LAPPEENRANTA UNIVERSITY OF TECHNOLOGY
Department of Information Technology
Master of Science Thesis

A SURVEY AND CLASSIFICATION OF SOFTWARE TESTING TOOLS

The topic of the master's thesis has been accepted in the Departmental Council of The Department of Information Technology, 20.01.2010.

Supervisors: Professor Kari Smolander
D.Sc. (Tech) Ossi Taipale

Lappeenranta, 05.05.2010

Sergey Uspenskiy
Kalliopellonkatu 10 B 10
53850 Lappeenranta
sergey.uspenskiy@lut.fi
+358 404 420 158

ABSTRACT

Author: Sergey Uspenskiy

Title: A survey and classification of software testing tools

Department: Information technology

Year: 2010

Master's thesis. Lappeenranta University of Technology.

61 pages, 14 figures, 6 tables and 2 appendices

Supervisors: Professor Kari Smolander

D.Sc. (Tech) Ossi Taipale

Keywords: software testing, validation, quality assurance, testing tools, automation.

Software testing is one of the essential parts in software engineering process. The objective of the study was to describe software testing tools and the corresponding use. The thesis contains examples of software testing tools usage. The study was conducted as a literature study, with focus on current software testing practices and quality assurance standards.

In the paper a tool classifier was employed, and testing tools presented in study were classified according to it. We found that it is difficult to distinguish current available tools by certain testing activities as many of them contain functionality that exceeds scopes of a single testing type.

PREFACE

The study was carried out as a part of the MASTO research project at Lappeenranta University of Technology during the period from February 2010 to May 2010 and was intended for educational purpose.

I would like to thank my supervisors Dr. Ossi Taipale and Prof. Kari Smolander for the opportunity to work at the thesis and for the advice, which I have received during writing this thesis.

It was very interesting, but rather difficult to choose the right pieces of information among the enormous amount of literature and studies available in the field of software testing. Nonetheless, despite the amount of researches, some topics are not overviewed in full or there are a lot of disagreements. That illustrates how the field of software testing is changing and expanding as the whole information technology industry does.

Lappeenranta, May 2010

Sergey Uspenskiy

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

ANSI	American National Standards Institute
API	Application Programming Interface
DTA	Direct Test Access
EATDD	Executable Acceptance Test Driven Development
ERP	Enterprise Resource Planning
ETSI	European Telecommunications Standards Institute
HTML	Hyper Text Markup Language
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
QA	Quality assurance
QC	Quality control
SUT	System under test
TDD	Test Driven Development
TTCN	Testing and Test Control Notation
UI	User Interface

1. INTRODUCTION

Information systems of different level of complexity are now integral parts of daily life. Any information system consists of hardware and software parts. At the earliest stage hardware part was much more expensive than software. Software cost was estimated about 5% of the whole system cost. However, software flexibility and extensibility allowed using it for multiple tasks on the same hardware. Gradually the complexity of software has grown, and today it covers from 30% to 90% of complete systems cost depending on type. Aggregate expenses on software development and maintenance are exceeding those for hardware already (Miller & Sanders, 1999).

According to Kit (1995) systems are becoming more complex and aimed at more critical tasks to perform. The sophisticated software systems require more elaborated processes to be implemented. At the same time, contemporary software engineering industry is characterized with a high level of competition. In order to be successful, software developers need to implement, introduce and maintain software in time and with satisfactory quality. The software quality can be explained as aggregation of features, properties and peculiarities, which define advantages and disadvantages of software (Sinicin et al., 2006).

In order to correspond to the growing complexity Quality Assurance (QA) engineers must use more and more sophisticated tools to perform software testing. There are numerous software systems arising to support testing process and even more have already been employed by organizations. However, it is difficult to distinguish which testing tools should be used for a concrete testing activity, because these software tools frequently have similar facilities. There is a gap of research in the field of selection of testing tools according to the organization's process or software project. The tools are often chosen according to the current needs or project and not considering long-term organization goals. Furthermore, the decision to change testing software can implicate additional

costs for the organization. For example, the process of switching from one requirements test software to another can take more than a year in a large organization and consume a lot of resources. Consequently the question of choosing the right testing tools should be taken into consideration at the pioneering stages.

The research question of testing tools classification covers a wide theoretical area. It requires an understanding of testing and quality assurance standards for the software development processes. The study was conducted as a literature study and explored aspects of testing activities and the corresponding testing tools.

The structure of the thesis is following: chapter 2 introduces a brief explanation on software testing methods and types, providing a theoretical ground for the following chapter, where the concrete testing tools are studied. The results of this Master of Science Thesis will be used in the development of the ISO/IEC 29119 Software Testing Standard (Part 4, Testing Techniques), where organization test strategy, including both test automation and tools, will be documented.

2. SOFTWARE TESTING

Software testing is an integral part of software development process. In order to agree upon a definition, first we express what is not software testing.
Software testing does not include:

- Development, even if test engineers can write code, including tests development (test automation can be compared to programming itself), develop some supporting tools for testing purposes. Nonetheless, testing — is not a development process.
- Analysis of requirements specification. Although, during a testing process sometimes requirements have to be specified more exactly, and sometimes requirements have to be analyzed. But this activity is not the body of testing, and it has to be done rather as a necessity.
- Management, despite that many organizations have test management positions. Certainly test engineers have to be controlled, but the testing itself is not management.
- Technical writing. Nevertheless test engineers have to document tests and activities.

To reflect the changes in software testing, we can show how the definition of testing has evolved in time:

- 1980

  The process of executing an application in order to find errors (Myers, 1980)
- 1990

  The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component (IEEE standard 610.12-1990, 1990).

- 1990

  Testing is not a process. It is intellectual discipline aimed at obtaining reliable software without unnecessary efforts on its control (Beizer, 1990).
- 1999

  Testing is defined as a technical investigation of the software in order to obtain information about its quality from stakeholders' point of view (Kaner, 1999).
- 2004

  The IEEE Guide to Software Engineering Body of Knowledge defines testing as checking compliance between the real program behavior and its expected behavior on the finite set of random tests (SWEBOK, 2004).

From this perspective, one can assume that from the year 1980 to 2004 the theory changed so deeply that the essence of the topic has changed completely. Definitions provided by Myers, Beizer or Kaner are describing software testing as an activity aimed at something. While in SWEBOK it is defined what is testing activity, but there is no information about the purposes of testing.

In this thesis we will use software testing definition derived from (Taipale, 2007), because it is not connected with any specific life cycle model or a development method:

> Testing is verification and validation.

Both verification and validation are activities aimed to software quality control and error detection. Having the same goals, they differ in sources of properties, rules, and restrictions that are being checked during these activities.
Verification checks conformance between the artifacts that are being created during software development and maintenance process and those artifacts that have been created earlier or used as a source input data, as well as artifacts and devel-

opment process compliance with rules and standards. A verification definition is provided by the IEEE standard 610.12-1990 (1990):
Verification is the process of evaluating a system or a component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Validation checks conformance of any artifacts, which are being created or used during development or maintenance, with user or customer needs and requirements. The validation's definition is given in IEEE standard 610.12-1990 (1990): Validation is the process of evaluating a system or a component during or at the end of the development process to determine whether it satisfies specified requirements.

Verification answers the question: are we building the product right? Validation answers the question: are we building the right product? (Boehm, 1979) The differences between verification and validation are illustrated on Figure 1.

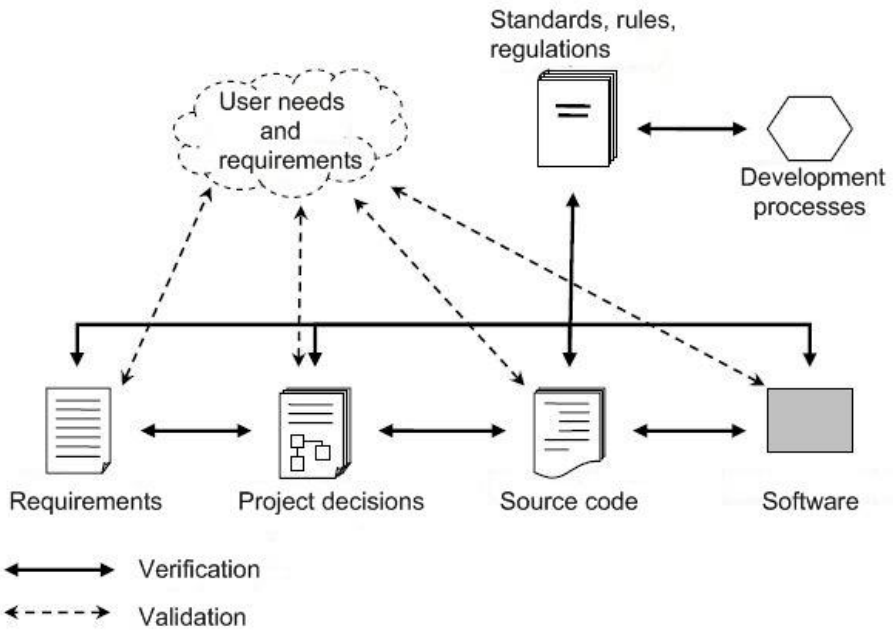


Figure 1. Correspondence between verification and validation (Kulyamin, 2008)

2.1. Objectives of software testing

The main purpose of testing can be described as a negative feedback to project participants about the quality of software product (Figure 2, Sinicin et al., 2006).

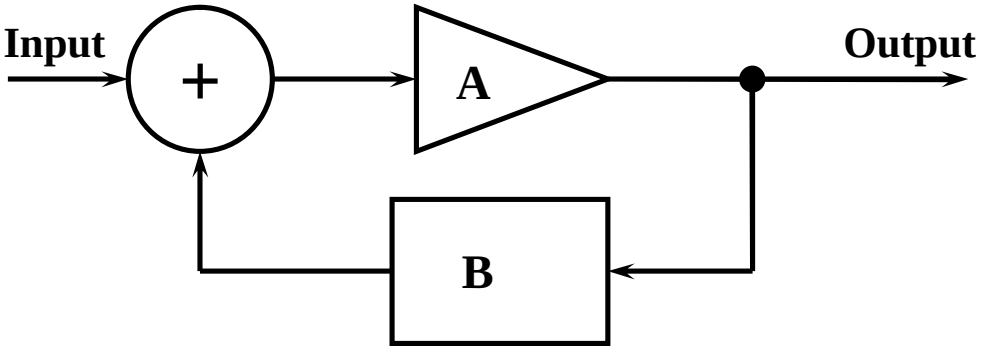


Figure 2. Testing provides a negative feedback

As defined in General System Theory, feedback describes a situation when some process output data (or information about the result of an event) comes to the input and will influence the same process or event in the future. This feedback can be both positive and negative. It is considered that a positive feedback is added to an input signal and reinforces it. A negative feedback decreases the input signal.

Both of them are equally important. Negative feedback stabilizes the system by reducing input signal. But the constant reducing of input may equal it to zero. Positive feedback strengths the input but may cause system unstable if the signal will be too large. For a system to be useful, it requires both types of feedback.

In software development positive feedback is certainly some information from end users or customers, for example, new functionality requests. Negative feedback can be also obtained from end users in a form of negative responses. Or it

comes from test engineers. The sooner the negative feedback is provided, the weaker signal this feedback has to modify. And therefore less energy and resources are required for modifying the signal. That is the reason why testing should be started as soon as possible, at the earliest steps of a project, providing a negative feedback at the design phase. Or even before, at the phase of requirements collection and analysis.

2.1.1. Testing, Quality Control, Quality Assurance

From this point of view the known abbreviation QA can not be a synonym for testing, despite the fact that testing covers most of QA activities. Quality assurance can not be treated as a negative feedback. In case of assuring, it is assumed that we provide some arrangement for the quality of software development to increase.

On the opposite, Quality Control (QC) can be considered testing in the wide sense of the word, because control is providing a negative feedback at various phases of a software project. The correspondence between testing, QC and QA is illustrated with Figure 3.

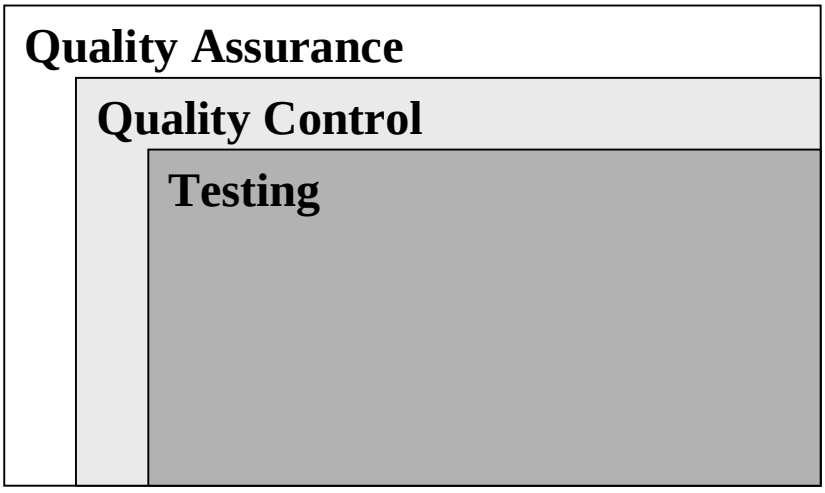


Figure 3. Testing — QC — QA

From SWEBOK (2004) definition, it can be derived that two basic tasks of a software tester include:

1. Controlling the execution of a program and providing test cases for checking program behavior.
2. Observing the execution process and comparing it with expected results.

Depending on testing types one can deviate from these actions. For example, in usability testing some additional resources can be used, for example, possible users from target group for evaluation.

In case of automated testing, a test engineer is not directly observing program execution; he delegates these tasks to a special tool or a program. This tool compares the results of the program execution with expected results and provides test engineer with the conclusion. There are more terms needed to be mentioned for defining two tasks mentioned above: stimulus, reaction and oracle. Stimulus is a data, which comes to program as input. Reaction – an output of the system. Oracle is a mechanism for comparing the output with the outputs that the system should provide and determining whether test has passed or failed (Kaner, 2004).

2.1.2. Testing program's interfaces

A program can be presented as a mechanism for processing information: gathering input information and providing output. It can have multiple inputs and outputs at the same time. Meaning that the program has multiple interfaces of various types:

- User interface (UI) for interaction with user.
- API for interaction between programs.
- Network protocol, can be also used for interaction between programs.
- File system for accessing disk.

- Environment' state which program can read or modify.
- Events.

The goals of software tester are creating test situations, using these interfaces, and evaluate program behavior. Corresponding processes and documents are depicted at Figure 4.

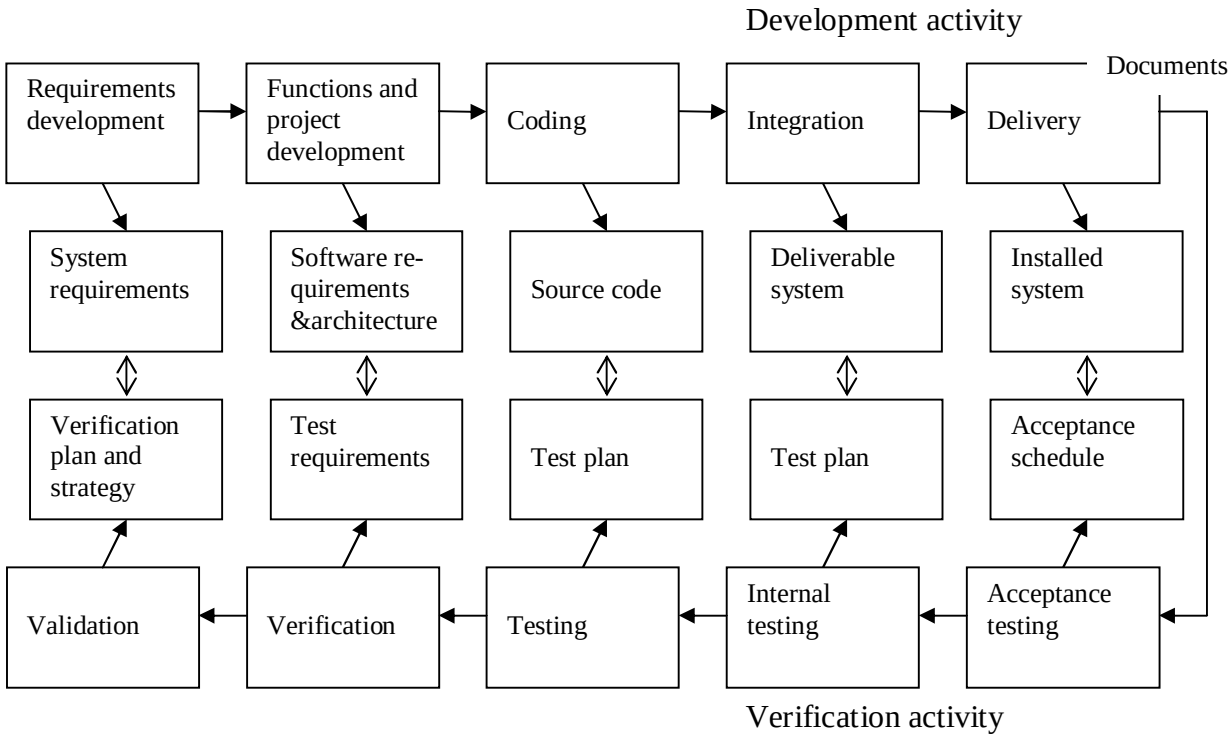


Figure 4. Processes and documents in software development (Sinicin et al., 2006)

2.2. Categories of software testing

2.2.1. ISO 9126 classification

Discussing software testing types, ISO 9126 (2001) provides six perspectives of quality, which are summarized in Table 1.

Table 1. Quality characteristics according to ISO 9126 (2001)

Characteristic	Description
Functionality	This is one of the most important quality aspects. The most significant subcharacteristic is suitability – appropriateness of a set of functions for specified tasks. Other subcharacteristics are accuracy – provision of right or agreed results or effects; and interoperability – ability to interact with specified systems, standards compliance.
Reliability	In this aspect the following subcharacteristics are concerned: maturity – the frequency of failure by faults in the software; fault tolerance – ability to maintain a specified level of performance in case of software faults or of infringement of its specified interface; and recoverability – capability to re-establish its level of performance and recover the data directly affected in case of a failure.
Usability	This aspect concerns understandability – the users' effort for recognizing the logical concept and its applicability; learnability – the users' effort for learning its application; operability – the users' effort for operation and operation control.
Efficiency	Concerns time behavior – response and processing times and throughput rates in performances its function; and resource behavior – the amount of resource used and the duration of such use in performing its function.
Maintainability	It is more internal quality aspect rather than external. This is more significant for the developers of software, rather than for the end users or customers. For this aspect more often analytical quality control methods are used. Static code analysis, code reviews etc. are not made with testing tools. Maintainability concerns code analyzability – the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified; changeability – the effort needed for modification, fault removal; stability – the risk of unexpected effect of modifications; testability – the effort needed for validating the modified software.
Portability	This includes adaptability – software should function in various environments; installability – the effort needed to install the software in a specified environment; coexistence – it should run simultaneously with other software.

Based on these quality aspects, six testing types can be defined:

- functionality testing;
- reliability testing;
- usability testing;
- efficiency testing;
- maintainability testing;
- portability testing.

However, this classification can not be generally accepted as single or complete. One can find classifications based on different aspects rather than quality perspectives presented in ISO 9126.

2.2.2. SWEBOK classification

There are more classifications of existing testing types. One of the most widely distributed is adopted from (SWEBOK, 2004) based on the target of the test. This classification is summarized in Table 2.

Table 2. Testing classification based on the target of test according to SWEBOK (2004)

Testing type	Description
Unit testing	It verifies if isolated software pieces, which can be tested separately, are functioning. Units can be individual subprograms or larger components of related units. This testing type is defined in IEEE standard 1008-1987 (1987), which also provides an approach to systematic and documented unit testing. Typically this testing implicates access to the source code being tested and support of debugging tools, and involves the programmers who developed the code.
Integration testing	This is the process of verifying the interaction between software components. In (SWEBOK, 2004) it is suggested that classical integration strategies (such as top-down or bottom-up) are used with traditional, hierarchically structured software; and modern integration strategies are rather architecture-driven. This is a continuous activity, at each stage of which software engineers need to concentrate on the level they are integrating.
System testing	In system testing the behavior of a whole system is taken into consideration, as the most of functional failures should already have been identified at unit and integration testing phases. This testing type is appropriate for validating non-functional requirements, such as security, speed, accuracy, and reliability. Also at this level external interfaces to other applications can be evaluated.

2.3. Overview of verification methods

In this section verification methods, intended first of all for evaluating technical artifacts of software life cycle are overviewed. These methods can be separated in the following groups – Figure 5 (Kulyamin, 2008).

Figure 5. Classification of verification methods

- *Review* of software life cycle artifacts can be divided into management review, technical review, walkthrough, inspection and audit. From the middle of 1990's scenario based software evaluation methods are being developed. Compared to other verification methods, reviews can be done

using only artifacts, not with artifacts' models (as in formal methods), or with artifacts' output (as in dynamic methods). Review can be applied to any quality of software or any artifact at any stage of project. It allows error revealing at the phase of artifact's development, minimizing defect's time existence and defect consequences for artifact's derivatives. At the same time, review can not be automated and requires active people participation. Empirical observation shows that reviews' effectiveness as located defects to consuming resources ratio is higher than with other verification methods. Reports show that from 50% to 90% of all stated errors during software life cycle can be located with reviews (Boehm, 2001; Deimel, 1995). Regarding to the early defect detection, expenditures can be from 5% to 80% of resources for detection at the test stage (Laitenberger, 2002). At the same time reviews' effectiveness is highly dependant on experience and motivation, as well as process organization and interaction between participants (Wong, 2006).

- *Static analysis* of software life cycle artifacts is used for verifying formal rules of correct artifact development and locating abundant defects using templates. This analysis can be highly automated and done completely relying on testing tools. However it is applicable only to a source code or some specific artifacts' presentation format and can show only limited set of defect types. Testing tools based on static analysis are widely used, because they do not require any additional training and are convenient in use. Many effective static analysis techniques can be a part of compilers or even transformed into semantic rules for programming languages.
- *Formal verification methods* use formal requirements models, software and environment behavior models for software analysis. It is performed with specific techniques such as theorem proving, model checking or abstract interpretation. Formal methods can be applied only to those qualities or artifacts, which can be represented with a formal mathematic model. Accordingly, implementing these methods is time consuming. It requires building formal models, which can not be automated. Neverthe-

less it is used in areas where the cost of an occurred error is significantly high, since it allows detecting complex error, which can not be located with reviews or other testing methods. During the last ten years tools based on formal methods have arisen, aimed at limited tasks of software verification, which can be effectively used at large industrial projects (Barnes, 2003). More frequently in practice formal methods are used with hardware (Kern, 1999).

- *Dynamic verification methods* provide analysis and evaluation of software system characteristics based on its real effort or efforts of some models or prototypes of the system. Examples of it include testing or simulation testing, monitoring, and profiling. In order to implement dynamic methods, one should have a working system or components, or at least its' prototypes, so it can not be used at initial development phases, but it is useful for controlling system's characteristics in its real environment, which can be difficult to do with other approaches. The errors detected with dynamic methods are often treated as most serious. These methods require some initial efforts to implement: tests development and test or monitoring system development, but test tools can be implemented once and then re-used for various types of software. It requires just tests re-implementing. At the same time, creating tests set which can provide an adequate quality measuring of a complex system is a resource consuming task, but it allows locating defects in requirements or project documents, while testers analyze them.
- *Synthetic methods*. During the last 10-15 years many studies have been conducted and tools developed, where elements of above mentioned methods are implemented. Thus dynamic methods with elements of formal methods were distinguished: model driven testing, runtime verification and passive testing (Broy et al., 2005). Some of the testing tools use both formalization of software aspects and static code analysis. The basic idea of these methods is combining advantages of various verification methods.

The classification given is mainly based on historical reasons and researchers of new methods usually try to conform to this classification, but if new synthetic methods continue arising, more thorough classification may be required.

2.4. Summary

To help one better understand the role of software testing in development process it can be depicted in the following figure (Figure 6):

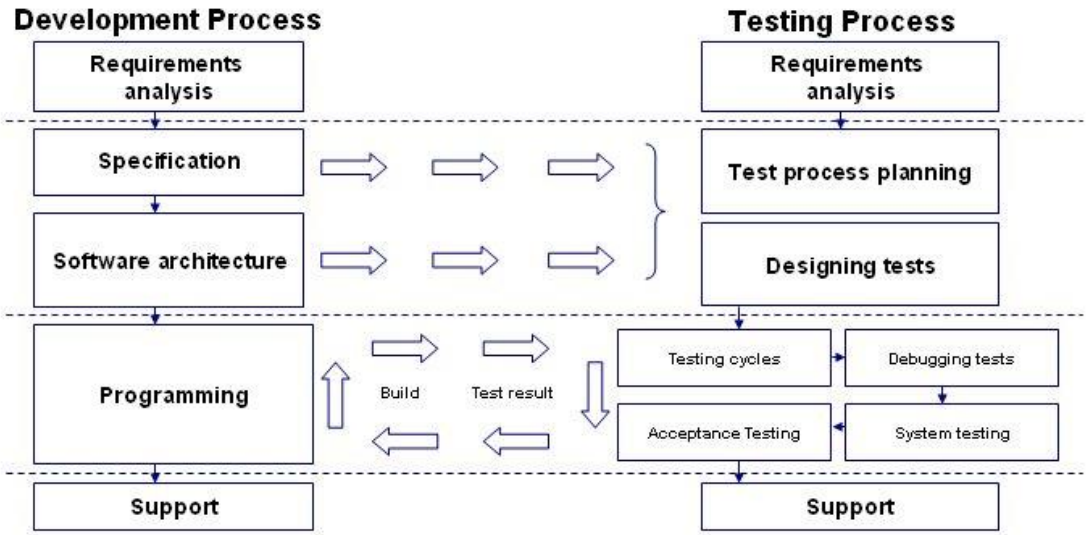


Figure 6. Testing in software development process

As software complexity grows, development processes become more sophisticated. Consequently software testing need to change, in order to correspond and support the development.

Literature and studies show that there is a lack of comprehension among software testers in using software testing tools. It is also true that sometimes testing

is mistakenly considered not a resource-intensive activity, which does not require any supportive tools (Myers, 2004).

This chapter gives a presentation of software testing as verification and validation, and describes verification methods. Validation activities and supportive tools are introduced in the next chapter, where tools are classified by a concrete activity and common characteristics for each tool group are discussed. In the near future, with new testing tools arising, more detailed tool reviews and classification may be needed.

3. VALIDATION TOOLS

In the previous chapter software testing has been introduced as verification and validation, and verification methods have been overviewed. The current chapter 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 7), 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 3.

Table 3. Test automation tools classification according to SWEBOK (2004)

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.

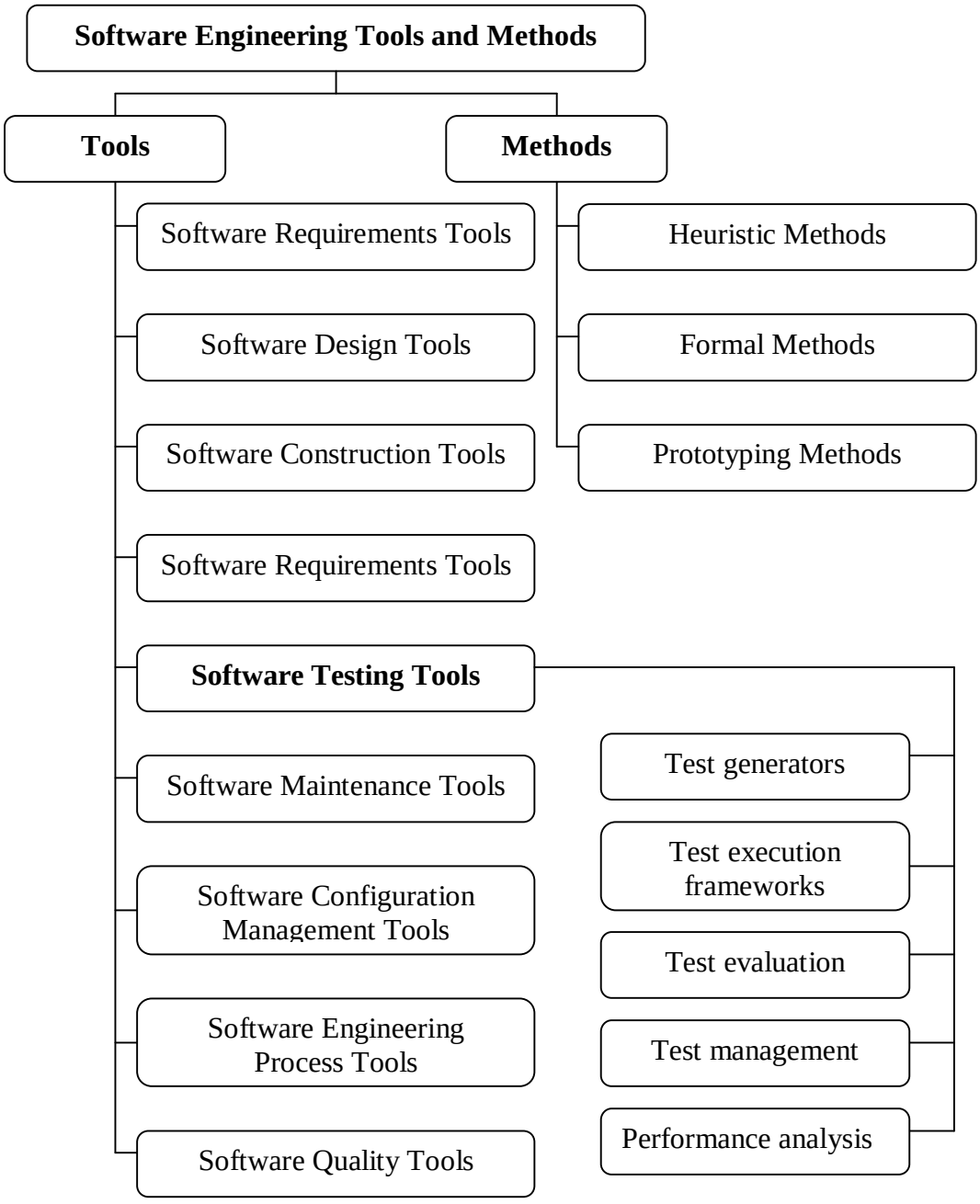


Figure 7. Software Engineering Tools and Methods (SWEBOK, 2004)

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 thesis, the classification is adopted from (Taipale, 2007), where the validation activity is divided into unit testing, integration testing, functional testing, system testing, and acceptance testing.

3.1. 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.

3.1.1. Automated test model

At present time some classifiers are available. A classifier derived from (Suhorukov, 2010) is based on a model of automated test (Figure 8). This classifier is supposed for test automation tools. Test automation can be defined as an activity when software tester just executes a test (or test sequence) and analyzes the results (Fewster, 1999).

The automated test model used in this classifier is general enough, so a test engineer can utilize it for modeling 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.

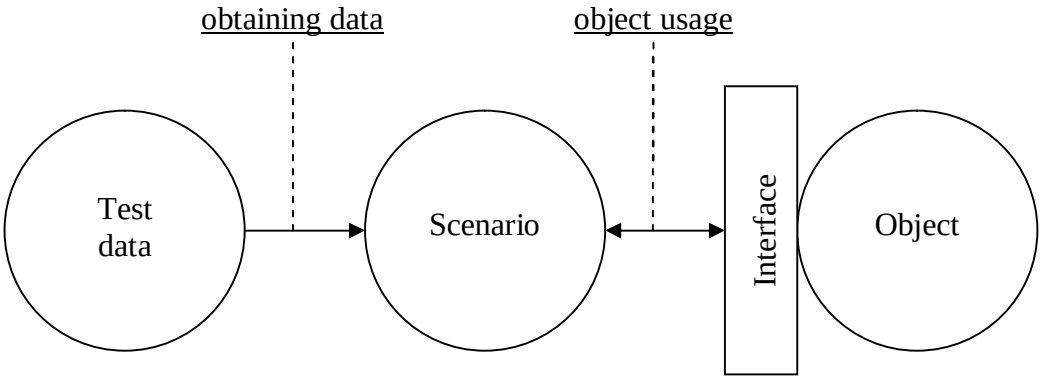


Figure 8. Automated test model schema (Suhorukov, 2010)

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 can not modify it, since the data is defined separately or is generated during a test case.

3.1.2. 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 9.

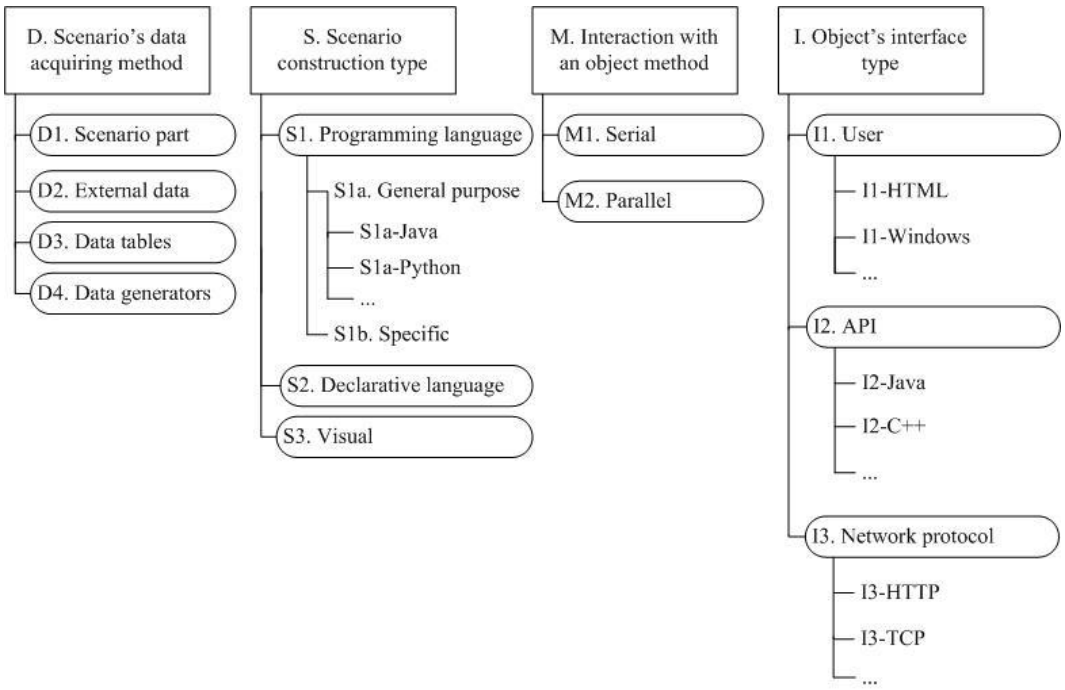


Figure 9. Test automation tools classifier (Suhorukov, 2010)

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.

3.2. Unit testing

Unit testing is fundamental to the way that people develop software (Sen, 2010). 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 dependant (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 (Beck, 2003). 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.

3.2.1. 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 (DeMillo et al., 1987). They are categorized in Table 4.

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 4. 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.

3.2.2. 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 (Nyberg & Kärki, 2005).
Furthermore, mapping must provide the same operational semantics as mapped language. In (Nyberg & Kärki, 2005), a sample C/C++ to TTCN-3 mapping is proposed (Figure 10).

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 11). 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++

```
class Base {};
class Derived :
   public Base {};
class Derived2 :
   public Base {};
class SubClass :
   public Derived,
        Derived2 {};
```

- Using TTCN-3 `import`

```
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 10. Mapping C++ to TTCN-3 – Inheritance

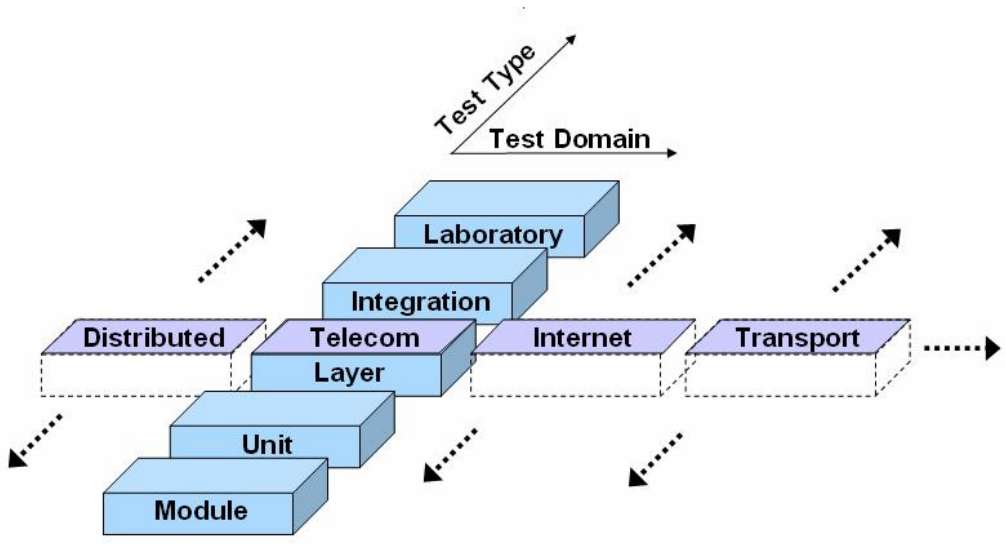


Figure 11. TTCN Usage (ETSI Official TTCN-3 Web Site)

In the appendix (Appendix II) a sample of the same TTCN-3 test case is presented in various formats, adopted from ETSI Official TTCN-3 Web Site.

3.3. 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 (Pressman, 2000).

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 (Offutt et al., 2000).

3.3.1. Top-down integration

Top-down integration is an incremental approach to integration testing. Referring to (Pressman, 2000) 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 (Hartmann et al., 2000) 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.

3.3.2. Bottom-up Integration

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 (Pressman, 2000):

1. Components combined into *clusters* (or *builds*), which are designated for a specific subfunction.
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.

3.3.3. 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 (Pressman, 2000).

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 (SWEBOK, 2004), 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 12.

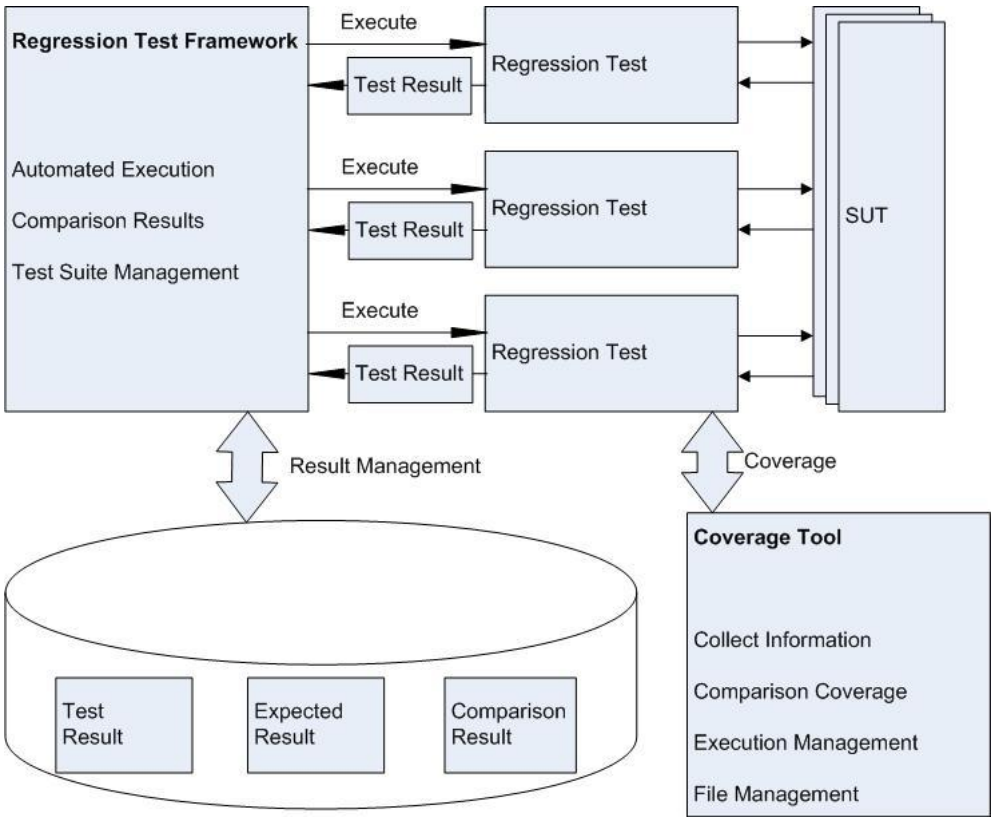


Figure 12. Architecture of regression testing software

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.

3.4. 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 (ISO 9126, 2001) for functionality quality characteristic.

3.4.1. Functional architecture

According to (Yphise, 2002) a functional testing tool must provide resources, which are summarized in Table 5.

Table 5. 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 13 (Yphise, 2002).

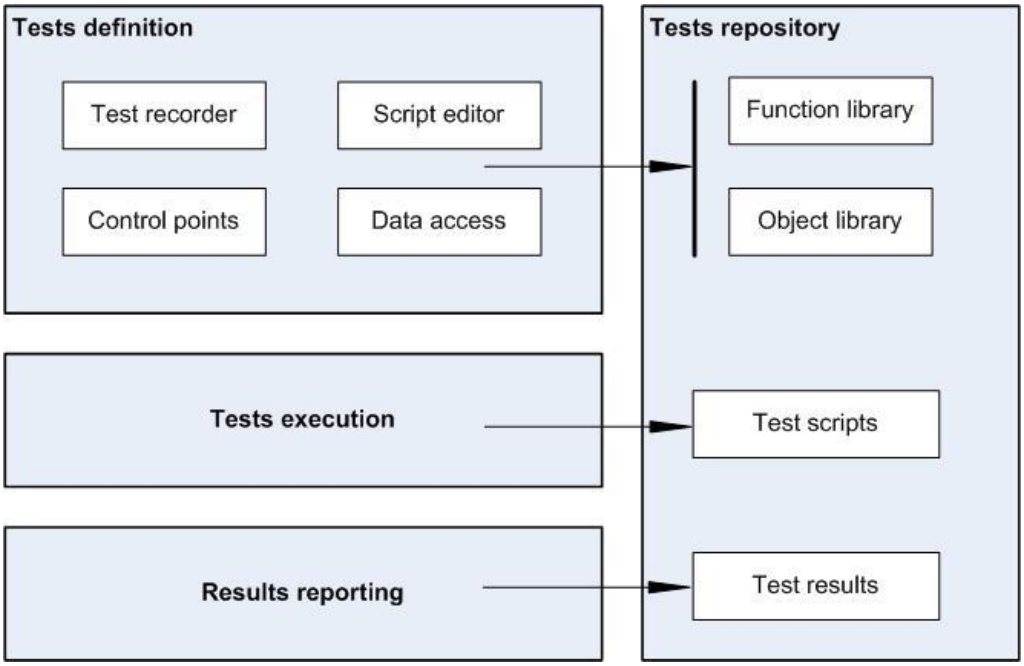


Figure 13. Functional test automation tool

3.4.2. 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 e-commerce, 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 specially for functional testing. This can be explained with ISO 9126 Standard (2001), which considers functional quality characteristic as one of the most valuable.

3.5. 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 (Pressman, 2000). 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 (Beizer, 1984) this activity is split into recovery, security, stress and performance testing.

3.5.1. 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 (Michael et al., 2009). Nevertheless, there is a significant number of black box test tools designed for testing application security issues. According to (Michael et al., 2009), 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 6.

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 6. 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.

3.5.2. Performance and stress testing

Performance tests are often coupled with stress testing (Pressman, 2000). 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 (SWEBOK, 2004).

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.

3.6. 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 (SWEBOK, 2004). 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 (Myers, 2004). 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.

3.6.1. 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 (Hendrickson, 2008; Park & Maurer, 2008) 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 (Steindl, 2007). EATDD also involves creating tests before actual code. Acceptance tests specify the behavior the software should have.

In (Mugridge & Cunningham, 2005), 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 (Hendrickson, 2008).

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 (Park & Maurer, 2008).

4. 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 chapter 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 14.

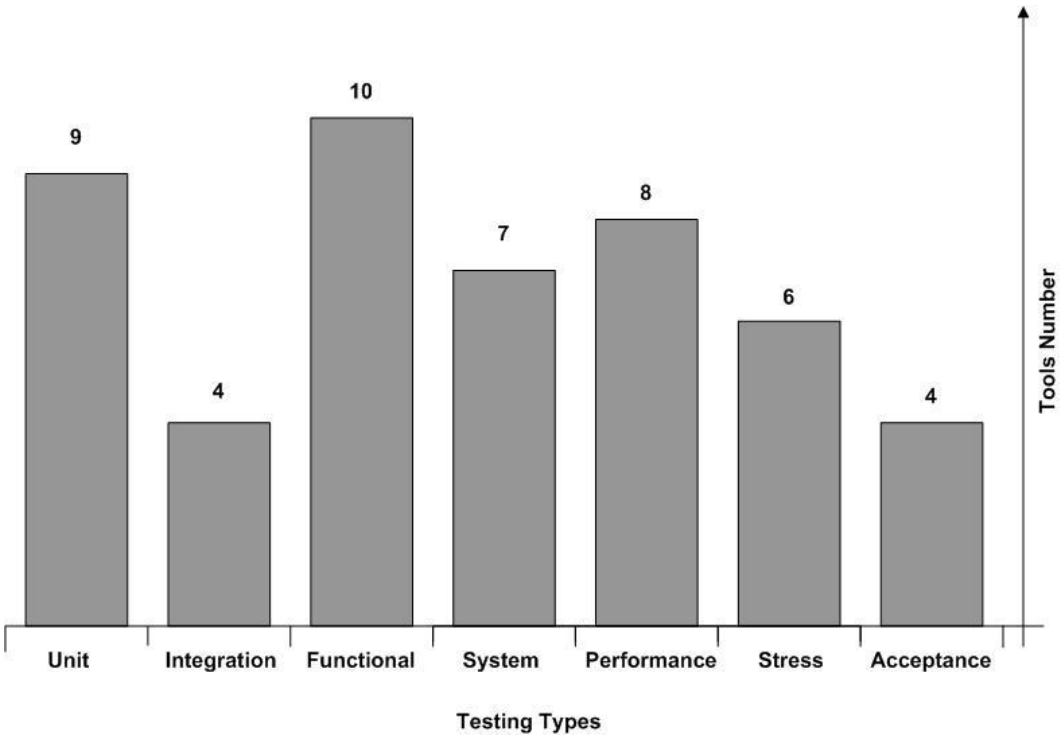


Figure 14. Testing tools for application testing by testing types

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 (Myers, 2004). 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.

The weakness of the thesis is that in given period of time it was not possible to examine all presented tools in depth. Many of them are sophisticated systems and require a lot of time to setup and employ. As for future solution a sample SUT can be presented specially for testing purposes. So that tools introduced in study can be applied to this system providing a practical demonstration of their usage. Another possible improvement is to conduct a comparison between testing tools, which belong to a similar group, in order to represent concrete tool's advantages and disadvantages.

5. CONCLUSIONS

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 can not 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 thesis 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.

REFERENCES

ApTest – Software QA Testing and Test Tool Resources.
Available: http://www.aptest.com
Accessed: 05.03.2010.

Barnes, J. (2003), *High Integrity Software. The SPARK Approach to Safety and Security*. Addison-Wesley.

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

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

Beizer, B. (1990), *Software Testing Techniques*. Van Nostrand Reinhold International Company Limited, New York, second edition.

Black, R. (2002), *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*, 2nd ed. New York, NY, John Wiley & Sons.

Boehm, B. W. (1979), *Software Engineering; R&D Trends and Defense Needs.* In R. Wegner, ed. Research. Directions in Software Technology. Cambridge, MA:MIT Press.

Boehm, B. W., Basili, V. (2001), *Software Defect Reduction Top 10 List.* IEEE Computer, 34(1):135-137, January 2001.

Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (2005), *Model Based Testing of Reactive Systems*. LNCS 3472, Springer.

Chillarege, R. (1999), *Software Testing Best Practices*, IBM Research.

Deimel, L., Rifkin, S. (1995), *Applying Program Comprehension Techniques to Improve Software Inspections.* The Software Practitioner 5(3):4-6, May-June 1995.

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

Dustin, E., Rashka, J. and Paul, J. (1999), *Automated software testing: introduction, management, and performance,* Addison-Wesley, Boston.

ETSI Official TTCN-3 Web Site.
Available: http://www.ttcn-3.org/
Accessed: 05.03.2010.

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

FitNesse. Acceptance Testing Framework.
Available: http://www.fitnesse.org/
Accessed: 05.03.2010.

Java-Source. Open Source Testing Tools in Java.
Available: http://java-source.net/open-source/testing-tools/
Accessed: 05.03.2010.

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.

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

IEEE/ANSI (1987), *IEEE Standard for Software Unit Testing,* 1008-1987.

IEEE/ANSI (1990), *IEEE Standard Glossary of Software Engineering Terminology*, 610.12-1990.

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

Kaner, C. (1999), *Testing Computer Software*, John Wiley and Sons, NY, pp. 37-40.

Kaner, C. (2004), *A Course in Black Box Software Testing*.
Available: http://www.testingeducation.org/BBST
Accessed: 05.03.2010.

Kern, C., Greenstreet, M. (1999), *Formal Verification in Hardware Design: A Survey*. ACM Transactions on Design Automation of Electronic Systems, 4:123-193, April 1999.

Kit, E. (1995), *Software Testing in the Real World: Improving the Process*, Reading, MA, Addison-Wesley.

Kulyamin, V. (2008), *Software Verification Methods*, Moscow, pp. 4-8, 29-35, in Russian.

Load Testing Tools resources.
Available: http://www.load-testing-tools.com/
Accessed: 05.03.2010.

Laitenberger, O. (2002), *A Survey of Software Inspection Technologies. In Handbook on Software Engineering and Knowledge Engineering*, v. 2, pp. 517-555, World Scientific Publishing.

Miller, H., Sanders, J. (1999), *Scoping the Global Market: Size Is Just Part of the Story*, IT Professional, 1(2), pp. 49-54.

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

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

Myers, G. J. (1980), *Software Reliability*, Mir, Moscow.

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

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

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

OpenTTCN DocZone.
Available: http://wiki.openttcn.com/
Accessed: 05.03.2010.

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

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

Sen, A. (2010), *Get to know CppTest*, IBM Corporation.

Sinicin, S., Nalutin, N. (2006), *Software Verification, Lections Course*, Moscow, in Russian.

Software-testing – Testing and Software Quality.
Available: http://www.software-testing.ru
Accessed: 05.03.2010.

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

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.

SWEBOK (2004), *IEEE Guide to Software Engineering Body of Knowledge*.

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

Voas J. (1999), *Software Quality's Eight Greatest Myths*, IEEE Software, September/October 1999, pp. 118-120.

Wong, Y. K. (2006), *Modern Software Review: Techniques and Technologies*, IRM Press.

Yphise (2002), *Functional test automation tools. Software Assessment Report*, Technology Transfer.

APPENDIX I: TEST AUTOMATION TOOLS CLASSIFICATION

Tool	Data	Scenario	Method	Interface	Tool's or developer's website
Abbot	D1	S1a-Java	M1	I1-Swing	http://abbot.sourceforge.net/
API Sanity Autotest	D1	S1a-C++	M1	I2-C++	http://ispras.linux-foundaton.org/index.php/API_Sanity_Autotest
Business Process Testing	D2	S3	M1	I1-Multiple	http://www.hp.com/
Canoo WebTest	D1	S2	M1	I1-HTML	http://webtest.canoo.com/
Cantata++	D1	S1a-C++	M1	I2-C++	http://www.ipl.com/products/tools/pt400.uk.php
Conformiq Qtronic	D3	S1a-TTCN	M1	I3-Mutiple	http://www.conformiq.com/qtronic.php
cPAMIE	D1	S1a-Python	M1	I1-HTML	http://pamie.sourceforge.net/
CppTest	D1	S1a-C++	M1	I2-C++	http://cpptest.sourceforge.net/
CppUnit	D1	S1a-C++	M1	I2-C++	http://cppunit.sourceforge.net/
CUnit	D1	S1a-C	M1	I2-C	http://cunit.sourceforge.net/
DTM Data Generator	D4	S3	M1	I2-SQL	http://www.sqledit.com/dg/
DTM DB Stress	D3	S3	M2	I3-Multiple	http://www.sqledit.com/stress/index.html
FitNesse	D1	S2	M1	I2-Multiple	http://www.fitnesse.org/
HttpUnit	D1	S1a-Java	M1	I3-HTTP	http://httpunit.sourceforge.net/
Jemmy	D1	S1a-Java	M1	I1-Swing	https://jemmy.dev.java.net/
JMeter	D3	S3	M2	I3-Multiple	http://jakarta.apache.org/jmeter/
JProbe	D2	S1a-Java	M2	I2-Java	http://www.quest.com/jprobe/
JUnit	D1	S1a-Java	M1	I2-Java	http://www.junit.org/
JVerify	D1	S1a-Java	M1	I2-Java	http://www.mmsindia.com/
LoadRunner	D3	S1a-C	M2	I3-Multiple	https://www.hp.com/
MessageMagic	D3	S1a-TTCN	M2	I3-Mutiple	http://www.elvior.com/messagemagic/
NeoLoad	D3	S3	M2	I3-Multiple	http://www.neotys.com/
Nikto	D1	S2	M1	I3-Multiple	http://cirt.net/nikto2

(to be continued)

APPENDIX I (continued)

Tool	Data	Scenario	Method	Interface	Tool's or developer's website
NUnit	D1	S1a-Multiple	M1	I2-.NET	http://www.nunit.org/
OpenSTA	D3	S1b	M2	I3-HTTP	http://www.opensta.org/
OpenTTCN Tester	D3	S1a-TTCN	M1	I3-Multiple	http://www.openttcn.com/
Oracle Functional Testing	D3	S3	M1	I1-HTML	http://www.oracle.com/
Oracle Load Testing	D3	S3	M2	I3-HTTP	http://www.oracle.com/
QALoad	D3	S1a-C++	M2	I3-Multiple	http://www.microfocus.com/
QEngine	D1	S3	M1	I1-HTML	http://www.manageengine.com/products/qengine/index.html
QF-Test	D1	S3	M1	I1-Swing	http://www.qfs.de/en/qftest/index.html
QuickTest Professional	D3	S1a-VBS	M1	I1-Multiple	http://www.hp.com/
Rational App Scan	D1	S3	M1	I3-Multiple	http://www.ibm.com/
Rational Functional Tester	D3	S1a-Multiple	M1	I1-Multiple	http://www.ibm.com/
Rational Performance Tester	D3	S3	M2	I3-Multiple	http://www.ibm.com/
Rational Robot	D1	S1b	M1	I1-Multiple	http://www.ibm.com/
Selenium	D1	S3	M1	I1-HTML	http://seleniumhq.org/
SilkPerformer	D3	S3	M2	I3-Multiple	http://www.borland.com/
SilkTest	D3	S3	M1	I1-Multiple	http://www.borland.com/
soapUI	D3	S3	M2	I3-Multiple	http://www.soapui.org/
TestComplete	D3	S1a-Multiple	M1	I1-Multiple	http://automatedqa.com/products/testcomplete/
TestNG	D1	S1a-Java	M1	I2-Java	http://testng.org/
TestPartner	D3	S1a-VBA	M1	I1-Multiple	http://www.microfocus.com/
The Grinder	D1	S1a-Jython	M2	I3-HTTP	http://grinder.sourceforge.net/
TOSCA	D2	S3	M1	I1-Multiple	http://www.tricentis.com/
VectorCAST/C++	D1	S1a-C++	M1	I2-C++	http://www.vectorcast.com/software-testing-products/c++-unit-testing.php

(to be continued)

APPENDIX I (continued)

Tool	Data	Scenario	Method	Interface	Tool's or developer's website
WAPT	D3	S3	M2	I3-HTTP	http://loadtestingtool.com/
WebInspect	D1	S3	M1	I3-Multiple	http://www.hp.com/

APPENDIX II: TTCN-3 TEST CASES PRESENTATION FORMATS

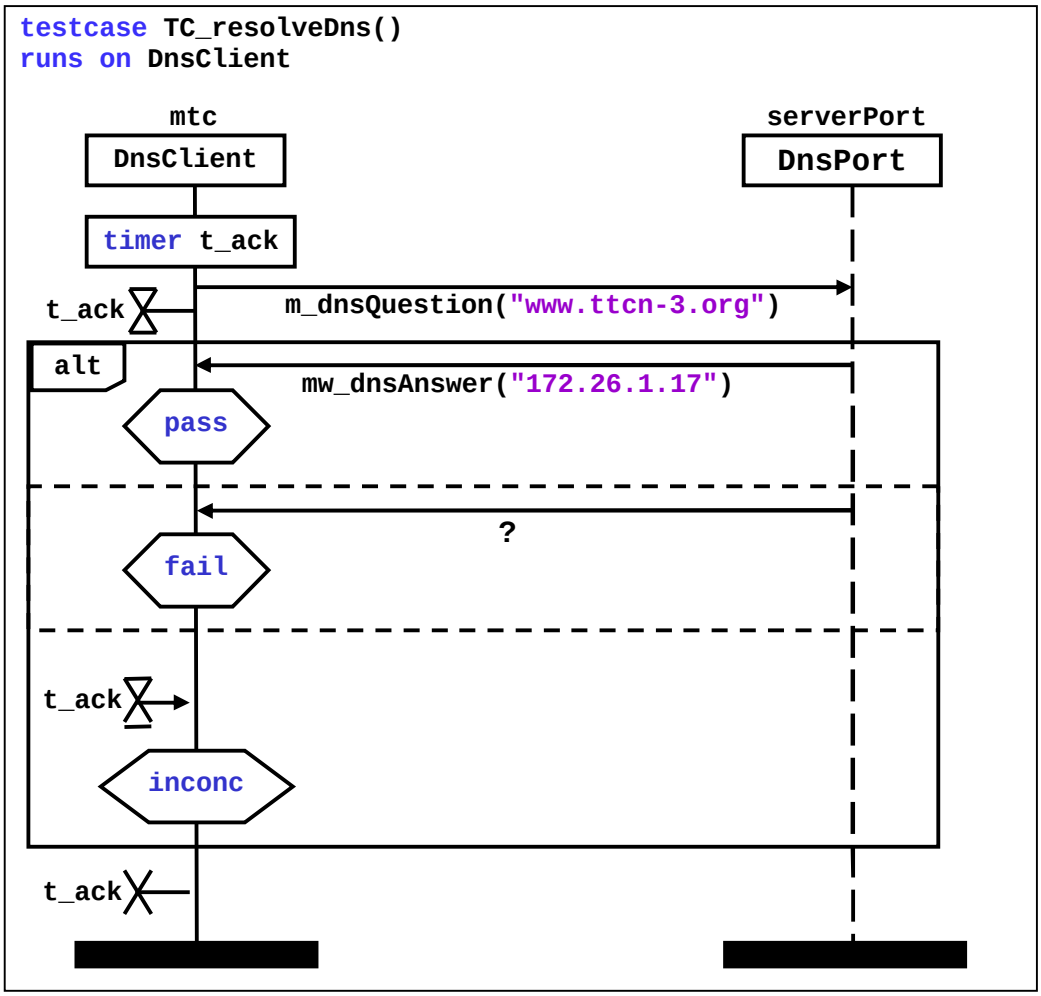
Example Core Format

```
testcase TC_resolveDns() runs on DnsClient
{
     timer t_ack;
     serverPort.send(m_dnsQuestion("www.ttcn-3.org"));
     t_ack.start(1.0);
     alt {
      [] serverPort.receive(mw_dnsAnswer("172.26.1.17")) {
            setverdict (pass);
         }
      [] serverPort.receive { // any other message
            setverdict(fail);
         }
      [] t_ack.timeout {
            setverdict(inconc);
         }
     }
   t_ack.stop;
}
```

(to be continued)

APPENDIX II (continued)

Example Graphical Format



(to be continued)

APPENDIX II (continued)

Example Tabular Format

Testcase			
Name	TC_resolveDns()		
Group			
Purpose			
System Interface			
MTC Type	DnsClient		
Comments			
Local Def Name	Type	Initial value	Comments
t_ack	timer		

Behavior

```
serverPort.send(m_dnsQuestion("www.ttcn-3.org"));
t_ack.start(1.0);
alt {
     [] serverPort.receive(mw_dnsAnswer("172.26.1.17"))    {
     setverdict (pass); }
     [] serverPort.receive { // any other message
     setverdict(fail); }
     [] t_ack.timeout    {
     setverdict(inconc); }
}

t_ack.stop;
```

Detailed Comments:	