Testing against natural language Requirements

Harry M. Sneed
Anecon GmbH, Vienna, Austria
Email: Harry.Sneed@t-online.at

Abstract: *Testing against natural language requirements is the standard approach for system and acceptance testing. This test is often performed by an independent test organization unfamiliar with the application area. The only things the testers have to go by are the written requirements. So it is essential to be able to analyze those requirements and to extract test cases from them. In this paper an automated approach to requirements based testing is presented and illustrated on an industrial application.*

Keywords: *Acceptance Testing, System Testing, Requirements Analysis, Test Case Generation, Natural Language Processing*

A test is always a test against something. According to the current test literature this something can be the code itself, the design documentation, the data interfaces, the requirements or the unwritten expectations of the users [1]. In the first case, one is speaking of code–based testing where the test cases are actually extracted from an analysis of the code. In the second case, one is speaking of design–based testing where test cases are taken from the design documents, e.g. the UML diagrams. In the third case, we speak of data–based testing, where test cases are generated from the data structures, e.g. the SQL schema or the XML schema. In the fourth case, we speak of requirements–based testing, where we extract the test cases from the requirements documents. This is also known as functional testing. In the fifth and final case, we speak of user–based testing, in which a representative user invents test cases as he goes along. This is also referred to as creative testing [2].

Another form of testing is regression testing in which a new version of a previous system is tested against the older version. Here the test cases are taken from the old data or from the behavior of the old system. In both cases one is comparing the new with the old, either entirely or selectively [3].

1. Functional testing

In this paper the method of requirements–based testing is being described, i.e. testing against the functional and non–functional requirements as defined in an official document. This type of testing is used primarily as a final system test or as an acceptance test. Bill Howden referred to this as functional testing [4]. It assumes that other kinds of testing, such as code–based unit testing and/or design–based integration testing have already taken place so that the software is executable and fairly reliable. It is then the task of the requirements–based test to demonstrate that the system does what it should do according to the written agreement between the user organization and the developing organization. Very often this test is performed by an independent test organization so as to eliminate any bias. The test organization is called upon not only to test, but also to interpret the meaning of the requirements. In this respect, the requirements are similar to laws and the testers are performing the roles of a judge, whose job it is to interpret the laws to apply to a particular case [5].

What laws and requirements have in common is that they are both written in natural language and they are both fuzzy. Thus, they are subject to multiple interpretations. Judges are trained to interpret laws. Testers are not always prepared to interpret requirements. However, in practice this is the essence of their job. Having an automated tool to dissect the requirement texts and to distinguish between different types of requirement statements is a first step in the direction of automated requirements testing. The Text Analyzer is intended to be such a tool.

2. Nature of natural language requirements

Before examining the functions of a requirement analysis tool, it is first necessary to investigate the nature of requirement documents. There may be certain application areas where requirements are written in a formal notation. There are languages for this, such as VDM, SET and Z, and more recently OCL – the object Constraint Language propagated by the OMG [6]. However, in the field of information technology such formal methods have

never really been accepted. There, the bulk of the requirements are still written in prose text.

Christof Ebert distinguishes between unstructured text, structured text and semi formal text [7]. In a structured text the requirements are broken down into prescribed chapters and sections with specific meanings. A good example of this is the ANSI/IEEE-830: Guide to Requirements Specification. It prescribes a nested hierarchy of topics including the distinction between functional and non–functional requirements [8]. Functional Requirements are defined in terms of their purpose, their sequence, their preconditions and post conditions as well as their inputs and outputs. Inputs are lists of individual arguments and outputs lists of results. Arguments and results may be even defined in respect to their value ranges. This brings the functional specification very close to a functional test case.

Non–functional requirements are to be defined in terms of their pass and fail criteria. Rather than depicting what flows in and out of a function, a measurable goal is set such as a response time of less than 3 seconds. Each non–functional requirement may have one or more criteria which have to be fulfilled in order for the requirement to be fulfilled.

In addition to the functional and non–functional requirements of the product, the ANSI/IEEE standard also stipulates that constraints, risks and other properties of the projects be defined. The end result is a highly structured document with 7 sections. Provided that standard titles or standard numbering is used, a text analysis tool should easily recognize what is being described even if the description itself is not interpretable. By having such a structured document a tester has an easier job of extracting test cases. The job becomes even easier if the structured requirements are supplemented by acceptance criteria as proposed by Christiana Rupp and others [9]. After every functional and non–functional requirement a rule is defined for determining whether the requirement is fulfilled or not. Such a rule could be that in case of a withdrawal from an account, the balance has to be less than the previous balance by the withdrawal amount.

Account = Account@pre – Withdrawal;

An acceptance criterion is equivalent to a post condition assertion so that it can be readily copied into a test case definition.

Semi formal requirements go one step further. They have their text content placed in a specific format, such as the use case format. Use cases are typical semi formal descriptions. They have standardized attributes which the requirement writer must fill out, attributes like trigger, rule, precondition, post condition, paths, steps and relations to other use cases. In the text these attributes will always have the same name so that a tool can readily recognize them. Most of the use cases are defined in standard frameworks or boxes which make it even easier to process them [10].

A good semi formal requirements document will also have links between the use cases and the functional requirements. Each requirement will consist of a few sentences and will have some kind of number or mnemonic identifier to identify it. This identifier will then be referred to by the use case. One use case can fulfill one or more functional requirements. One attribute of the use case will be a list of such pointers to the requirements it fulfills [11].

At the upper end of a semiformal requirement specification arithmetic expressions or logical conditions may be formulated. Within an informal document there can be scattered formal elements. These should be recognizable to an analysis tool.

In the current world of information technology, the requirement documents range from structured to semi formal. Even the most backward users will have some form of structured requirements document in which it is possible to distinguish between individual functional requirements as well as between constraints and non functional requirements. More advanced users will have structured, semi formal documents in which individual requirements are numbered, use cases are specified with standardized attributes, and processing rules are defined in tables. Really sophisticated requirement documents such as can be found in requirements engineering tools like Doors and Rational Requisite Pro will also have links between requirements, rules, objects and processes, i.e. use cases [12].

3. The Testing Strategy

A software system tester in industry is responsible for demonstrating that a system does what it is supposed to do. To accomplish this, he must have an oracle to refer to. The concept of an automated oracle for functional testing was introduced by Howden in 1980 [13]. As foreseen by Howden then,

the test oracle was to be a formal specification in terms of pre and post conditions. However the oracle could also be a natural language text provided the text is structured and has some degree of formality.

In regression testing the oracle is the input and output data of the previous version. In unit testing it is the pre and post conditions of the methods and the invariant states of the objects. In integration testing it is the specification of the interfaces and in system testing it is the requirements document [14]. Thus, it is the task of the system tester to extract test cases from the functional and non–functional requirements. Using this as a starting point, he then proceeds to carry out seven steps on the way to achieving confidence in the functionality of a software system. These seven steps are:

- identifying the test cases
- creating a test design
- specifying the test cases
- generating the test cases
- setting up the test environment
- executing the test
- evaluating the test.

3.1 Identifying the test cases

Having established what it is to be tested against, i.e. the test oracle, it is first up to the tester to analyze that object and to identify the potential test cases. This is done by scanning through the document and selecting all statements about the behavior of the target system which need to be confirmed. These statements can imply actions or states, or they define conditions which have to be fulfilled if an action is to take place or a state is to hold [15].

Producing a customer reminder is an action of the system. The fact that the customer account is overdrawn is a state. The rule that when a customer account is overdrawn the system should produce a customer reminder is a condition. All three are candidates for a test case. Testing whether the system produces a customer reminder is one test case. Testing if the customer account can be overdrawn is another test case, and testing whether the system produces a customer reminder when the customer account is overdrawn is a test case which combines the other two.

In scanning the requirements document the tester must make sure to recognize each action to be performed, each state which may occur and each condition under which an action is performed or a state occurs. From these statements the functional test cases are extracted. But not only the functional test cases. Statements like the response time must be under 3 seconds and the system must recognize any erroneous input data are non functional requirements which must be tested. Every statement about the system, whether functional or non–functional is a potential test case. The tester must recognize and record them [16].

3.2. Creating a test design

Of course, this is only the beginning of a system test. Once the test cases have been defined they must be ordered by time and place and grouped by test session. A test session encompasses a series of test cases performed within one dialog session or one batch process. In one session several requirements and several related use cases are executed. The test cases can be run sequentially or in parallel. The result of this test case ordering by execution sequence is part of the test design.

3.3 Specifying the test cases

Following the test design is the test case specification. This is where the attributes of the test cases are filled out in detail down to the level of the input and output data ranges. Each test case will already have an identifier, a purpose, a link to the requirements, objects and use cases it tests, as well as a source, a type and a status. It may even have a pre and post condition depending on how exact the requirements are. Now it is up to the tester to design the predecessor test cases, the physical interface or database being tested and to assign data values.

Normally the general test case description will be in a master table whereas the input and output values will be in sub tables one for the test inputs and one for the expected outputs. In assigning the data, the tester will employ such techniques as equivalence classes, representative values, boundary values and progression or degression intervals. Which technique is used, depends on the type of data. In the end there will be for each test case a set of arguments and results [17].

3.4 Generating the test data

Provided the test data definitions are made with a formal syntax, the test data itself can then be automatically generated. The tester may only have to oversee and guide the test data generation process. The basis for the test data generation will be the interface descriptions such as HTML forms, XML schemas, WSDL specifications and SQL database schemas. The values extracted from the test case, specifications are united with the structural

information provided by the data definition formats to create test objects, i.e. GUI instances, maps, records, database tables and other forms of test data [18].

3.5 Setting up the test environment

In the 5th step the test environment is prepared. Test databases must be allocated and filled with the generated data. Test work stations are loaded with the client software and the input test objects. The network is activated. The server software is initialized. The source code may be instrumented for tracing execution and test coverage.

3.6 Execution the test

Now the actual test can be started, one session at a time or several sessions in parallel depending on the type of system under test. The system tester will be either submitting the input data manually or operating a tool for submitting the data automatically. The latter approach is preferable since it is not only much faster, but also more reliable and above all repeatable. While the test is running the execution paths are being monitored and the test coverage of the code is being measured.

3.7 Evaluating the test

After each test session or test run the tester should perform an analysis of the test results. This entails several sub tasks. One sub task will be to report any incidents which may have occurred during the test session. Another task will be to record and document the functional test coverage. A third and vital task is to confirm the correctness of the data results, i.e. the post conditions. This can and should be done automatically by comparing the actual results with the expected results as specified in the test cases. Any deviations between the actual and the specified data results should be reported. Finally the tester will want to record various test metrics such as the number of test cases executed, the number of requirements tested, the number of data validated, the number of errors recorded and the degree of test coverage achieved [19].

4. Automating the requirement analysis

As can be gathered from this summary of the system tester's tasks, there are many tasks which lend themselves to automation. Both test data generation and test data validation can be automated. Automated test execution has been going on for years and there are several tools for performing this. What are weakly automated are the test case specification and the test design. Not automated at all are the activities setting up the test environment and identifying the test cases [20].

The focus of this paper is on the latter activity, i.e., identifying and extracting test cases. It is the first and most important task in functional system testing. Since the test we are discussing here is a requirements based test, the test cases must be identified in and extracted from the requirements document.

The tool for doing that is the text analyzer developed by the author. The same tool goes on to create a test design, thus covering the first two steps of the system testing process. The „Text Analyzer" was conceived to do what a tester should do when he begins a requirements–based system test. It scans through the requirements text to pick out potential test cases.

4.1 Recognizing and selecting essential objects

The key to requirements analysis is to have a natural language processor which extracts information from the text based on key words and sentence structure. This is referred to as text mining, a technique used by search engines on the internet. [21] The original purpose of text mining was to automatically index documents for classification and retrieval. The purpose here is to extract test cases from natural language text.

Test cases relate to the objects of a system. Objects in a requirement document are either acted upon or their state is checked. Therefore, the first step of the text analysis is to identify the pertinent objects. For this all of the nouns must be identified. This is not an easy task, especially in the English language, since nouns can often be verbs or compound words such as "master record". In this respect other languages such as German and Hungarian are more precise. In German nouns begin with a capital letter which makes the object recognition even easier.

A pre scanner can examine the text to identify and record all nouns. However, only the human analyst can determine which nouns are potential objects based on the context in which they are used. To this end all of the nouns are displayed in a check box and the user can uncheck all nouns which he perceives to be irrelevant. The result is a list of pertinent nouns which can be recorded as the essential objects. Depending on the scope of the requirements

document their number can be anywhere from 100 to 1000.

Besides that, object selection is apt to trigger a lengthy and tedious discussion among the potential users about which objects are relevant and which are not. In presenting the list of potential objects it becomes obvious, how arbitrary software systems are. In order to come up with an oracle to test against, the users must first come to a consensus on what the behavior of the system should be. Confronting them with the contradictions in their views helps to establish that consensus. [22]

4.2 Defining key words in context

As a prerequisite for the further text analysis, the user must identify the key words used in the requirement text. These key words can be any string of characters, but they must be assigned a predefined meaning. This is done through a key word table. There are currently some 20 predefined notions which can be assigned to a key word in the text. These are:

SKIP = ignore lines beginning with this word
REQU = this word indicates a requirement
MASK = this word indicates a user interface
INFA = this word indicates a system interface
REPO = this word indicates a report
INPT = this word indicates a system input
OUTP = this word indicates a system output
SERV = this word indicates a web service
DATA = this word indicates a data store
ACT = this word indicates a system actor
TRIG = this word indicates a trigger
PRE = this word indicates a precondition
POST = this word indicates a post condition
PATH = this word indicates a logical path or sequence of steps
EXCP = this word indicates an exception condition
ATTR = this word indicates any user assigned text attribute
RULE = this word indicates a business rule
PROC = this word indicates a business process
GOAL = this word indicates a business goal
OBJT = this word is the name of an object.

By means of the key words, the analyzer is able to recognize certain requirement elements embedded in the prose text.

4.3 Recognizing and extracting potential test cases

The next step is for the tool to make a second scan of the document. This time only sentences in which an essential object occurs are processed, the others are skipped over. Each sentence selected is examined whether it is an action, a state query, or a condition. The sentence is an action when the object is the target of a verb. The sentences „The customer account is updated daily" and „The system updates the customer account" are both actions. The "account" is the object and "updates" is the action. The test case will be to test whether the system really updates the account.

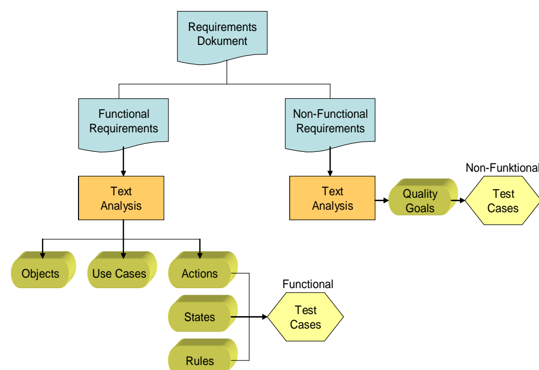

Figure 1: Actions, States & Rules

The sentence „The account is overdrawn when the balance exceeds the credit limit" is a state which needs to be tested and the sentence „If an account is overdrawn, it should be frozen until a payment comes in" is a condition combining an object state with an object action. The object is the "account". The state is "overdrawn". The action is "should be frozen". There are actually two tests here. One is to confirm that an account can become overdrawn. The other is to confirm that an account is frozen when it is overdrawn.

To qualify as a statement to be tested, a sentence must contain at least one relevant object. In the sentence "If his credit rating is adequate, the customer can order a book." there are three relevant objects - credit, customer and book - so this qualifies the sentence to be processed further. The clause "if his credit rating is adequate" indicates that this is a condition which needs to be tested. There are many words which can be used to identify a condition. Besides the word "if" there are other words like "should", "provided", "when", etc. there are also word patterns like "in case of" and "as long as". When they occur the statement is assumed to be a condition.

If the sentence is not a condition it may be a state declaration. A state declaration is when a relevant object is declared to be in a given state, i.e.
"The customer must be registered."
The word "customer" is a selected object and the word pattern "be registered" indicates a state that the object is in. Predicate verbs such as "be, is, are, were, etc" denote that this is a state declaration.

If the sentence is neither a condition nor a state, it may be an action. An action is indicated by a verb which acts upon a selected object e.g.
"The system checks the customer order."
Here the order is a relevant object and "checks" is a verb which acts upon it. Normally these verbs will end with an "s" if they are in present tense and with "ed" if they are in past tense. So this makes it easier to recognize them. The advantage of requirement texts as opposed to texts in general is that they are almost always written in the third person, thus reducing the number of verb patterns to be checked.

Sentences which qualify as a statement to be tested are extracted from the text and stored in the test case table. Assuming that all sentences are embedded in the text of a section,, a requirement or a use case, it is possible to assign test cases to individual requirements, use cases or simply to titled sections. If a test case originates from a requirement it receives the number or title of that requirement. If the test cases are created from a use case, then they bear the title of that use case. If these structural elements are missing the test case is simply assigned to the last text title. Relations are also established between test cases and objects. Test cases extracted from a particular sentence will have a reference to the objects referred to in that sentence.

A generated test case will have an id, a purpose, a trigger, a pre-condition and a post-condition. The id of the test case is generated from the system name and a running number. The condition "if the customer's credit rating is adequate, he can order a book" implies two pre conditions

1. the customer's credit rating is adequate
2. the customer's credit rating is not adequate

There are also two post conditions

1. the customer has ordered a book
2. the customer has not ordered a book

This shows that for every conditional clause there should be two test cases

- one which fulfils the condition, and
- another which does not fulfil the condition.

They both have the same trigger, namely "the customer orders a book".

These are samples of functional test cases. Non–functional test cases are all either states or conditions. The sentence „The system should be able to process at least 2000 transactions per hour" is a state denoted by the verb „should be". The sentence „In case of a system crash, the system has to be restarted within 2 minutes" is a condition determined by the predicate „In case of", followed by an action „restarted". Both requirements must be tested. The tool itself can only distinguish between functional and non–functional test cases based on the objects acted on or whose state is checked. Here again the user must interact by marking those objects such as „system" which are not part of the actual application.

4.4 Storing the potential test cases

The result of the text analysis is a table of potential system test cases. If the requirements document is structured so that the individual requirements are recognizable, the test cases will be ordered by requirement. If there are use case definitions, the test cases extracted from a particular use case will be associated with that use case. Otherwise, the test cases will be grouped by subtitles.

In the end every test case, whether functional or non-functional will have at least the following attributes:

- a test case Id
- a test case purpose = the sentence from which the case was taken
- a test case type = {action | state | condition }
- a precondition
- a post condition
- a trigger
- a reference to the objects involved
- a reference to the requirements being tested
- a reference to the use case being tested

5. Generating a test design

It is not enough to extract potential test cases. The test cases also need to be assigned to an overall test framework. The test framework is derived from the structure of the requirements document.

Requirements should be enhanced by events. An event is something which occurs at one place at one time. Use cases are such events. An account withdrawal is an example of a use case event. A money transfer is another event. Printing out an account statement is yet another event. Events are triggered by a user, by the system itself or by some other system.

In system testing it is essential to test every event, first independently of the other events and then in conjunction with them. An event will have at least two test cases - a positive and a negative outcome, but it may have many. In the case of an account withdrawal, the user may give in a bad PIN number, he may have an invalid card, the amount to be withdrawn may exceed the daily limit or his account may be frozen. There are usually 4 to 20 test cases for each use case.

In generating a test design the text analyzer tool orders the test cases by event. The event is the focus of a testing session. Requirements and essential objects are assigned to an event so that it becomes clear which functions and which objects are tested within a session. If recognizable in the requirements text, the user or system interface for an event is also assigned. This grouping of all relevant information pertaining to an event is then presented in an XML document for viewing and editing by the tester. In so doing, the text analyzer has not only extracted the potential test cases from the requirements, it has also generated a test design based on the events specified.

6. Experience with automated requirements analysis

The German language version of the text analyzer was first employed in a web application for the state of Saxony at the beginning of 2005. The requirements of that application were split up among 4 separate documents with 4556 lines of text. Some 677 essential objects were identified. Specified with these objects were 644 actions, 103 states and 114 rules. This led to 1103 potential test cases in 127 use cases. The generated test case table served as a basis for the test case specification. As might be expected, several test cases were added, so in the end there were 1495 test cases to be tested. These test cases revealed 452 errors in the system under test as opposed to the 96 errors discovered in production giving an error discovery rate of 89%. This demonstrated that the automatic extraction of test cases from requirements documents, complemented by manual test case enhancement is a much cheaper and more efficient way of exposing errors than a pure manual test case selection process [23]. Besides that it achieves higher functional test coverage. In this project over 95% of the potential functions were covered.

Since this first trial in the Saxon e-Government project the German language version has been employed in no less than 12 projects to generate test cases from the requirements text including a project to automate the administration of the Austrian Game Commission, a project to introduce a standard software package for administering the German water ways, and a project to develop a university web site for employment opportunities.

The English language version has only recently been completed, but has already been used in 3 projects – once for analyzing the use cases of a mobile phone billing system, secondly for analyzing the requirements of an online betting system, and thirdly to generate test cases for a Coca Cola bottling and distribution system. In the case of the mobile billing system, a subsystem with 7 use cases was analyzed in which there were 78 actions and 71 rules for 68 objects rendering 185 test cases. The online betting system had 111 requirements of which 89 were functional and 22 were non-functional. There were 69 states, 126 actions and 112 rules for 116 specified objects from which 304 test cases were extracted.

The specification of the Coca Cola distribution system is particularly interesting because it used neither a list of requirements nor a set of use cases, but instead a table of outputs to a relational database. In the first column of the table was the name of the output data, in the second the data description, in the third the algorithm for creating the data and in the fourth the condition for triggering the algorithm. A typical output specification is depicted in Table 1.

Name	Definition	Source	Condition
A400	Total Number of Bottles	XX20 – Quantity from Mobile Device	Transtype <5 (Sampling) Transtype > 7 (Breakage) ARTIDF = '1A' or '1R'

Table 1: Output Specification

For this one output 6 test cases were generated
Transtype <5 & ARTIDF = '1A'
Transtype <5 & ARTIDF = '1R'
Transtype <5 & ARTIDF ! = '1A' & ARTIDF ! = '1R'
Transtype > 7 & ARTIDF = '1A'
Transtype > 7 & ARTIDF = '1A'
Transtype > 7 & ARTIDF = '1R'
Transtype > 7 & ARTIDF ! = '1A' & ARTIDF ! = '1R'

There were 5 output specification tables with an average of 34 outputs per table. All together 458 test cases were generated for the 171 conditions and 108 actions specified in one single document.

7. Conclusions

The case studies noted emphasize how important it is for the text analyzer to be able to process any text no matter how it is structured. Some requirement documents will only contain a list of functional and non–functional requirements as was the case with the betting system. Other documents like the mobile phone billing specification are built around use cases with specific attributes. Then there are requirement documents consisting almost wholly of different table types. A text analyzer must be able to recognize the structural elements of a specification and to parse the natural language statements as well as the arithmetic and logical expressions contained therein [24].

The text analyzer presented here is an attempt to attain these far reaching goals. Due to the diversity of requirement documents, this is no easy task. However, if the goal is to test a system against its requirements then the test cases have to be taken from the requirements documents. There is no alternative to analyzing the requirement texts and, since the requirement texts are formulated in natural language, there is no alternative to natural language processing. There is still much work remaining to be done on refining the text analyzer, but it is a start and it has already proven its usefulness.

References:

1] Kaner, C. / Nguyen, H.: Testing Computer Software, Wiley & Sons, New York, 1999

2] Beizer, B.: Black – Box Testing Techniques for Functional Testing of Software and Systems, John Wiley & Sons, New York, 1995

3] Notkin, D. / Xie, T.: „Checking inside the Black-Box – Regression Testing by Comparing Value Spectra", IEEE Trans. on S.E. Vol. 31, No. 10, Oct 2005, p. 869

4] Howden, W.: Functional Program Testing & Analysis, McGraw – Hill, New York, 1987

5] Posten, R.: „Specification – based Testing – What it is and how it can be automated", in IEEE Tutorial „Automating Specification – based Software Testing", IEEE Computer Society Press, Los Alamitos, CA, 1996, p. 9

6] Warmer, J. / Kleppe, A.: The Object Constraint Language – precise modeling with UML, Addison – Wesley, Reading MA, 1998

7] Ebert, C.: Software Requirements Management, dpunkt Verlag, Heidelberg, 2006

8] IEEE: ANSI/IEEE Std. 830- Guide to Requirements Specification, IEEE Standards, Piscataway, N. J., 1998

9] Rupp, C.: Requirements Engineering and Management, Hanser Verlag, München, 2001

10] Firesmith, D.: „Use Case Testing Pattern", Object Magazine, No. 5, Vol. 9, Jan. 1996, p. 32

11] Lee, J./ Xue, N.-L.: „Analyzing User Requirements by Use Cases – A Goal driven Approach", IEEE Software, July, 1999, p. 92

12] Antoniol, G. / Canfora, G. / DeLucia, A.: „Maintaining Traceability during Object – oriented Software Evolution", IEEE Proc. of 15th ICSM, Sept. 1999, p. 211

13] Howden, W.: „Functional Program Testing", IEEE Trans..S.E., Vol. 6, No.2, March 1980, p. 162

14] Peters, D./ Parnas, D.: „Using Test Oracles generated from Program Documentation", IEEE Trans. on S.E., Vol. 24., No. 3, March, 1998, p. 161

15] Bach, J.: „Reframing Requirements Analysis", IEEE Computer, Vol. 32, No. 6, 2000, p 113

16] Poston, R.: „Action-driven Test Case Design" in Tutorial: Automating Specification – based Software Testing, IEEE Press, Los Alamitos, CA., 1996, p. 47

17] Beizer, B.: Software Testing Techniques, van Nostrand Reinhold, New York, 1983, p. 107

18] Michael, C. / McGraw, G. / Schatz, M.: „Generating Software Test Data by Evolution", IEEE Trans. on. S.E., Vol. 27, No. 12, Dec. 2001, p. 1085

19] Yang, J./ Coppit, D.: „Software Assurance by bounded exhaustive Testing", IEEE Trans. on. S.E., Vol. 31, No. 4, April 2005, p. 328

20] Fewster, M. / Graham, D.: Software Test Automation, Addison – Wesley, Harlow, G.B., 1999

21] Miller, T.W.: Data and Text Mining – a business application approach, Prentice-Hall, Upper Saddle River, N.J., 2005

22] Miriyala,K. / Harandi,M.: "Automatic derivation of formal software specifications from informal Descriptions", IEEE Trans. On S.E., Vol. 17, No. 10, Oct. 1991, p. 1126

23] Sneed, H.: „Testing an eGovernment Website", in Proc. of 7th Web Site Evolution, WSE 2005, Computer Society Press, Budapest, Sept. 2005

24] Haumer, P. / Pohl, K. / Weidenhaupt, K.: „Requirement Elicitation and Validation with Real World Scenarios", IEEE Trans. on S.E., Vol 24., No. 12, Dec. 1998, p. 1089

This document was created with Win2PDF available at http://www.daneprairie.com.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.