Overview of the Test Driven Development Research Projects and Experiments

Aleksandar Bulajic
Metropolitan University, Belgrade, Serbia
aleksandar.bulajic.1145@fit.edu.rs

Samuel Sambasivam
Azusa Pacific University, Azusa, CA, USA
ssambasivam@apu.edu

Radoslav Stojic
Metropolitan University, Belgrade, Serbia
Radoslav.Stojici@fit.edu.rs

Abstract

Benefits offered by Test Driven Development are still not fully exploited in industrial practice, and a number of projects and experiments have been conducted at universities and at large IT companies, such as IBM and Microsoft, in order to evaluate usefulness of this approach. The aim of this paper is to summarize results (often contradictory) from these experiments, taking into account the reliability of the results and reliability of the project design and participants. Projects and experiments selected in this paper vary from projects that are accomplished at universities by using undergraduate students to project accomplished by professionals and industrial teams with many years of experience.

Keywords: Agile software development, Test Driven Development, TDD Research Projects, TDD, Test First.

Introduction

There is no doubt that Test Driven Development (TDD) approach is important shift on the field of software engineering. Among many benefits that the TDD claims, the focus in this paper is on productivity, test coverage, reduced number of defects, and code quality. A lot of researchers analyzed the TDD effectiveness comparing it with the traditional (waterfall) approach.

Even though this software development method has been introduced more than ten years ago, there is still doubt regarding to all benefits that this approach claims (Causevic, Sundmark, & Punnekkat, 2011; Kollanus, 2011; Siniaalto & Abrahamsson, 2007).

Causevic et al. (2011) selected 48 relevant papers from a large number of TDD research papers in order to identify the limiting factors of the industrial adoption of the TDD approach.

While the most of authors did not make discrimination between negative, neutral or positive research results effects, Kollanus (2011) in his paper moved focus to critical viewpoints on TDD. Kollanus

used the same guidelines for systematic literature reviews as Causevic et al. (2011), and researched scientific journals, magazines and conference proceedings. Kollanus (2011) identified 40 empirical projects.

However, the analyzed projects design and reported results illustrate challenges that every researcher meets when research project designs and reported results are not standardized. Some of the reported conclusions were contradictory to each other. Even the most of presented project on the first look seem to use the same or very similar design when some of the important details are taken in account the picture about project similarities can be radically changed, as well as question how much of the TDD approach is really used can quickly be raised.

Thus, the review papers did not provide reliable information about particular projects design and results. These papers rather summarize results and made conclusion based on number of positive and negative results and TDD effects. On the other hand, they implied that TDD is more effective without making differences related to a project type, to a project size and environment where research projects were accomplished.

We adopt the hypothesis that effectiveness of TDD should be considered in the context of specifically class of projects (e.g. small company at level 1 of CMMI or large company at level 3, 4 or 5), postulating that developers skill and process maturity is very important factor. Thus, we select the projects accomplished by leading software companies such as IBM and Microsoft and respected research institutions such as North Carolina State University, Karlsruhe University and VTA Research Finland. The primary focus in these projects are results related to productivity, test coverage, defect density reduction, code quality, and what fits very well to primary aim of this paper, which are, base conclusions on analysis of reliability of the results, as well as reliability of the project design and participants.

This paper will try to give an answer, based on conducted research projects and experiments, what kind of benefits can be verified and confirmed by collected evidence, and how reliable are sources of information. Except to review and present results of the huge number of the empirical research projects accomplished on the Universities and in the different companies, our focus is on the reference cases that are most used in the literature and research projects as reference cases for the TDD research project design and as support for conclusions related to the TDD advantages and weaknesses.

The “Test Driven Development” section introduces the TDD and provides more details about each step of the TDD process described in the Figure 1 and is reused from previous authors work related to TDD (Bulajic & Stojic, 2011)

The “TDD Research projects and Experiments” section contains a collection of research projects and experiments and is divided to research projects and experiments where project teams participants were professionals and industrial teams and research projects and experiments where participants were students.

The “Research Projects and Experiments Design and Results Analysis” section contains a critical analysis of the selected research projects and experiments design and reported results.

Both “This paper Contribution” and the “Conclusion” sections contains summary of this paper contribution and final authors’ conclusion.

Test Driven Development

Test Driven Development (TDD) rules defined by Kent Beck (Beck, 2002) are very simple:

1. Never write a single line of code unless you have a failing automated test,
2. Eliminate duplications.

The first principle is fundamental for the TDD approach because this principle introduces technique where a developer first writes a test and then implementation code. Another important consequence of this rule is that test development is driving implementation. Implemented requirements are by default testable; otherwise, it will not be possible to develop a test case (Bulajic & Stojic 2011).

Second principle, today is called Refactoring, or improving a design of existing code. Refactoring also means implementing a modular design, encapsulation, and loose coupling, the most important principles of Object-Oriented Design, by continues code reorganization without changing existing functionality. Figure 1 (Bulajic & Stojic, 2011) illustrates a Test Driven Development process:

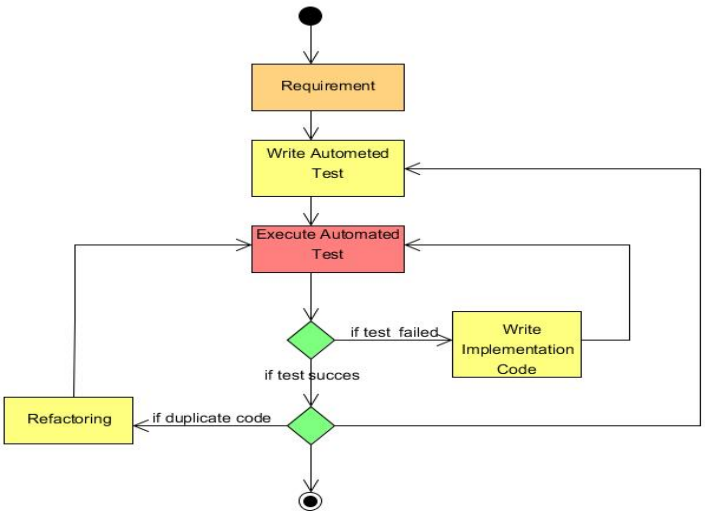

Figure 1 Test Driven Development workflow diagram

The TDD process steps are described as:

1. Requirement,
2. Write an Automated Test,
3. Execute the Automated Test,
4. Write Implementation Code and repeat step 3 as long as the Execute Automated Test fails,
5. Refactoring of existing code when test is executed successfully,
6. Repeat the whole process by going to step 1 and implementing other Requirements.

Availability and using of the *automated testing tool* is mandatory. A test is chosen from a tests list that is already created during brainstorm sessions. This list contains a list of tests that "*verifies requirement and describes the completion criteria*". There are few important things that shall be remembered (Newkirk & Vorontsov 2004):

1. The first implemented test does not have to be the first test on the list. The developer is free to write the code for any test on the list in any possible order,
2. The test list is not a static and adding or changing tests is welcome,
3. The test shall be as simple as possible. Developer should remember that Unit test is testing a particular class or a particular method inside of the same class,
4. The new test shall fail; otherwise there is no reason to develop a test.

A decision to stop further development can be made when all test cases from a test case list are implemented, all tests passed (all tests are green), and all test code is covered by implementation code. This means that all requirements are implemented and according to the TDD methodology, there is no reason to use more time and resources for further development. More detailed discussion is given in Bulajic and Stojic 2011.

Related Work

Except to use a lot of time on a literature survey we choose to provide references to the projects that already did research of empirical projects that are previously accomplished and made project result analysis. Causevic et al. (2011) followed guidelines recommended by Kitchenham and Charter in the "Guidelines for Performing Systematic Literature Reviews in Software Engineering" and collected documents and papers from the following databases':

- IEEExplore,
- ACM Digital Library,
- ScienceDirect,
- El Compendex,
- SpringerLink,
- ISI Web of Science,
- Wiley Inter Science Journal Finder.

The initial search that selected 9462 papers were further refined and reduced to 48 papers, according to the following four rules, where first two rules were mandatory, together with one of the last two rules where one or the other rule shall be satisfied (Causevic et al., 2011):

1. The paper shall be research paper,
2. Shall be evaluation,
3. The research aims shall be clearly stated,
4. Context/setting shall be adequate described.

While Causevic et al. (2011) did not make discrimination between negative, neutral or positive research results effect, Kollanus (2011) moved focus to critical viewpoints on TDD. Kollanus (2011) identified 40 empirical projects. Even these two papers provided good overview of the research projects and empirical experiments, these papers did not provide sufficient information about particular projects design and results analysis. These papers rather summarize results and made conclusion based on number of positive and negative results and TDD effects.

Selected TDD Research Projects and Experiments

In the following sections are presented design details and result analysis of the research projects that are accomplished by leading software companies such as IBM and Microsoft and respected research institutions such as North Carolina State University, Karlsruhe University and VTA Research Finland.

These projects design and reported results illustrate challenges that every researcher meets in case when research project design and reported results are not standardized. Even the most of presented project on the first look seems to use the same or very similar design, when some of the important details are taken in account the picture about project similarities can be radically changed, as well as question how much of the TDD approach is really used.

The projects and experiments used developers that were randomly selected and divided into two groups. The first group developed applications by using a TDD approach that is also called a Test-First approach, where they write the test code first and then the implementation code. The

second group acted as a control group and this group developed the same application by using a traditional development approach, or a waterfall approach, also known as a Test-Last approach. Traditional approach, Waterfall or Test-Last approach, in this case have the same meaning and describes approach where the code is written first and then is written a test code.

Another experiment design used the same group of developers and let this group develop a project by using the traditional approach and then develop a project by using a TDD approach.

The following sections contain researcher papers, case studies, and conclusions which are based on the experiments results. After reading of many of the papers that published research results on the TDD we found that there are basically two kinds of research projects:

1. Research projects accomplished by using graduate and undergraduate students,
2. Research projects accomplished by using professionals and industrial teams.

Even though both kinds of these projects provided documented results, we were in doubt how reliable results were. While the most of research projects and experiments did not take in account differences between participants' skills, experience or professionalism, and made conclusions based on the experiments' results, mixing these results without making these important differences can make confusion and correct conclusion.

Numbers of participants, as well as team size are important. We expect that more participants and more different teams would produce more reliable results. What else we see as important for getting correct picture about the TDD approach advantages and disadvantages, when compared to traditional software development approach, is a problem complexity. While simple problems are best for demonstrating methodology, these are not sufficient to draw reliable conclusion in the research projects and experiments where the primary goal is to find advantages and disadvantages of two different software development methods.

The research projects described in the following sections are using Object-Oriented Metrics (OOM) when compared to the TDD and traditional software development approaches. One example of the used OOM is Chidamber and Kemerer OOM (1994) that originally contains six metrics:

Metrics	Description	Desired value
Weighed Methods per Class (WMC)	Number of methods of a certain class without inherited methods (the sum of the complexities of all class methods; weight is mostly 1)	Low
Depth of Inheritance Tree (DIT)	The depth of a node of a tree refers to the length of the maximal path from the node to the root of the tree.	Low (tradeoff)
Number Of Children (NOC)	Number of immediate descendants of the class.	Low (tradeoff)
Coupling Between Objects Classes (CBO)	Number of couplings between a certain class and all other classes	Lower
Response Set For a Class (RFC)	Number of methods that can be performed by a certain class regarding a received message	Lower
Lack of Cohesion Metrics (LOCM)	Number of disjunctive method pairs (i.e., there exist no *shared* instance variables) of a certain class	Lower

In addition to the above metrics very often are used (Metrics_1.3.6 2012):

Metrics	Description	Desired value
Number of Classes	Total number of classes in the selected scope	Higher
Lines of Code (LOC/TLOC) and (MLOC)	LOC/TLOC: Total lines of code that will counts non-blank and non-comment lines in a compilation unit; MLOC: Method lines of code will counts and sum non-blank and non-comment lines *inside* method bodies	Lower

The list of OOM metrics can be very long and some authors discovered more than 200 of different metrics.

Research Projects Accomplished By Using Graduate and Undergraduate Students

Computer Science Department University of Karlsruhe

This empirical experiment is one of the most often used references in the TDD approach related experiments and case studies. The experiment used 19 students divided in two groups. The ten students used the Test First approach and this group has been called Test First Group (TFG). The rest of the nine students were in Control Group (CG) and this group used traditional software development approach, write implementation code and then write a test code. The both group of students had the similar programming experience that is estimate to median. The purpose of this experiment was (Muller & Hagner, 2002):

- The programming efficiency (how fast someone obtains solution),
- The reliability of code (how many failures can be observed, measured as portion of the passed assertions related to all possible executable assertions in the test),
- Program understanding (measured as proper calls of existing methods).

The experiment was divided in two phases (Muller & Hagner, 2002):

- Implementation Phase (IP), where participants developed code until they believe that the code is ready for acceptance test,
- Acceptance-test Phase (AP), where participants shall fix faults that makes acceptance test to fail.

After first phase (IP), test results shows that reliability of the programs from TFG group, except the three programs, were significantly lower than reliability of programs from the CG group and even less than the least reliable program from the CG group.

It is interesting to mention that before the experiment started, there has been completed power analysis “to get a closer look at the quality of a statistical hypothesis test”. (Muller & Hagner 2002) Even the analysis shows the poor power with a t-distribution of 0.645 (64.5% chances to find a difference between the groups), what is caused by small data point (ncg= 9 and ntfg=10 subjects) and only experiments with power more than 0.8 have a real chance to reveal any effect, experiment continued because it was not possible to collect more subjects for the experiment (Muller & Hagner 2002).

The following table summarize project design and results:

Design	**Description**
Number of developers	19 (graduate students)
Total time	Between July 2001 and August 2001 mostly during semesters breaks
Pair programming	No
Development method	TDD and non-TDD
Programming language	Java
TDD Project	"Graph Base", implementing of the main class of a given graph library containing only method declarations and method comments but not the method bodies
Test verification suite	20 tests and 522 assertions
Metrics	**Result**
Problem solving time	Small difference between both groups. Programming effort increases slightly when switching from traditional to test-first.
Time spent in Implementation phase (IP)	TFG spend less time for IP, even is not more efficient than CG related to overall working time.
Reliability after Implementation phase (IP)	Significantly lower. Except three programs all programs from the TFG are less reliable than worst one in the CG and two are even worse than 20% .
Reliability after Acceptance phase (AP)	Only three programs are less reliable than the median of 91% of the CG.. Five programs of the TFG achieved reliability over 96% compared to only one of the CG.
The variance of data distribution	Large variance 2.38 in the TFG compared to CG variance 1.0 make dangers interpret that TFG is better reliable.
Code reuse (number of reused methods)	No remarkable difference
Code reuse (number of failed method calls)	No remarkable difference
Code reuse (number of method calls that failed more than once)	Significantly less errors by reusing method more than once. Summary conclusion was: test first does not aid developer in a proper usage of exiting code but it guides him through ongoing testing in the process of fixing the fault.
Branch coverage	TFG had slightly better reliability in the final implementation.

Final conclusion was (Muller & Hagner 2002):

- If developer switches from traditional development to test-first approach it does not means that he can find the solution more quickly,
- Test-first delivers a slightly better reliability. This result is blurred by large variance of data points and bad results after first phase, Implementation Phase (IP). After IP test-first delivered significantly less reliable code,
- Test-first programmers reuse existing methods correctly and faster. This is caused by ongoing testing strategy and while fixing the fault developer learns about existing code,

- Five programs from the TFG achieved reliability over 96% and only one from the CG group.

Threats to validity of the study were identified as (Muller and Hagner 2002):

- Internal
  - The independent variable was not controlled technically and subjects in the TFG have been told in the requirements to use test-first approach. This question has been asked several times during experiment
- External
  - Differences in the skills between professional software engineers and students what can produce too optimistic and also too pessimistic results,
  - The XP education of participants occurred a short time ago,
  - Work conditions different than in the experiment can affect effectiveness of test-first approach.

VTT Technical Research Center of Finland

Maria Siniaalto and Pekka Abrahamsson (2007) accomplished controlled case study at "VTT Technical Research Center of Finland" by using nine senior undergraduate students. The case study was based on three projects for real customers that are accomplished in the period of nine weeks. Only one project was TDD project. Developers were divided in the group that uses the TDD approach and control group that uses traditional test last approach.

Following table summarize project design and results (Siniaalto & Abrahamsson, 2007):

Design	**Description**
Number of developers	9
Total time	9 weeks
Pair programming	Not known
Development method	Agile development method: Mobile-D
Programming language	Java
TDD Project 1 (LOC 7700), 4 developers	Research Data Management (WEB application;), Test-Last
TDD Project 2 (LOC 7000), 5 developers	Stock market browser (Mobile application) , Test-Last
TDD Project 3 (LOC 5800), 4 developers	PM tool (WEB application), TDD
Metrics	**Result**
WMC, DIT, NOC, RFC	No significant difference
CBO	No conclusion can be made
LCOM	Experience in TDD usage is required
Test coverage	Significant increase
Productivity	No effect

The final conclusion is summarized in following statements (Siniaalto & Abrahamsson, 2007):

- Claims that TDD leads to more loosely coupled objects cannot be confirmed,
- TDD does not automatically result in highly cohesive code,

- TDD leads to significantly increased test coverage.

The authors of this study made a list of threats that can jeopardize project results validity (Siniaalto & Abrahamsson, 2007):

- Programming experience,
- Level of project complexity,
- Project size,
- Distribution of the work,
- Use of students as study subjects,
- Non-TDD developers were encouraged to write tests, but were not informed about test coverage measurement.

Research Projects Accomplished By Using Professionals and Industrial Teams

Department of Computer Science, North Carolina State University

Boby George and Laurie Williams accomplished empirical project and comparative case study where TDD has been compared to traditional software development method. Empirical study of TDD has been completed by researchers in Germany at three companies: John Deere, Role Model Software, and Ericsson (George & Williams, 2003).

The purpose of this research project was to evaluate of the following two hypotheses:

- The TDD practice will yield superior external code quality when compared with code developed with a more traditional waterfall-like practice External code quality will be assessed based on the number of functional (black-box test cases) passed.
- Programmers who practice TDD will develop code faster than developers who develop code with a more traditional waterfall-like practice. Programmers speed will be measured by the time to complete (in hours) a specified program.

In this case three TDD experiment trials were executed with professional programmers which had experience level with TDD from beginner to expert. Developers were divided in the group of six pair developers that use the TDD approach and control group of six pair that use traditional test last approach. Effectiveness was measured by the time taken to develop the application. Quality of code was measured by results of a black-box functional test and by test code coverage analysis.

After the first trial has been found which delivered the code missing implementation of error handling. Only one control group pair wrote some worthwhile automated test case. Following table summarize project design and results (George & Williams, 2003):

Design	Description
Number of developers	24 (eight persons groups of developer at three companies)
Total time	
Pair programming	Yes
Development method	
Programming language	Java
TDD project (LOC 200), 24 developers divided in two groups of six pair developers	A Bowling game application adapted from an XP episode from Robert C. Martin book "Agile principles, patterns, and practices in C#"
Metrics	**Result**
TDD external code quality	Approximately 18 % more test passed
TDD productivity	Approximately 16 % more time to develop the application
Correlating Productivity and Quality	TDD pairs produced higher code quality. Two tailed Pearson Correlation had a value of 0.6661which was significant at the 0.019 level. Analysis indicates that higher quality is consequence of using more time and not solely due to the TDD practice itself.
Test coverage	Above industry standard that is between 80% and 90%. TDD covered 98% method, 82% statement and 97% branch coverage.

Throughout this research a survey analysis was completed and collected answers to three questions (George & Williams, 2003):

1. How productive is the practice for programmers?
2. How effective is the practice?
3. How difficult is the approach to adopt?

Following is a short summary of answers:

- *87.5 % developers believed "that TDD approach facilitates better requirements understanding*
- *95.8% believes that TDD reduces debugging effort*
- *78% of developers thought that TDD improves overall productivity of the programmer,*
- *92% of developers believed that TDD yields higher quality code*
- *79% believes that TDD promotes simpler design*
- *71% thought the approach was noticeably effective*
- *56% of the professional developers thought that getting into the TDD mindset was difficult*
- *23% indicated that the lack of upfront design phase in TDD was a hindrance*
- *average of the responses, 40% of the developers thought that the approach faces difficulty in adoption*

Finally authors concluded that (George & Williams, 2003):

- TDD approach seems to yield superior external code quality (18% more functional tests passed black-box test cases,
- TDD development took 16% more time for development,
- 80% of developers held that TDD was an effective approach and 78% believed that the approach improved programmers' productivity,
- TDD facilitates simpler design and lack of up-front design is not a hindrance,
- For some transition to TDD mindset is difficult.

Threats to validity of the study were identified as:

- Relatively small sample size (6 TDD and 6 control pairs),
- Results apply to the TDD with pair programming. The TDD does not specifically require pair programming,
- Application size was very small, and typical size of code was approximately 200 LOC,
- Developers had varying experience (from novice to experts). The thirds set of professional developers had only three weeks of TDD experience.

IBM Corporation and Department of Computer Science, North Carolina State University

This research project examines the efficacy of the TDD and primary focus was on reducing defects in the software intensive systems. Besides defect density reduction this project also commented on (Williams et al., 2003):

- TDD practice in the context of more robust design,
- Smoother code integration.

For this experiment have been used IBM developers that have developed device driver and seven previous releases of device driver software have been compared with new release developed by using the TDD approach.

Developers were IBM full-time employees, software engineers with minimum of bachelor's degree in computer science, electrical or computer engineering. The legacy team consisted of five engineers with significant experience in Java and C++ and knows very well application domain. The new release team consisted of nine full time engineers and no one of them knew TDD beforehand and three were unfamiliar with Java language (Williams et al., 2003).

The following table summarizes project setup and results:

Design	**Description**
Number of developers	14 (5 legacy (experienced) and 9 new release (TDD; some inexperienced))
Total time	119 man-months
Pair programming	No
Development method	
Programming language	Legacy (Java, C++), New release TDD (Java)
TDD Project (KLOC New, Base Total)	Device drivers on new platform Code size (Legacy KLOC 6.6, 49.9, 56.5) (New release KLOC 64.6, 9.0, 73.6)
Compared to project	56.6 KLOC, 45 man-months, 5 developers
Metrics	**Result**
TDD external code quality	Approximately 40 % lower defect density.
TDD productivity	Minimal affected
Test suite (34 KLOC)	2390 automated unite test cases, over 100 performance JUnit tests, 400 interactive tests
Test coverage	64.6 KLOC code has been covered by 34 KLOC of JUnit code (approximately 0.523 ratio per KLOC))

Final conclusion was that (Williams et al., 2003):

- TDD approach reduced defect density for approximately 40 %,
- Up-front test cases development drives a good requirement understanding,
- TDD delivers testable code,
- TDD creates a significant suite of regression test cases that are reusable and extendable asset that continuously improves quality over software lifetime.

Sanchez et al. 2007 made a post hoc analysis of this project and the same IBM team that practiced the TDD in five years. Besides a follow up analysis defect density, focus in this paper is testing effort and code complexity.

The analysis shows that testing effort average ratio across all ten releases is 0.61. This ratio has been calculated as TEST LOC/Source LOC.

Complexity of software was measured by cyclomatic complexity metrics. Analysis show that complexity increased slightly and suggests that use of TDD may reduce the increase in complexity as software ages.

The IBM did not strictly followed Beck's recommendation "Never write a single line of code unless you have a failing automated test". The IBM team worked from a design document, in the form of UML diagrams" The three developers wrote test before the implementation code, seven developers write the tests as they wrote code and one developer writes test after finishing a half of code implementation.

Microsoft Research and Center for Software Excellence, One Microsoft Way Redmond

Microsoft Corporation performed two case studies of using TDD n the two Microsoft divisions (Bhat & Nagappan, 2006), Windows division called A and MSN division called B. To make possible compute differences, both divisions agreed to use a project that has the same or very similar level of complexity. Developers in both projects did not know during development that their work was going to be assessed.

The main focus in this project was to:

- Compare differences in software quality between the TDD and non-TDD methodology.
- Overall development time in case of implementing the TDD and non-TDD methodology.

Quality of software was measured in terms of defects density. Case A that has been performed in the Windows division developed networking common library as a re-usable set of modules for different projects within the networking tools team. This library provides seventeen different classes that provide common functionality used in networking tools (Bhat & Nagappan, 2006).

Case B, performed in the MSN division worked with web services application

The following table summarizes project setup and results for Case A:

Design	Description
Number of developers	6 (High experienced)
Total time	24 man-months
Pair programming	No
Development method	
Programming language	C/C++
TDD project (6 KLOC)	Networking common library, 6 KLOC
Compared to project	4.5 KLOC, 12 man-months, 2 developers
Metrics	**Result**
TDD external code quality	2.6 times improvement
TDD productivity	35% increased development time
Test suite (KLOC)	4
Test coverage (Test LOC / Source LOC)	0,66
% Block coverage (unit-tests)	79%

Following table summarizes project setup and results for Case B:

Design	Description
Number of developers	5 - 8 (Medium experienced)
Total time	46 man-months
Pair programming	No
Development method	
Programming language	C++/C#
Project 1 (26 KLOC)	Web services application common library, 26 KLOC
Compared to project	149 KLOC, 144 man-months, 12 developers
Metrics	**Result**
TDD external code quality	4.2 times improvement
TDD productivity	15% increased development time
Test suite (KLOC)	23.2
Test coverage (Test LOC / Source LOC)	0,89
% Block coverage (unit-tests)	88%

Final conclusion in this paper was that both code quality and development time were increased. Further this paper promised to return of investment analysis to determine cost-benefits tradeoff between the increase in development time and the resulting improvement in software quality. (Bhat & Nagappan, 2006).

Threats to validity of the study were identified as:

- Higher motivation of developers that were using TDD methodology.
- The project developed by using TDD might be easier.
- Empirical study needs to be repeated in different environment and in different context before generalizing results.

IBM Corporation and Microsoft Corporation and North Carolina State University

This study is a product of a case study that involves three development teams at Microsoft and one development team at IBM. The contents of this study is follow up on the two previously reviewed studies in this paper in the "IBM Corporation and Department of Computer Science, North Carolina State University" and "Microsoft Research and Center for Software Excellence, One Microsoft Way Redmond" sections.

While the "Microsoft Research and Center for Software Excellence, One Microsoft Way Redmond" presented two different project in the Windows and MSN divisions, this study presented also third project completed at Microsoft Corporation in the Development Division. Following table contains summaries about third project setup and results completed in the Microsoft Development division (Nagappan et al., 2008):

Design	**Description**
Number of developers	7 (High experienced)
Total time	20 man-months
Pair programming	No
Development method	
Programming language	C# (Visual Studio)
Project (155.2 KLOC)	Application software, 155,2 KLOC
Compared to project	222 KLOC, 42 man-months, 5 developers
Metrics	**Result**
TDD productivity	20 % - 25 % increased development time
Test suite (KLOC)	60.3
Test coverage (Test LOC / Source LOC)	0.39
% Block coverage (unit-tests)	62%

Final conclusion in this study was (Nagappan et al., 2008):

- Reducing of defect density (IBM 40%, Microsoft 60% - 90%)
- Increase of time taken to code feature (15% - 35%).

Threats to validity of the study were identified as (Nagappan et al., 2008):

- Higher motivation of developers that were using TDD methodology.
- The project developed by using TDD might be easier.

- Conclusions from empirical studies in software engineering are difficult because any type of process depends to a large degree on a potentially large number of relevant context variables.
- Researchers become more confident in a theory when similar findings emerge in different context.

Microsoft is using a so called hybrid version of the TDD approach. This means that these projects had detailed requirements documents specification that drove test and development effort and there were also design meetings and review sessions. The legacy team did not use any agile methodologies nor the TDD practice (Nagappan et al., 2008).

Research Projects and Experiments Design and Results Analysis

Empirical research projects presented in the previous sections represent typical project design and organization. Developers were divided in the two groups where one group was control group that used traditional approach and other group that have used the TDD approach. The next what we got from the most of the papers is a number of developers included in each group and name, size of the project, and very bread definition or the research purpose. For example while George and Williams (2003) define two hypotheses and indicators of how these hypotheses will be measured, Muller and Hagner (2002) define three different focuses of their paper, but later made power analysis and calculated probability that will lead to non-rejection of the null hypothesis and cause Type II error. A Type II error, known as a false negative, occurs when the null hypothesis is not rejected although this hypothesis is false. Power analysis shows that there is only 64.5% chances to reveal any effect because it was not possible to collect more subjects for the experiment. To have a real chance, to reveal any effect by experiment the power analysis shall be more than 80% (Muller & Hagner, 2002).

While Muller and Hagner (2002) described the most of project design details, most of other papers did not provide sufficient information about project design. Even in Muller and Hagner traditional approach is described as design, implementation and test, details about the process that has been defined for groups (the TFG group and the CG group) are missing. In case of this project there were already created empty methods where developers need to insert call to graph library methods. Question is what design in this case means when the project already contains methods with empty bodies?

Other papers provided much less information and it is not clear on how the whole process was executed. For example, we do not know how much of design each group did. How much common introduction has been presented to both groups? What kind of work each group do? It would be interesting to know if both groups received a project description on piece of paper or in some kind of presentation and then each group is responsible for defining the development process, analysis, design and test and coding.

Time to understand requirement cannot be avoided and some kind of design shall exist in advance. Many people are talking about agile development method that just starts coding. This is wrong and agile also need to understand requirement or requirements before starts to produce tests, as well as traditional approach that we assume is analysis, design, coding and testing do. Differences are in the amount of time used for each of activities that precede start of coding. In case of agile approach, this time is shortening, as much as possible and coding can starts as soon as only a part of requirement is known. In case of predesigned projects this is not very probably and most probably is that requirements are known in advance. But we cannot see that from a project description in the presented papers

In case of the project accomplished by IBM and Microsoft, the TDD projects are compared to projects that are first of all different size and that had used different number of developers. For example in case of Microsoft projects and case B, Web services application common library (Bhat & Nagappan, 2006), the TDD project that had 26 KLOC and used 5 – 8 developers and last 45 man-months has been compared to a project developed by traditional approach that had 149KLOC and 12 developers and last 144 man-months. If we look at other projects that Microsoft presented (Bhat & Nagappan, 2006), we can quickly notice that code coverage has been reduced by project size. For example the Application software project that has,155,2 KLOC had 62% test coverage, while previously mentioned project, case B, Web services application common library (Bhat & Nagappan, 2006), had 88% of test coverage.

While the IBM study compared the TDD implementation by using the same project releases developed by using the TDD approach and non-TDD approaches, Microsoft compared the TDD projects with different projects that are developed by using non-TDD approach. In this case compared project complexity can differ significantly as well as number of participants and participants' skills. For example Microsoft project in the Windows division which used the TDD approach was approximately 6 KLOC large and involved 6 highly skilled developers and compared project that used non-TDD approach had 4.5 KLOC involved 2 developers. We are not informed about skills level of these two developers, but we are informed that the TDD project development time was 24 man-months and non-TDD project development time was 12 man-months. While the TDD project delivered about 25% of source code more than non-TDD project, number of developers in the TDD project was 3 times higher and it takes double the time to be completed. These simple analyses can raise a lot of questions and put doubt in study results. If we simply divide development time by number of developers, in this case 24 man-months by 6 developers, then we can find that the TDD project was completed in 4 months. If we do the same in case of compared non-TDD project and divide 12 months by 2 developers we will got 6 months. Where conclusion that the TDD increased development time by 25% - 35% came from? This conclusion is not documented properly and published studies just references that this is a Management estimate.

Becket and Putnam (2010) show that "*size of the project team directly influences number of defects that a project creates*". If number of developers is double as much (for example from 16 is increased to 32), the number of defects will be increased by 35.3%. If we know this, how we can compare projects that are different in size, use a different amount of the developers and believe that results are reliable? One of these examples is already mentioned: In Case B, "Web services application common library" (Bhat & Nagappan, 2006).

Next, very important, and what can give a quite different picture, is defect severity. From the studies and papers we cannot see how many of the each severity categories defects were discovered by the TDD approach and how many were discovered by traditional development approach. One example of the different defects severity category is for example, Severity 1 and 2, where are blocking errors that cannot be tolerated and these kinds of errors usually makes application to crash and correction needs changes in the application code. Severity 3 can be normal, what means that there is a workaround, or system can after workaround execute desired functionality. Severity 4 or 5 can be cosmetic errors that do not affect application software proper functioning. Number of errors discovered in each of these categories is important for making final conclusion. While only one defect that has Severity 1 can block the whole application, hundreds of cosmetic errors can be easily ignored. But small errors might not be underestimated and corrections of small errors, according to some statistics, can be 40 times more error prone than writing the new development code (Humphrey, 1989).

The use of students can be acceptable in well-designed research projects for initial researches and collecting initial experience and understanding of the research issues. But there are always questions how reliable on the reliability of these kinds of resources as well as final results.

The TDD projects accomplished by Microsoft and IBM teams were not respected strictly this methodology recommendations. In case of Microsoft development teams, requirement specification documents were written and design discussed on design review sessions (Nagappan et al., 2008).

Beacause of IBM development team's design, the IBM team worked from a design document, in the form of UML diagrams. The three developers wrote the test before the implementation code, seven developers wrote the tests as they wrote code and one developer wrote the test after finishing a half of code implementation (Sanchez et al., 2007). This also means that even high professionals are involved in the process of software development and even the TDD process described in this case was clear and simple to follow, following process guide lines cannot be assumed, and there is a need for continued reviewing, monitoring, and education. These examples illustrate very well issues related to the research projects and experiment design. These so called hybrids approaches as well as using of different metrics can make project results' comparison very difficult and even impossible.

What about a research project that used the Graph Library (Muller & Hagner, 2002) and required from participants to insert code in the already created methods with empty method bodies?

However, one can always argue that the project purpose is to test a Test-First approach, but the TDD is more than the Test-First approach. The TDD describes set of rules that drives implementation code design. Many suggested changing the name of Test Driven Development approach to the Test Driven Design approach! If design is known in advance, then, from the TDD point, a very important characteristic vanished.

When we read the results of the research (Nagappan et al., 2008) about defect density reduction at IBM for 40% and at Microsoft from 60% to 90% in case when the TDD approach is used, our first comment was that this can be possible only in case that in these two companies something was already very wrong regarding to Quality Assurance procedures.

The first principle of TDD, test-first approach, obviously favors test coverage, even it is not clear from the test results if the number of tests is higher in case of TDD approach or test coverage of existing code is better. But we can, at least conclude that TDD approach, if strictly followed, by default makes more tests because the TDD approach favor small and simple tests that are just covering a requirement as much as necessary, not less not more. The test is in the case of the TDD approach driving force and there is already recommendation to change the name of the Test Driven Development to the Test Driven Design.

Test coverage is important, but the question is how much test coverage is important? To find the answer we need to look at statistics and that what we already know. Empirical studies of real projects found that increasing code coverage above 70% - 80% is time consuming, leads to a relatively slow bag detection rate and is not cost effective (Cornett, 2011). This of course depends on the project type and "The ANSI/IEEE Standard for Software Unit Testing section 3.1.2 specifies 100% statement coverage as a completeness requirement. Section A9 recommends 100% branch coverage for code that is critical or has inadequate requirements specification" (Cornett, 2011) "Inadequate requirements specification" can fit well to the TDD approach. In the case of the TDD approach, development start does not wait that requirements are properly specified. Development starts as soon as only a part of requirement is known. This is a mantra of agile software development method, but any of the empirical projects did not report 100% of the test coverage?

There is also reported different test coverage during different types of test, and it can make sense to plan for example 90% of test coverage during unit testing and 80% of test coverage during integration testing and 70% of test coverage during system testing. While low test coverage can mean that test was not properly executed, high test coverage guarantees nothing (Cornett, 2011).

Defect severity is very important and from test results we cannot see how many defects were blocking, normal or cosmetic, if we just divide defects severity in these few simple categories. Differences in defect severity are very important and weight of one blocking error can be more important that any number of defects in cosmetic defects category.

Unit tests are not known for finding blocking errors. The most of these kinds of errors are found by integration and system testing. Such errors are usually a combination of the several method calls in a particular order and internal and external data modification, as well as caused by application interfaces. Internal in this case means global and local variables defined inside of the code and external data are data that are stored outside of the application space.

Conclusion that the TDD delivers more reliable code is blurred by results of the experiment at Karlsruhe University (Muller & Hagner, 2002) where reliability in the first experiment phase, also called "Implementation Phase", was significantly worse and except the three programs from the test-first group, all others programs reliability was worse than reliability of the worse program created by the traditional approach. Final results in this project after "Acceptance Phase" and errors correction, confirmed that the TDD approach delivered slightly more reliable code than a code developed by traditional approach to software development. Final conclusion was that the test first approach does not aid developer in a proper usage of exiting code but it guides him through ongoing testing in the process of fixing the fault (Muller & Hagner, 2002).

We are wondering what the result can mean if the waterfall method has additional time for development? The research projects and experiments are designed from the TDD point of view. The question is how reliable can be comparison of the project that needs more time and is using different methodology if this project shall be completed in shorter time that fits very well to different method requirements?

In case of this kind of experiment can be interesting to switch developers between these two groups to find out how much developer experience affects research results. This means that developers from the test-first group shall use traditional approach and developers from the control group shall use test-first approach. How many programs from each group will in this case achieve high test coverage? Such experiment can confirm if software quality depends of the developer skills and experience or that is a method that improves code quality. Strong indication that skills are driving force behind quality is a fact that one program from CG group that used traditional approach delivered superior performances related to code coverage (Muller & Hagner, 2002) A better project design can have too equal groups of developers and let them to develop the same project. Then these two groups shall switch development method and develop other project. Both results shall be compared

Other experiments accomplished at IBM Corporation and Microsoft Corporation and at North Caroline University supported conclusion that the TDD approach delivers more reliable code.

We can believe that by limiting amount of implementation code to as much as necessary, to satisfy requirement, or better to say to proof that code can deliver the expected result, the TDD approach limits the number of defects, because each new line of source code can introduce additional defect. Defect can occur as conflict with existing code and different workflows or data combination. Minimizing amount of source code at least makes easier to understand possible conflicts and makes modification safer. However, minimizing source code can affect application robustness and reliability in case when error handling is not properly implemented. The result from the research experiment George and Williams 2003 shows that the TDD does not encourage implementation of proper error handling. Never mind how much sense some believes can do, these need to be confirmed by experiment. Very often experiment results surprise.

Important advantage of TDD is that a PROCESS drives the writing of test cases that at least has one cover implementation code. The traditional approach does not ensure that implementation code is covered by any test case. Even traditional approach is never writing whole application and testing it afterwards, but rather is writing a piece of code and then testing it, it is interesting why the TDD approach reduces defect density. What is wrong or missing in case of the traditional software development?

One of the possible answers is that the traditional approach very often does these tests manually and do not use automatic test tools. This means that many tests that are executed manually and result verified visually, are not repeated in whole and in the same order when new code is written or old code changed. The major difference is that the TDD approach requires creating of test suite that can be executed by automatic test tools, such as, a Unit test tool which enables regression testing accomplished as quickly as possible and repeats exactly the same steps in the same order as it was executed previously. The test suite is continuously updated and enlarged and documents implementation. Repeating test in the same order can actually hide a defect, but in case of the complex testing of interfaces, web services and remote calls, this can be necessary.

While in case of the TDD approach, test coverage and regression test execution are most important part of the development process and deliverables, in case of traditional approach these are depending of the internal development team culture, skills and experience, and requires additional time and resources that needs to be spend on supervision and control. This means that the TDD process focus is on the testing requirement, but not on requirement implementation.

There are significant differences in the results when some process steps are replacing places and are executed in the different order. In case of the traditional approach, implemented code is output from the first step, next step, create test case, and uses this code as an input. The implemented code more or less affects test design. Of course in case of the black-box testing this could not be important, but in case of Unit testing when developer who made a code is writing a test, this difference is important. There are few important differences in cases when process specifies that first step is writing of implementation code:

- Requirement is implemented as soon as developer believe that understand what is required,
- Test is created according to implementation code, what can be different than required by specification
- Requirement testability is not tested properly,
- Implementation code is not optimized according to requirement, and very often is modified copy/paste code from previously similar solution.

When process describes that first step is a test implementation and test is an input to next step, implement code step then:

- Creating test needs proper requirement understanding,
- Requirement misunderstanding is discovered as early as possible
- Test becomes driving force for implementation code design.

There is also another very important difference: while in the case of the traditional development approach, it is necessary for code review to find out if there is a test of particular class or method available, in case of the TDD approach test is created by default.

However test development needs additional development time and this cannot be avoided in case of the TDD approach. Results of the studies accomplished by IBM and Microsoft shows that the TDD uses approximately 15% - 35% more time for development (Bhat & Nagappan, 2006; George & Williams, 2003; Nagappan et al., 2008).

It is also important to remember that number of tests is not directly proportional to test quality. There can be a huge number of tests that are testing the same over and over, and wasting time and resources.

Very often used comment about the TDD approach is missing of the up-front design. In case of simple applications where requirements and requirements conflicts can be easily understandable, the TDD should not be a problem from the design point of view. In case of complex applications, the up-front design looks preferable. The reason for this conclusion is that the TDD by default introduces a Bottom-Up development approach and in this case in later project phases can be necessary to implement large code refactoring to satisfy additional requirements or optimize the solution. In the case of the TDD approach if refactoring is properly and continuously implemented, the code should be highly structured and easier to reuse. However, a highly structured code can affect application performance and make more difficult to follow, as well as to predict the result of changes. Highly structured code is also more difficult to understand and debug. This can be risky in the case of high code reusability.

Always keep in mind that each methodology interest is to be used and improved, based on the collective experience and best practice. This means that especially methodologies that pretend on using attribute "new" will claim that such an approach solves many "old" methodology issues.

The authors' own experience is that "new" is very often improvement of an "old" methodology usage, but "new" also introducing new issues and new complexities. However, a traditional methodology approach or the TDD methodology approach, both need to know about requirements in advance. The differences are in the time used for specifying requirements and also in the requirement size and complexity. TDD requires less time and starts implementation as soon as some parts of requirements are known. Requirement understanding is validated during development and at the end of iteration.

This Paper's Contribution

The "Research Projects and Experiments Design and Results Analysis" section contains the critical and very detailed analysis of the projects design and published results. The following is a short summary of this paper contribution:

- Critical review of the TDD empirical projects design,
- Critical analysis of empirical projects results,
- Critical analysis of test coverage myth.
- Proposition how to improve evaluation results of TDD approach

While there are a lot of papers where are analyzed or presented results of the empirical projects, they are missing critical analysis of empirical project design and organization and especially an analysis of how much of the TDD approach has been used?

We analyzed Microsoft hybrid approach, where design has been presented before process start. Developers at IBM admitted that the TDD test first approach was not followed consistently and test cases were developed after the implementation code or in parallel with implementation. The IBM developer team also used up-front design presented in the UML diagrams. What about a project where are already made empty methods and developers need to insert code in these methods? How reliable these results can be if we know this in advance and know that the whole project takes 200 LOC?

When different practices are mixed, it can be difficult to separate how much each of practice affects final results.

Conclusion

This paper analyzed results of published research projects and experiments where the primary purpose was to receive confirmation about the TDD claimed benefits and advantages. The paper also focused on analysis on the reliability of the results and reliability of the empirical projects design and participants.

It is difficult to draw a conclusion that the TDD methodology claims are proven in general, because results differ significantly. It is not surprise that TDD is not yet widely used in the industrial teams because existing evidence is not sufficient and conclusions and results can be quite contradictory. The following reasons why the projects and their corresponding results are not easy to compare may be identified as:

- Using of different design methodologies,
- Using of different metrics,
- Using of developers that had varying experience,
- Empirical studies are based on projects in various environments (i.e. various levels of CMMI),
- Analyzed projects were of different size and goal,
- Project design often used hybrid approach that is different than the TDD recommendations.

A large sample of analyzed projects in previous survey articles contributed to the fact that drawn conclusions are more general, but lead to the fact that not many conclusions are generally valid.

What we can identify is consistent in most of the research projects and experiments of that the TDD approach provides better code coverage. Better code coverage is obviously caused by the TDD rule that test shall be written first and the rule that development stops when code makes all tests executed successfully.

The claim that the TDD approach is using the same amount or less time for project development cannot be confirmed and according to research papers this approach uses approximately more time for development

The claim that TDD improves internal software design and makes further changes and maintenance easier cannot be confirmed. It seems that the design primarily depends on the developer's skills and experience, as well as the implementation of best practice and internal standards.

Thus, neither hypothesis "TDD is superior over traditional approach" nor vice versa cannot be considered proven. Therefore, some specific class of projects and development environments exists in which TDD is superior over traditional approach. However, it is not easy to identify this class without additional analysis and research. In addition, the question "what about TDD effectiveness in the uncertain development conditions?" should be more carefully studied since majority of software industry is working at level 1 of CMMI with the need that processes must be very flexible and must be adapted to varying (and often chaotic) market conditions and must work with limited resources. The authors' believe that this is a challenge worth to devote their future work.

References

Beck, K. (2002). *Test driven development by example.* Addison-Wesley.

Becket, D. M., & Putnam, D. T. (2010). Predicting software quality. *Journal of Software Technology, 13*(1).

Bhat, T., & Nagappan, N. (2006). Evaluating the efficacy of test driven development: industrial case studies. Center for Software Excellence, Redmond, WA and Microsoft Research, Redmond, WA, *ISESE '06 Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering.*

Boddoohi, M. (2010). *An evaluation of software architecture – Using aspects.* North Caroline State University.

Bulajic, A., & Stojic, R. (2011). Analysis of the test driven development by example. *Scientia Journal "Res Computeria", 2*(6I) 01 - 09.

Causevic, A., Sundmark, D., & Punnekkat, S. (2011). *Factors limiting industrial adoption of test driven development: A systematic review.* Målardalen University, School of Innovation, Design and Engineering, Västerås, Sweden.

Chidamber, S. F., & Kemerer, C. F., (1994). A metrics suit for object oriented design. *IEEE Transactions in Software Engineering, 20*(6).

Cornett, S. (2011). *Minimum acceptable code coverage.* Bullseye testing Technology, 2006-2011.

Fowler, M. (1999). *Refactoring improving the design of existing code.* Addison-Wesley.

George, B., & Williams, L. (2003). *An initial investigation of test driven development in industry.* Department of Computer Science, North Carolina State University. Available at http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf

Humphrey, W.S. (1989). *Managing the software process*. Addison Wesley.

Kollanus, S. (2011). *Critical issues on test-driven development*. Faculty of Information Technology, University of Jyvåskyla, Finland, Springer-Verlag Berlin Heidelberg.

Nagappan, N., Maximilien, E. M., Bhat, T. & Williams, L. (2008). *Realizing quality improvement through test driven development: results and experiences of four industrial teams.* Three Development Teams at Microsoft and One Development Team at IBM

Newkirk, J. W., & Vorontsov, A. A. (2004). *Test-driven development in Microsoft .NET*. Microsoft Press.

Metrics _1.3.6 (2012). *Metrics 1.3.6*. sourceforge.net. Available at http://metrics.sourceforge.net/

Muller, M. M., & Hagner, O. (2002). *Experiment about test-first programming*. Computer Science Department University Karlsruhe.

Opdyke, W. F (1992). "PhD thesis", University of Illinois at Urbana-Champaign available at Internet ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z.

Roussev, B. (2003). Teaching introduction to programming as part of the IS component of the business curriculum. *Journal of Information Technology Education, 2*, 349-356. Retrieved May 12, 2003 from http://www.jite.org/documents/Vol2/v2p349-356-43.pdf

Sanchez, J. C., Williams, L. & Maximilien, E. M. (2007). *On the sustained use of a test-driven development practice at IBM.* Agile 2007 Conference

Siniaalto, M., & Abrahamsson, P. (2007). *A comparative case study on the impact of test-driven development on program design and test coverage*. VTT Technical Research Center of Finland. Available at http://www.computer.org/portal/web/csdl/doi/10.1109/ESEM.2007.35

Williams, L., Maximilien, E. M., & Vouk, M. (2003). *Test-driven development as a defect-reduction practice.* IBM Corporation and North Carolina State University, 14th International Symposium on Software Reliability Engineering. Available at http://www.computer.org/portal/web/csdl/doi/10.1109/ISSRE.2003.1251029

Biographies

Aleksandar Bulajic is working for IBM Denmark as a consultant and full time employee. He is working more than twenty five years as a computer expert and consultant for some of the top companies in the world.

Aleksandar Bulajic is PhD Candidate at Faculty of Information Technology, Metropolitan University. He graduated from the University of Liverpool, with a Master's in Science degree (Cum Laude) in Information Technology, and from the Economic University with a Bachelor's (BA) degree.

His papers and articles were presented and published at several international IT conferences in USA, Australia, Canada and Serbia, and presented at Liverpool University and published in professional journals and on IBM Intranet. He wrote and published several novels and a stage play, and is working on screenplay manuscripts. Besides his professional work and writings, his current interests and writings are mostly related to film and theatre and interdisciplinary multimedia experiments. He is a member of Liverpool University Alumni Community and Informing Science Institute. E-mail: **LANB@45.dk** **aleksandar.bulajic.1145@fit.edu.rs**

Dr. Samuel Sambasivam is Chairman and Professor of the Computer Science Department at Azusa Pacific University. His research interests include optimization methods, expert systems, client/server applications, database systems, and genetic algorithms. He served as a Distinguished Visiting Professor of Computer Science at the United States Air Force Academy in Colorado Springs, Colorado for a year. He has conducted extensive research, written for publications, and delivered presentations in Computer Science, data structures, and Mathematics. He is a voting member of the ACM and is a member of the Institute of Electrical and Electronics Engineers (IEEE).

Radoslav Stojic has a thirty years experience on software development, testing and certification in European aeronautic industry.

He has been working on research and development projects ranging from FBW flight control to walking robots and flight simulators.

He is currently teaching software quality and testing, artificial intelligence and software for computer games at Faculty of Information Technology at Belgrade.