with “virtual” users. The tool then gathers the statistics on virtual users’ experience. These types of software are often distributive in nature. In general performance testing tools can be divided into load generators, monitors and frameworks (such as LoadRunner, Jmeter, soapUI), and profilers (such as JProbe, Eclipse TPTP), which are used for finding performance bottlenecks, memory leaks and excessive memory consumption.

VII. Acceptance testing

Acceptance testing is aimed to explore how well users interact with the system, whether customer is satisfied with the results. It is final testing phase before deployment, but the tests themselves need to be designed as early as possible in the development life cycle. This makes sure that customer’s expectations are appropriately defined so that the system will be built in accordance with them. From this point of view acceptance test cases are derived from user requirements and the results of testing is acceptance or rejection of the product.

This testing activity differs from others in aspect that it may or may not involve the developers of the system, and can be performed by the customer [1]. If some errors are identified during acceptance testing, after developers correct them or after any change, the customer should go through acceptance tests again. In this manner, acceptance testing can be compared with regression testing [16]. It means that, as the project grows the number of acceptance tests increases (the same as with regression testing), because the customer gets better understanding of the final product, so the acceptance testing tools are required. Developers write unit tests in order to determine if the code is doing things right. Customers write acceptance tests in order to determine if the system is doing the right things.

Acceptance test driven development in Agile

One of the inventions in Agile methodology was the test-driven development (TDD), when the tests are written before writing the code. Then those tests are used for evaluating development process. In [17, 18] it is argued the benefits of the extension of TDD to the requirements/specification level, when the requirements are written in form of executable acceptance tests, so-called executable acceptance test driven development (EATDD).

It imposes that a feature is not specified until its acceptance test is written, and the feature is not done until all its acceptance tests pass [19]. EATDD also involves creating tests before actual code. Acceptance tests specify the behavior the software should have.

In [20], the inventor of Framework for Integrated Test (or “Fit”) Ward Cunningham advocates usage of spreadsheets for conducting acceptance tests. Spreadsheets provide the customer with the ability to write acceptance tests and enter data, which can be exported to text format. These data can be used by a development team for creating test scripts.

There are several tools for acceptance testing supporting EATDD. One of those is the above mentioned open source framework Fit, which was developed as an extension for xUnit environment, and supports most of modern programming languages (.Net, Java, Python, Ruby, C++, etc). FitNesse is Fit-based framework which was designed to support acceptance test automation.

The customer can write tests in a form of editing HTML tables, supporting it with an additional text. The developers can write supporting code (code fixtures) as the corresponding system feature has been implemented. Code fixtures can be regarded as a bridge between these tables and the SUT. Then the tool can parse the tables, execute tests and provide outputs as a modified HTML document. When requirements are captured in a format supported by a test framework, the acceptance tests then become a form of executable requirements [17].

The EATDD forces software stakeholders to come to an agreement about the exact behavior of the resulting product. It allows the development to be driven by the requirements, rather than letting requirements perspective out of sign as the development processes. Acceptance test driven development directly links requirements and QA [18].

VIII. DISCUSSION

The software testing activities study was conducted focusing on tools description and common features concerning each of the activity. The software testing tools were classified according to the model that was described in section 3.1.

48 testing tools have been collected and classified. Tools classification is summarized in the appendix (Appendix I). The resulting distribution of the tools over testing types is presented on Figure 8.

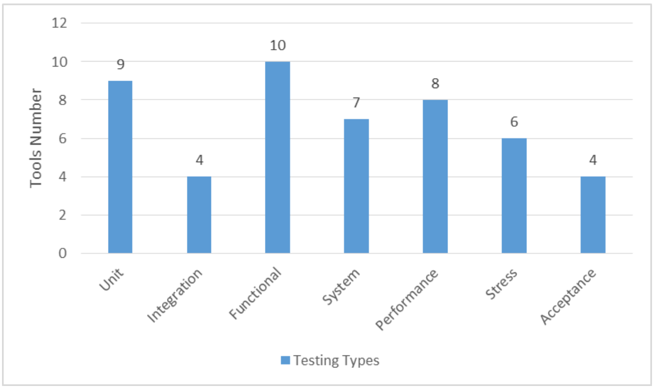


Figure 8. Testing tools for application testing by testing types

The above results showed that there is a large number of testing tools intended for functional, unit and performance testing. For functional testing there is a number of ways to ensure that a SUT meets functional requirements. Unit testing is a necessary for large systems and it can be considered as the basic phase in testing. While unit testing allows parallelism in testing process by presenting the opportunity to test multiple modules simultaneously and therefore can be easily automated. As for performance and stress testing, these activities are almost impossible to conduct manually and intended for an automatic execution by its nature, so there is a wide area for such type of tools usage.

At the same time, the smallest number of classified tools is intended for integration testing. This is due to a fact that unit testing frameworks often can be used for an integration testing, if it is regarded as an incremental unit testing.

It was rather difficult to identify system testing tools, because this activity is often split into many activities, and system testing is called the most difficult and misunderstood testing process [16]. This makes the right choice of system testing tools vital, because of the severity of errors, which can be detected at this phase.

Furthermore, it is worth noticing that an acceptance testing activity is not well yet automated. Obviously there is a lack of tools for this type of testing. So the tool usage for both system and acceptance testing is quite restricted. These comments can be taken into account when building a set of tools that overpass the borders in current software testing automation.

IX. CONCLUSION

The study illustrated that there is a lack of studies directed to overview and classify software testing tools. Even though there is an understanding between researchers that the correct selection of tools for software testing is one of the vital elements in assuring the quality of the whole project. Most of papers in the field of software testing are concentrated on testing methods description with no direct connection to tools, which are based on those methods.

The practitioner’s approach to software testing requires more information about currently available testing tools. With the growing software complexity and shorter development cycles, it is becoming evident that manual testing cannot provide quality level required for the market. As well as wrong testing tools choice for the project results in inadequate quality measurements or replacement of the tools during the project. Both wrong selection and change of testing tools during a development process affect software quality and as a result the project’s success.

The classifier used in this paper can be employed in appropriate choice of testing tool or set of tools for a software project. On the one hand it can be helpful for orientation in the wide subject field of software testing, reducing the amount of time required for specialists to find a proper solution. On the other hand it can be used as a quick introduction to a fast-developing area of testing and currently available testing tools for non-experts in this field.

As the conclusion more classification of tools may be needed. These classifications can be applied to testing a various set of projects depending on software type and development methodology.

X. REFERENCES

[1] SWEBOK (2004), IEEE Guide to Software Engineering Body of Knowledge.

[2] Taipale, O. (2007), Observations on software Testing Practice; Doctor of science thesis; Lappeenranta University of Technology.

[3] Suhorukov, A. (2010), Targeted training for the model and classifier for automate testing tools, Educational Technology and Society, January 2010, vol. 13, no. 1, pp. 370-377, in Russian.

[4] Fewster, M., Graham, D. (1999) Software Test Automation: Effective use of test execution tools, ACM Press, New York.

[5] Sen, A. (2010), Get to know CppTest, IBM Corporation.

[6] Beck, K. (2003), Test-Driven Development By Example. Addison-Wesley, Boston.

[7] DeMillo, R.A., McCracken, W.M., Martin, R.J. (1987), Software testing and evaluation, Banjamin/Kummings Publishing Company, Inc., California.

[8] Nyberg, A. & Kärki, M. (2005), Introduction to the C/C++ to TTCN-3 mapping, Nokia.

[9] Pressman, R. S. (2000), Software engineering: a practitioner's approach, McGraw-Hill, NY.

[10] Offutt, A., Abdurazik, A. and Alexander R. (2000), An Analysis Tool for Coupling-based Integration Testing, The Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS ’00), pp. 172–178

[11] Hartmann, J., Imoberdorf, C. and Meisinger, M. (2000), UML-Based Integration Testing, Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis, pp. 60-70.

[12] ISO/IEC (2001), ISO/IEC 9126-1, Software engineering - Product quality -Part 1: Quality model.

[13] Yphise (2002), Functional test automation tools. Software Assessment Report, Technology Transfer.

[14] Beizer, B. (1984), Software System Testing and Quality Assurance, Van Nostrand Reinhold.

[15] Michael, C., Radosevich, W. (2009), Black Box Security Testing Tools, Cigital Inc.

[16] Myers, G. J. (2004), The Art of Software Testing, Second Edition, John Wiley & Sons, NY.

[17] Hendrickson, E. (2008), Driving Development with Tests: ATDD and TDD, Quality Tree Software, Inc.

[18] Park, S., Maurer, F. (2008), The Benefits and Challenges of Executable Acceptance Testing, University of Calgary.

[19] Steindl, C. (2007), Test-Driven Development at the Acceptance Testing Level, Catalyst.

[20] Mugridge, R., Cunningham, W. (2005), Fit for Developing Software: Framework for Integrated Tests. Addison-Wesley.