Available online at www.sciencedirect.com
SCIENCE DIRECT®

ELSEVIER Electronic Notes in Theoretical Computer Science 116 (2005) 3–15

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Introducing Test Automation and Test-Driven Development: An Experience Report

Lars-Ola Damm[1,2,3,4] Lars Lundberg[2,6] David Olsson[3,6]

Abstract

This paper identifies and presents an approach to software component-level testing that in a cost effective way can move defect detection earlier in the development process. A department at Ericsson AB introduced a test automation tool for component-level testing in two projects together with the concept test-driven development (TDD), a practice where the test code is written before the product code. The implemented approach differs from how TDD is used in Extreme Programming (XP) in that the tests are written for components exchanging XMLs instead of writing tests for every method in every class. This paper describes the implemented test automation tool, how test-driven development was implemented with the tool, and experiences from the implementation. Preliminary results indicate that the concept decreases the development lead-time significantly.

Keywords: Software test automation, test-driven development, component testing, unit testing, test tool.

1 Introduction

Studies indicate that testing accounts for at least 50% of the total development time [10], [11]. One reason for this is that the verification activities late in development projects tend to be loaded with defects that could have been prevented or at least removed earlier (when they are cheaper to find and

[1] This work was jointly funded by Ericsson AB and The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.ipd.bth.se/besq).
[2] Dept. of Sw. Eng. and Comp. Sc., Blekinge Institute of Technology, Ronneby, Sweden.
[3] Ericsson AB, Soft Center, SE-372 25 Ronneby, Sweden.
[4] Email: Lars-Ola.Damm@Ericsson.com
[5] Email: Lars.Lundberg@bth.se
[6] Email: David.Olsson@Ericsson.com

doi:10.1016/j.entcs.2004.02.090

remove [5], [10], [21]). When many defects remain to be found late in a project, schedules are delayed and the verification lead-time increases [13].

A software development department at Ericsson AB (from now on referred to as the department) develops component-based software for the mobile network. The department wanted to decrease the verification lead-time and avoid the risk for delayed deliveries by introducing a new tool and process for automated testing on a component level. To achieve this, they needed to determine what was required of the tool, process and organization.

Thus, the paper has four main questions to answer:

(i) What characteristics should a test automation tool for component level testing have?

(ii) What is an appropriate process supporting the use of such a test automation tool?

(iii) What aspects need to be considered when introducing a new process and tool for component level testing?

(iv) What is the expected lead-time difference for the projects that introduce such a tool and process?

The method used for answering the first two questions was a combination of analysis of lessons learned from previous improvement attempts together with a thesis study [7] on how to increase the test efficiency at the department. The thesis study included qualitative and quantitative enquires, analysis of project statistics, and a literature study. Answers to the other two questions were captured from qualitative and quantitative interviews with users of the introduced concept.

The paper is outlined as follows. Section 2 gives a background including the choice of process for the concept, i.e. test-driven development. Section 3 presents the actual implementation together with some lessons learned and expected lead-time gains. Section 4 maps the new test automation tool against related techniques for test automation and then also describes what is required of an organization before being able to implement such a concept. Section 5 summarizes the work through some conclusions.

2 Background

The approach used before the introduction of this new concept comprised a test tool (called DailyTest) that could test isolated components before delivery to the function test department. In this context, a component is an executable asset that communicates with other components through common interfaces. Further, a component contains about 5-30 classes and the products that the department develops consist of about 10-30 components each.

DailyTest could execute scripts consisting of simple commands that for example loads a component and then sends a sequence of requests on it. The reasons why the tool did not work satisfactory were that it was limited (few commands and no looping) and more importantly, it was not properly integrated with the development process. When the department had realized this, the earlier mentioned thesis study [7] evaluated the test process with the purpose to identify tool and process changes that would increase the test efficiency. The thesis study showed that the cost of finding and fixing defects increases significantly the later in the development process they are found. This is also considered a common fact in software development [5], [10], [21]. Since the thesis study also discovered that the developers normally put little effort on testing isolated components, the thesis suggested that this is where to focus the improvement efforts.

Testing of isolated software components is the first test level in the department's quality assurance strategy, Basic Test in Ericsson terminology. As in ordinary unit testing, the purpose of Basic Test is to verify the design specification. However, as opposed to unit testing, Basic Test is not a white-box test technique since the components' interfaces hide the internal component design. Nevertheless, since it is the product developers that perform Basic Test on the components, they still know the internal design structure. The reason why the developers start doing testing on a component level is because it is not cost-effective for them to test every class/method. The reason for this is that the department's products have a higher testability on the component level, i.e. the components are independent executables and the XML requests that the components send between each other are easier to verify (testability is further discussed in Section 4.2.2).

After determining that the improvement efforts should focus on the Basic Test level, the thesis study determined that the main reasons for why the developers at the department did not Basic Test all functionality was because of insufficient test tools (e.g. DailyTest) and process deviations when the deadline pressure was high due to delayed schedules. When the development activities were delayed, the projects tended to deliver the code untested hoping that it in a miraculous way would work anyway. Likewise, this phenomenon seems far from uncommon in the software industry [14], [21].

From the experiences and findings discussed above, the thesis study suggested a new tool for how to automatically Basic Test the products at the department, which they thereafter implemented and introduced in two upcoming projects.

2.1 Test-driven development

To make sure that the new Basic Test tool would not become a shelfware, the department put considerable efforts in integrating the tool with the development process and they chose between keeping their previous standard process or to introduce the new concept test-driven development (TDD) [2].

The main difference between TDD and a typical test process is that in TDD, the developers write the tests before the code. A result of this is that the test cases drive the design of the product since it is the test cases that decide what is required of each unit [1], [2]. Therefore, TDD is not really a test technique [1], [6]; it should preferably be considered a design technique. Furthermore, TDD simplifies the design and makes sure that the implementation scope is explicit, i.e. it removes the desire for gold plating [1]. Nevertheless, the most obvious advantage of TDD is the same as for test automation in general, i.e. the possibility to do continuous quality assurance of the code (including regression testing) [9]. This gives both instant feedbacks to the developers about the state of their code and most likely, a significantly lower percentage of defects left to be found in later testing and at customer sites [19]. Further, with early quality assurance, a common problem with test automation is avoided; that is, when an organization introduces automated testing late in the development cycle, it becomes a catch for all defects just before delivery to the customer. The corrections of found defects lead to a spiral of testing and re-testing which delays the delivery of the product [18].

The most negative aspect concerning TDD is that in worst case, the test cases duplicate the amount of code to write and maintain. However, this is the same problem as for all kinds of test automation [12], and to what extent the amount of code increases depends on the granularity of the test cases and what module level the test cases encapsulates, e.g. class level or component level. Thereby, since the department would use TDD on a component level, they would decrease the amount of test case code to write and maintain. Additionally, in comparison to classes/methods, it is easier to automate uniform interfaces that use XML as data format and that are more robust to changes.

Nevertheless, the department foremost chose to use TDD because it can eliminate the previously mentioned risk for improperly conducted Basic Test. When the test cases are developed before the code, it is consequently impossible to deliver the code without developing the test cases. Meanwhile, there is no reason not to Basic Test the product when the executable test cases already are developed. To summarize, the primary purpose of the new concept was to increase the amount of tested code in Basic Test.

3 Description of the Basic Test concept

After providing some background information, sections 3.1 and 3.2 give a technical tool description and Section 3.3 describes how the tool was integrated with the development process at the department. After that, Section 3.4 lists some observations and lessons learned and finally, Section 3.5 presents the expected lead-time gains from introducing the concept.

3.1 Choice of tool and language

Since the purpose of Basic Test is to test the components in isolation, the Basic Test tool needed to be attached to the components' interfaces, i.e. simulating

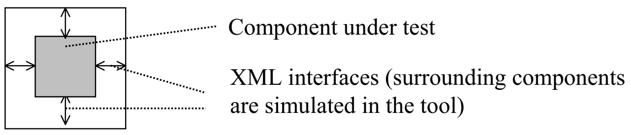
Component under test
XML interfaces (surrounding components are simulated in the tool)

Fig. 1. Basic Test tool - attachment to components

the surrounding components. This attachment is demonstrated in Figure 1.

DailyTest, the previous Basic Test tool the department used, attached to the components in a similar way as in Figure 1. The reason why it was not preferable to enhance DailyTest instead of developing a new tool was that to make DailyTest as powerful as needed, it would almost become a program language. Naturally, it is not beneficial to do that when there already exist several powerful standard languages such as C++ and Java.

After choosing not to enhance the existing test tool, the department needed to decide whether to develop a new tool using a standard language or buying a commercial tool. First of all, the TDD tools normally used in Extreme Programming (e.g. cppUnit) [2] were not appropriate in this context since they operate on a class/method level; that is, they do not support component level interfaces (e.g. XML communication). Further, since Basic Test requires tight integration with the product architecture, the test tools that commercial vendors develop are hard to use for this purpose since they require some kind of interface that can connect to the components to test. Such an adaptation would involve additional costs and more importantly, such tools would put unwanted constraints on how to develop, execute, and monitor the tests. This combined with the fact that it was relatively cheap to develop an in-house tool when the product architecture had such high testability (see Section 4.2.2) made it preferable to develop the tool in-house. Regarding the choice between using test case generators (see Section 4.1) or writing them manually, test case generators were not considered beneficial for the department. This because the department thought that it would be more cost-effective to write the design specifications as executable test cases directly instead of first writing formal design specifications manually and then generate test cases from them.

Next, the department needed to choose a language to write the test cases in. The major benefits of using standard script languages as for example Visual Basic and JavaScript are that programmers tend to be less error-prone compared to when using system-programming languages [17]. Still, the department chose to use C++ as test case language. Firstly, because the developers do not need any training on how to use it since they write the product code in it. Secondly, C++ has the power to handle features that are not included in the actual tool, e.g. it is possible to make direct calls on functions in the product code when the tool has no support for it implemented. Finally, when the test cases are written in the same language as the product code, the developers can take advantage of the programming tools already available [9], [20].

A, Test code example (C++)

```
Tool.startTest(“Test1”);
  theComponent.startComponent();
  Tool.startTest(“Test1:1”);
    ToolSender.sendMessage(Request.xml,
                                 ExpResult1.xml);
    ToolSender.sendMessage(Request2.xml,
                                 ExpResult2.xml);
  Tool.endTest();
  Tool.startTest(“Test1:2”);
     ToolReceiver.receiveMessage(ExpRes3.xml);
  Tool.endTest();
  theComponent.stopComponent();
Tool.endTest();
```

B, Test result example (XML)

```
<Test name=“Test1” status=”Failed”>
  <Statistics>
       <StartTime>2003:01:01-12.24</>
       <EndTime>2003:01:01-12.25</>
       <NumberExecutedTests>2</>
       <NumberPassedTests>1</>
  </Statistics>
  <Test name=“Test1:1” status=”Failed”>
    <Request1>
       <Result>Ok</Result>
    </Reques1>
    <Request2>
       <Result>Not Ok</Result>
        <ResultFile>ComparisonRes.xml</>
    </Request2>
  </Test>
   <Test name=“Test1:2” status=”OK”>
    <Request1>
       <Result>Ok</Result>
    </Request1>
  </Test>
</Test>
```

Fig. 2. Basic Test tool - Example

3.2 Test case syntax and output style

After deciding to develop an in-house Basic Test tool with C++ as test case language, the department chose to use the framework-driven approach when designing the tool, i.e. the tool isolates the component to test from the test cases through wrappers and utility functions (see Section 4). Figure 2:A shows an example of a test that was implemented in the new Basic Test tool.

In addition, the tool contains several commands for handling component states and for sending and receiving requests. All actions are controlled by the tool and during the execution, the tool compares each sent/received request with its expected result and then logs the result in an XML file. The XML requests are generated from their corresponding DTD’s (Data Type Definition).

Figure 2:B shows an example of a log output from a test execution. The department chose XML as output format because it is already used as standard data format in their products and because nowadays, XML is a standard format to which many other tools and parsers easily can be attached. For example, it is easy to develop a GUI that parses the XML data into tree structures of test results (according to the tag structure in the XML). In such a GUI, the developers can monitor their test executions and the managers monitor the test progress.

3.3 Adjustments to the development process

Just providing a good tool does not ensure successful test automation; a tool only becomes as good as the people using it [21]. Since this tool was to be used with TDD (see Section 2.1), the test cases should be written before the code. As earlier mentioned, the department introduced TDD on a component

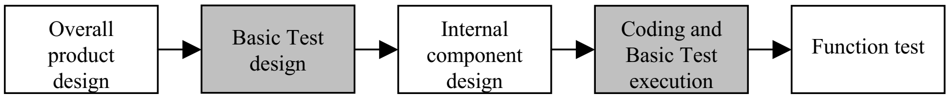

Fig. 3. Basic Test process at the department

level instead of on a class level, i.e. the developers construct a set of test cases for each component instead of a set of test cases for each class. The test strategy was to capture all inputs and outputs of each component. Thereby, the tests represent the external design of each component. This also led to test cases that could serve as a part of the design documentation, replacing some of the old design (e.g. component specifications). The result of this was a more thorough design (in comparison to plain English, C++ does not leave room for misinterpretations) and that some design time could be saved when being able to remove some of the old design documentation.

Figure 3 displays how the Basic Test concept was incorporated with the process levels at the department. The new activity "Basic Test design" is when the developers construct the components' test cases and it comprises both implementation and inspection of the test cases.

With a tool and a process for how to use the tool established, there is only one major thing left to put in place: a standard for how to write test cases. Otherwise, the department would end up with spaghetti tests that are hard to understand and maintain leading to that the benefits with test automation would be lost [9]. Test automation is development and the test cases are software programs testing other software programs; therefore, the test cases are subject to the same design and construction rules as the programs to test [21], [22]. Thus, the tests should follow ordinary guidelines for structured programming so that they become simple, reusable and easy to understand and maintain. Further, the tests should be independent [17] since this enables the possibility to only execute individual tests when short execution time is crucial. Additionally, when having independent tests, the source of defects can only be within that test, which makes it is easier to locate the source of a defect. In the Basic Test tool, the 'startTest'/'endTest' constructs (see Figure 2) enable such a test independence. Finally, to enable daily regression testing, the tests must be constructed for repeatability, i.e. it must be possible to execute them as regression tests without human interaction [19]. To be able to track and repair the defects found in regression testing, good naming conventions for the tests are necessary. As can be seen in Figure 2, all tests have a nametag which the department used for specifying what feature each test is supposed to verify. Such traceability makes it possible to at all times know the progress of each feature (percent developed/passed).

To ensure that the developers would follow the standard for writing test cases, someone should inspect all test cases before letting the developers start implementing the product code. Moreover, the inspections must ensure that the tests not just show that the code will work, but also the opposite [23]. To achieve that, adequate test coverage must be obtained, e.g. cover all behavior variants (including error cases and boundary values).

Another challenge the department had to address was to implement the new Basic Test concept on products that already existed since several years and therefore, the components contained a lot of old functionality that did not have such tests. Since developing tests for all old functionality when introducing the tool would be far too costly for a single project to handle, the department chose only to develop tests for new and modified functionality. Henceforth, the strategy was to develop tests for more and more of the old functionality during upcoming projects (i.e. in future releases of the product).

3.4 Observations and lessons learned

During the implementation and introduction of the new concept, several observations were made and some lessons were learned. This section provides a list of those that were identified by two of the authors of this paper that participated in the target projects and from qualitative interviews with managers and developers at the department after the introduction of the concept.

The department established that...

...test automation requires high product testability. According to related work, it is the product interfaces that determine the opportunities for test automation [17]. The notion of testability is further discussed in Section 4.2.2.

...as also reported in [21], it is important with a thorough framework design because it makes test case development easier and is robust to future changes.

...test automation on a component level requires adjustments to the product architecture since component-level test architectures must have a tight integration with their product architectures (also acknowledged in [20]).

...making people add new tests every time a new defect is found requires continuous remainders and monitoring. Otherwise, developers tend to deliver the bug fixes untested. Furthermore, the authors in [19] state that adding new tests for each fault strengthens the test suite.

...test execution speed is important. Other studies support this experience with the claim that the faster the test execution speed is, the more likely developers will execute the tests themselves [19].

...what gets measured gets done [13], [23]. When the department started the introduction of the new concept for Basic Test, the target projects gave it modest attention. However, when the projects started measuring the progress for number of test cases developed and passed, the usage rates increased.

...beware of attempts to deviate from the agreed process. When in time-pressure, projects tend to neglect some activities which might give near-time benefits but that might be devastating in the long run [13]. Therefore, such deviations should not be allowed without being agreed on by all involved parties (e.g. the line organization).

. . . test-driven development makes people think through the design more instead of rushing to coding directly without knowing what to implement yet.

. . . when introducing new ways of working, it requires significant efforts to convince people in the organization that the new methods will improve the productivity. If not succeeding to do this, the introduction of the new methods will most likely fail due to resistance among managers and developers to accept the new ways of working (further discussed in Sections 4.2.1 and 4.2.2).

. . . test tools easily become the excuse for every problem [8]. Reasons for a problem can be in either the test tool, the development environment or in the application under test; still, the Basic Test tool became the scapegoat for most problems since it is in the tool the problems first are discovered no matter where they origin.

. . . automated testing requires dedicated resources. As also considered a common fact [17], [21], test automation cannot be managed as a spare-time activity. In comparison, developing or buying the tool is rather cheap; it is activities such as setting standards for test case writing, teaching users how to write good test cases, and tool maintenance that are costly.

. . . benefits from test automation are hard to obtain in the first project release. Other reports support this experience by claiming that upfront costs eliminate most benefits in the first project and benefits from regression testing are usually not realized until the second release [16].

. . . test automation should be introduced in small steps, e.g. as a pilot project to avoid taking unnecessary risks (if something is introduced in the wrong way but only in small scale, the cost of fixing the problem is most likely lower). This advice is also given in the research literature [9], [12], [17].

. . . minimizing maintenance costs is the most difficult challenge in test automation. The test cases must be robust to changes during bug fixes and in new product versions. Since this is hard to achieve, the most common problem in test automation is probably uncontrolled maintenance costs [17].

3.5 Expected lead-time gains

This paper does not provide actual results on costs and benefits of the introduced concept. However, preliminary project evaluations indicate significantly decreased fault rates. Further, Figure 4 presents the result of a study where all the developers (in a questionnaire) estimated the lead-time difference after they had used the concept in product version X. On average, they estimated that the project lead-time would decrease more and more when using the concept (e.g. 25% in the third project). Also note that the developers did not think that the introduction costs in the first version delayed the project, i.e. they estimated gains already from the beginning.

Another finding (from qualitative interviews with the project managers) was that not only decreased lead-time was the reason for using the con-

Product version	Expected lead-time difference (development time)
Version X (2003)	-2%
Version X+1 (2003-2004)	-19%
Version X+2 (2004)	-25%

Fig. 4. Expected lead-time gains with new concept

cept; they thought that increased progress control (percent test cases executed/completed), and increased delivery precision (due to increased progress control and improved quality assurance) were at least as important.

4 Discussion

4.1 Related techniques for test automation

The purpose of this section is to relate the techniques used by the department's test automation tool with other techniques used in the software test-automation industry. The presented techniques are described together with some of their pros and cons. However, note that the primary objective of this section is to benchmark the tool, not to perform a technique evaluation.

4.1.1 Capture-replay

The basic concept of capture-replay is that a tool records actions that testers have performed manually, e.g. mouse clicks and other GUI events (that later can be re-executed automatically). Capture-replay tools are simple to use [3], but according to several experiences not a good approach to test automation, since the recorded test cases easily become very hard to maintain [9], [16], [17], [21]. The main reason is that they are too tightly tied to details of user interfaces and configurations, e.g. one change in the user interface might require re-recording of 100 test scripts [21].

4.1.2 Script techniques

A powerful approach to test automation is to write some kind of test scripts for the test cases. Script techniques provide a language for creating test cases and an environment for executing them [20]. Approaches to scripting varies in several dimensions:

Simple scripts *versus* highly structured scripts: Scripts may vary in complexity from simple linear scripts to keyword driven scripts with conditional/looping functionality and further to real programming languages [9].

Standard scripts/languages *versus* vendor scripts: Scripting tools either use a standard script programming language (e.g. Visual Basic, C++), or their own proprietary language (vendor script) [17].

Scripts control flow and actions *versus* data-driven testing: This choice is about whether the script or the input data controls the execution. In data-driven testing, an input test data file control the flow and actions [21].

Standard scripting *versus* framework driven testing: Instead of operating against the product interfaces directly, framework driven testing adds another layer of functionality to the test tool where the idea is to isolate the software from the test scripts. A framework provides a shared function library that becomes basic commands in the tool's language [16], [21].

4.1.3 Test-case generators

Test case generators are the most advanced test automation tools. They exist in several variants: structural generators (generate test cases from the structure of the code); data-flow-generators (use the data-flow between software modules as base for the test case generation); functional generators (generate test cases from formal specifications); and random test data generators [3]. Test case generators can generate several test cases fast but are still not always more cost-effective since expected results need to be added manually. Further, the generated test cases/executions of the test cases must also be checked manually to verify that they test the right functionality.

4.2 Other considerations

Before implementing a new concept as the one described in this paper, both the organization and its products must fulfill some prerequisites; otherwise, the risk for failure is substantial. This section describes a few prerequisites in relation to the situation at the department.

4.2.1 Maturity of the organization

First, the development process that the developers follow needs to be mature enough. If the process is poor, test automation will not help [9], [17]. Preferably, the test automation effort should be easy to adapt to current practices; if the change is too great, the risk for resistance among the developers increases [15]. Furthermore, if the developers do not want to work as directed, they will not [13]. At the department, the developers were aware of that neglecting Basic Test results in increased verification lead-time [7]. Therefore, they were open to improvements in Basic Test. Second, the managers must be committed to the new methods because it is they that have to grant the upfront costs with introducing test automation (that most likely will impact short-term budgets and deadlines negatively [4]). Test tool costs is just the tip of the iceberg [9], [17].

4.2.2 Maturity of the products (testability)

A product with high testability provides interfaces that are easy to develop test cases for, robust to changes, and whose data is easy to represent in test cases together with expected outputs that the test execution tool automatically can verify against the received outputs. The more testable the software is, the less effort developers and testers need to locate the defects [20]. For example, the

Basic Test tool was significantly cheaper to build when the products had a common communication interface to simulate.

5 Conclusions

The Ericsson department introduced a new test automation tool incorporated with an alternate approach to Test-Driven Development (TDD), i.e. TDD on a component level where the interfaces comprise socket connections exchanging XML data instead of classes and methods. With such an approach to test-driven development, robust and uniform component interfaces make test automation easier.

The main characteristic of the tool is that it uses C++, the same standard programming language as the developers write the product code in, because the developers are already familiar with it, it is more powerful than a script language, and the developers can take advantage of programming tools already available (e.g. compilers and debuggers).

Regarding process support, the software development department introduced the concept TDD to support the tool mostly because:

- When writing the test cases before the code, testing really happens.
- The test cases drive the design and make it more straightforward, i.e. with an explicit scope and no gold plating.
- TDD moves fault detection earlier in the development process when the faults are cheaper to find.

There are several aspects to consider when implementing test automation and TDD. Section 3.4 lists the most important ones (in form of observations and lessons learned).

The developers that have used the concept have estimated that the project lead-time will decrease more and more for each new project version that uses it (see Section 3.5). Further, preliminary project evaluations indicate significantly decreased fault rates from the introduction of the new concept.

References

[1] K. Beck, "Aim, Fire [test-first coding]", *IEEE Software*, vol. 18, no. 5, 2001, pp. 87-89.

[2] K. Beck, Test Driven Development – by example, Addison Wesley, 2003.

[3] B. Beizer, *Software Testing Techniques, 2nd Edition*, Van Nostrand Reinhold Company, New York, 1990.

[4] M. Berwick, "Optimising the Development and Test Process", in Kelly, M.: *Management and Measurement of Software Quality*, UNICOM SEMINARS, Middlesex, UK, 1993, pp. 51-64.

[5] B. W. Boehm, *Software Engineering Economics,* Prentice-Hall, 1981.

[6] A. Cockburn, *Agile Software Development*, Addison-Wesley, 2002.

[7] L-O. Damm, "Evaluating and Improving Test Efficiency", *Master Thesis: Blekinge Institute of Technology*, Dept. of Software Engineering and Computer Science, June 2002, MSE-2002-15, pp. 60.

[8] M. Fewster, "Test Automation Using an In-house Testing Tool: a Case History", in Kelly, M.: *Management and Measurement of Software Quality*, UNICOM SEMINARS, Middlesex, UK, 1993, pp. 193-204.

[9] M. Fewster, D. Graham, *Software Test Automation*, Addison-Wesley, 1999.

[10] B. Hambling, "Realistic and Cost-effective Software Testing", in Kelly, M.: *Management and Measurement of Software Quality*, UNICOM SEMINARS, Middlesex, UK, 1993, pp. 95-112.

[11] M. J. Harrold, "Testing: a roadmap", *International Conference on Software Engineering,* ACM, 2000, pp. 61-72.

[12] L.G. Hayes, *The Automated Testing Handbook,* Software Testing Institute, 1995.

[13] W.S. Humphrey, *Winning with Software*, Addison-Wesley, 2002.

[14] D. Ince, "Some of the Questions to Ask about Quality Assurance", in Kelly, M.: *Management and Measurement of Software Quality*, UNICOM SEMINARS, Middlesex, UK, 1993, pp. 9-18.

[15] A. K. Johnston, "Muzzling the Alligators: a Pragmatic Approach to Quality", in Kelly, M.: *Management and Measurement of Software Quality*, UNICOM SEMINARS, Middlesex, UK, 1993, pp. 19-30.

[16] C. Kaner, "Pitfalls and Strategies in Automated Testing", *Computer*, IEEE, vol. 30, no. 4, 1997, pp. 114-116.

[17] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing,* John Wiley & Sons, Inc, New York, 2002.

[18] A. Kehlenbeck, "Commentary: Automated test tools have not delivered on the quality promise...yet", *Software Magazine*, Proquest, 1997.

[19] E. M. Maximilien and L. Williams, "Assessing test-driven development at IBM", *Software Engineering: Proceedings. 25th International Conference on*, IEEE, 2003, pp. 564-569.

[20] J. McGregor, "Testing a Software Product Line", *Technical Report CMU/SEI-2001-TR-022,* Carnegie Mellon University, Software Engineering Institute, December 2001.

[21] D. J. Mosley and B. A. Posey, *Just Enough Software Test Automation*, Prentice Hall, 2002.

[22] R. Patton, *Software Testing*, Sams Publishing, 2001.

[23] S. R. Rakitin, *Software Verification and Validation for Practitioners and Managers, Second Edition*, Artech House, 2001.