

Dietmar Winkler  
Stefan Biffl  
Johannes Bergsmann (Eds.)

LNBIP 166

# Software Quality

**Model-Based Approaches for Advanced  
Software and Systems Engineering**

6th International Conference, SWQD 2014  
Vienna, Austria, January 2014  
Proceedings



EXPERIENCE THE VALUE OF QUALITY

 Springer

Lecture Notes  
in Business Information Processing

166

Series Editors

Wil van der Aalst

*Eindhoven Technical University, The Netherlands*

John Mylopoulos

*University of Trento, Italy*

Michael Rosemann

*Queensland University of Technology, Brisbane, Qld, Australia*

Michael J. Shaw

*University of Illinois, Urbana-Champaign, IL, USA*

Clemens Szyperski

*Microsoft Research, Redmond, WA, USA*

Dietmar Winkler  
Stefan Biffi  
Johannes Bergsmann (Eds.)

# Software Quality

Model-Based Approaches for Advanced  
Software and Systems Engineering

6th International Conference, SWQD 2014  
Vienna, Austria, January 14-16, 2014  
Proceedings



Springer

## Volume Editors

Dietmar Winkler  
Vienna University of Technology  
Institute of Software Technology  
and Interactive Systems  
Vienna, Austria  
E-mail: dietmar.winkler@tuwien.ac.at

Stefan Biff  
Vienna University of Technology  
Institute of Software Technology  
and Interactive Systems  
Vienna, Austria  
E-mail: stefan.biff@tuwien.ac.at

Johannes Bergsmann  
Software Quality Lab GmbH  
Linz, Austria  
E-mail: johannes.bergsmann@software-quality-lab.at

ISSN 1865-1348  
ISBN 978-3-319-03601-4  
DOI 10.1007/978-3-319-03602-1  
Springer Cham Heidelberg New York Dordrecht London

e-ISSN 1865-1356  
e-ISBN 978-3-319-03602-1

Library of Congress Control Number: 2013956013

© Springer International Publishing Switzerland 2014

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

*Typesetting:* Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

# Message from the General Chair

The Software Quality Days (SWQD) conference and tools fair started in 2009 and has grown to one of the biggest conferences on software quality in Europe with a strong community. The program of the SWQD conference is designed to encompass a stimulating mixture of practical presentations and new research topics in scientific presentations as well as tutorials and an exhibition area for tool vendors and other organizations in the area of software quality.

This professional symposium and conference offers a range of comprehensive and valuable opportunities for advanced professional training, new ideas, and networking with a series of keynote speeches, professional lectures, exhibits, and tutorials.

The SWQD conference is suitable for anyone with an interest in software quality, such as test managers, software testers, software process and quality managers, product managers, project managers, software architects, software designers, user interface designers, software developers, IT managers, development managers, application managers, and those in similar roles.

January 2014

Johannes Bergsmann

# Message from the Scientific Program Chair

The 6th Software Quality Days (SWQD) conference and tools fair brought together researchers and practitioners from business, industry, and academia working on quality assurance and quality management for software engineering and information technology. The SWQD conference is one of the largest software quality conferences in Europe.

Over the past years, a growing number of scientific contributions were submitted to the SWQD symposium. Starting in 2012 the SWQD symposium included a dedicated scientific track published in scientific proceedings. For the third year we received an overall number of 24 high-quality submissions from researchers across Europe, which were each peer-reviewed by three or more reviewers. Out of these submissions, the editors selected four contributions as full papers, an acceptance rate of 17%. Further, ten short papers, which represent promising research directions, were accepted to spark discussions between researchers and practitioners at the conference.

The main topics from academia and industry focused on systems and software quality management methods, improvements of software development methods and processes, latest trends in software quality, and testing and software quality assurance.

This book is structured according to the sessions of the scientific track following the guiding conference topic “Model-Based Approaches for Advanced Software and Systems Engineering”:

- Software Process Improvement and Measurement
- Requirements Management
- Value-Based Software Engineering
- Software and Systems Testing
- Automation-Supported Testing
- Quality Assurance and Collaboration

January 2014

Stefan Biffl

# Organization

SWQD 2014 was organized by the Software Quality Lab GmbH and the Vienna University of Technology, Institute of Software Technology and Interactive Systems, and the Christian Doppler Laboratory “Software Engineering Integration for Flexible Automation Systems.”

## Organizing Committee

### General Chair

Johannes Bergsmann                      Software Quality Lab GmbH

### Scientific Program Chair

Stefan Biffl                                  Vienna University of Technology

### Proceedings Chair

Dietmar Winkler                          Vienna University of Technology

### Organizing and Publicity Chair

Petra Bergsmann                          Software Quality Lab GmbH

## Program Committee

SWQD 2014 established an international committee of well-known experts in software quality and process improvement to peer-review the scientific submissions.

Maria Teresa Baldassarre	University of Bari, Italy
Armin Beer	University of Applied Sciences, Vienna, Austria
Miklos Biro	Software Competence Center Hagenberg, Austria
Ruth Breu	University of Innsbruck, Austria
Manfred Broy	Technische Universität München, Germany
Maya Daneva	University of Twente, The Netherlands
Oscar Dieste	Univesidad Politecnica de Madrid, Spain
Fajar Ekaputra	Vienna University of Technology, Austria
Frank Elberzhager	Fraunhofer IESE, Germany
Michael Felderer	University of Innsbruck, Austria

Gordon Fraser	University of Sheffield, UK
Christian Frühwirth	Aalto University, Finland
Marcela Genero,	University of Castilla-La Mancha, Spain
Jens Heidrich	Fraunhofer IESE, Germany
Frank Houdek	Daimler AG, Germany
Slinger Jansen	Utrecht University, The Netherlands
Petri Kettunen	University of Helsinki, Finland
Mahvish Khurum	Blekinge Institute of Technology, Sweden
Olga Kovalenko	Vienna University of Technology, Austria
Eda Marchetti	ISTI-CNR Pisa, Italy
Juergen Münch	University of Helsinki, Finland
Simona Nica	Graz University of Technology, Austria
Markku Oivo	University of Oulu, Finland
Oscar Pastor Lopez	Technical University of Valencia, Spain
Mauro Pezzè	University of Lugano, Switzerland
Dietmar Pfahl	University of Tartu, Estonia
Rick Rabiser	Johannes Kepler University, Austria
Rudolf Ramler	Software Competence Center Hagenberg GmbH, Austria
Andreas Rausch	Technische Universität Clausthal, Germany
Barbara Russo	Free University of Bolzano, Italy
Klaus Schmid	University of Hildesheim, Germany
Estefania Serral	Vienna University of Technology, Austria
Stefan Wagner	University of Stuttgart, Germany
Dietmar Winkler	Vienna University of Technology, Austria

## Additional Reviewers

Asim Abdulkhaleq	Marcel Ibe
Alarico Campetelli	Jan-Peter Ostberg
Peter Engel	Jasmin Ramadani
Daniel Mendez Fernandez	Joachim Schramm
Benedikt Hauptmann	Fabian Sobiech



# Table of Contents

## Keynote

Software Quality Assurance by Static Program Analysis . . . . .	1
<i>Reinhard Wilhelm</i>	

## Software Process Improvement and Measurement

An Industry Ready Defect Causal Analysis Approach Exploring Bayesian Networks . . . . .	12
<i>Marcos Kalinowski, Emilia Mendes, and Guilherme Horta Travassos</i>	

Business Intelligence in Software Quality Monitoring: Experiences and Lessons Learnt from an Industrial Case Study (Short Paper) . . . . .	34
<i>Alexander Kalchauer, Sandra Lang, Bernhard Peischl, and Vanessa Rodela Torrents</i>	

Dealing with Technical Debt in Agile Development Projects (Short Paper) . . . . .	48
<i>Harry M. Sneed</i>	

## Requirements Management

Statistical Analysis of Requirements Prioritization for Transition to Web Technologies: A Case Study in an Electric Power Organization . . . .	63
<i>Panagiota Chatzipetrou, Christos Karapiperis, Chrysa Palampouiki, and Lefteris Angelis</i>	

Challenges and Solutions in Global Requirements Engineering – A Literature Survey (Short Paper) . . . . .	85
<i>Klaus Schmid</i>	

Automated Feature Identification in Web Applications (Short Paper) . . .	100
<i>Sarunas Marciuska, Cigdem Gencel, and Pekka Abrahamsson</i>	

## Value-Based Software Engineering

Value-Based Migration of Legacy Data Structures . . . . .	115
<i>Matthias Book, Simon Grapenthin, and Volker Gruhn</i>	

## Software and Systems Testing

An Integrated Analysis and Testing Methodology to Support Model-Based Quality Assurance .....	135
<i>Frank Elberhager, Alla Rosbach, and Thomas Bauer</i>	
Effects of Test-Driven Development: A Comparative Analysis of Empirical Studies (Short Paper) .....	155
<i>Simo Mäkinen and Jürgen Münch</i>	
Isolated Testing of Software Components in Distributed Software Systems (Short Paper) .....	170
<i>François Thillen, Richard Mordinyi, and Stefan Biffel</i>	

## Automation-Supported Testing

Automated Test Generation for Java Generics (Short Paper) .....	185
<i>Gordon Fraser and Andrea Arcuri</i>	
Constraint-Based Automated Generation of Test Data (Short Paper) ...	199
<i>Hans-Martin Adorf and Martin Varendorff</i>	

## Quality Assurance and Collaboration

RUP Alignment and Coverage Analysis of CMMI ML2 Process Areas for the Context of Software Projects Execution (Short Paper) .....	214
<i>Paula Monteiro, Ricardo J. Machado, Rick Kazman, Cláudia Simões, and Pedro Ribeiro</i>	
Directing High-Performing Software Teams: Proposal of a Capability-Based Assessment Instrument Approach (Short Paper) .....	229
<i>Petri Kettunen</i>	
<b>Author Index</b> .....	245

# Software Quality Assurance by Static Program Analysis

Reinhard Wilhelm

Fachrichtung Informatik  
Universität des Saarlandes  
Saarbrücken, Germany

wilhelm@cs.uni-saarland.de

<http://rw4.cs.uni-saarland.de/people/wilhelm.shtml>

**Abstract.** Static program analysis is a viable, sound and automatic technique to prove correctness properties about programs, both functional properties as well as non-functional properties. It is one of the techniques, highly recommended for high criticality levels by several international software-quality standards for the domains of transportation, healthcare, factory automation, and electric/electronic systems. The precision of static analysis increases the more information is made available to it. This additional information can be given by programmer annotations, or it can be transferred from the model level in model-based software design. We give an introduction to static program analysis as a verification technology, describe several applications to the development of safety-critical systems, and show how it can be integrated into a model-based design flow.

**Keywords:** static program analysis, functional verification, non-functional properties, model-based design.

## 1 Introduction

Static program analysis is a viable, sound and automatic technique to prove correctness properties about programs, both functional properties as well as non-functional properties. Its origins lie in the area of compiler construction where it is used to prove the applicability of efficiency-improving program transformations. Such transformation must not change the semantics of programs. To prove that they preserve the semantics often requires a global static analysis of the program.

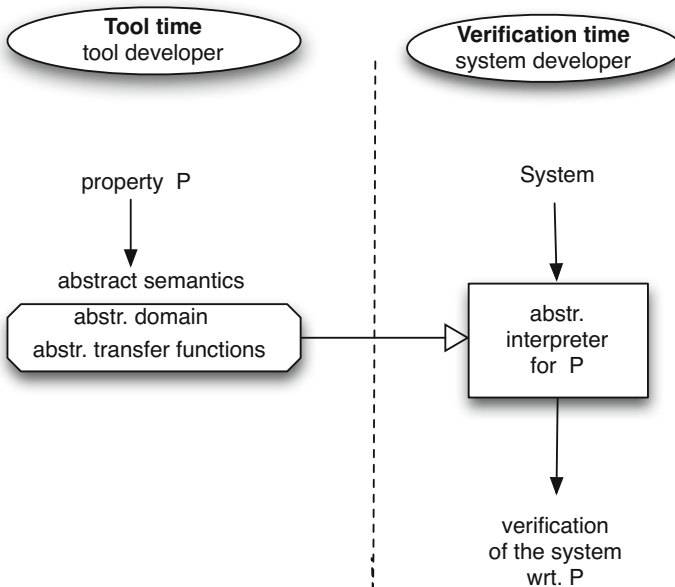
Many of the considered properties are *undecidable*. So, there is no hope that an automatic method could be sound and complete at the same time. *Soundness* means that the method never gives an incorrect answer. *Completeness* means that it will detect all properties that actually hold. No compromise is made on the side of *correctness*: a statement derived by static analysis will always hold. This is in contrast to bug-chasing tools, which are often also called *static analyzers*. In (sound) static-analysis tools, the compromise is made on the side

of *completeness*: the static analysis may fail to derive all correctness statements that actually hold and thus issue warnings that are, in fact, *false alarms*. Bug-chasing tools, in general, are both unsound and incomplete; they produce false alarms and they fail to detect all bugs.

The *precision* of static analysis, i.e., the set of valid correctness statements proved by the analysis, increases with more information made available to it. This additional information can be given by programmer annotations, or it can be transferred from the model level in model-based software design.

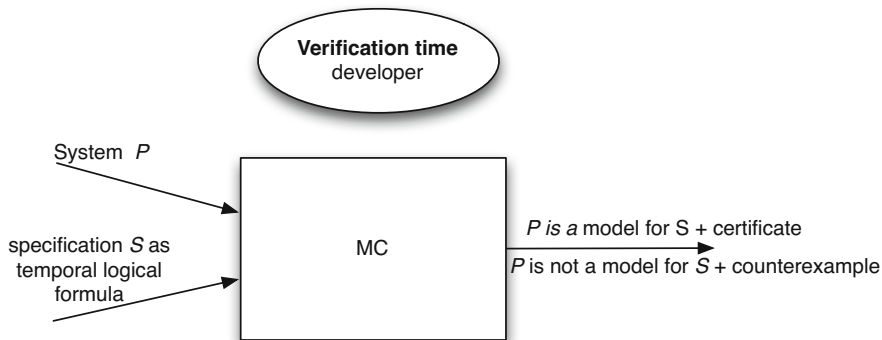
Static analysis has emancipated itself from its origins as a compiler technology and has become an important *verification method* [12]. Static analyses prove *safety properties* of programs, such as the absence of run-time errors [3,15], and they prove partial correctness of programs [14]. They determine *execution-time bounds* for embedded real-time systems [5,19] and check synchronization properties of concurrent programs [20]. Static analysis has become indispensable for the development of reliable software. It is one of several candidate formal methods for the analysis and verification of software.

The acceptance of these formal methods in industry has been slow. This is partly caused by the competences required from the users of the methods.



**Fig. 1.** Work distribution in static program analysis

Static program analysis has the best distribution of required competences of all formal methods, cf. Fig. 1. It splits the necessary competences between *tool developers* and *system developers*. The tool developers, highly qualified specialists in the underlying theory, design and implement a static analysis tool aimed



**Fig. 2.** Work distribution in model checking

at proving one or a set of correctness properties. The systems developer applies this tool. He does not need to know the intricacies of the underlying theory.

Other formal methods require competences not available with the systems developer. *Deductive methods* are not automatic, and therefore require interaction by highly skilled users. The verification of a C compiler [13], one of the landmarks in software verification, required strong research qualification. *Model checking* requires the user to specify the correctness properties in some temporal logic and to supply an abstract model of the system, both non-trivial tasks, cf. Fig. 2.

### 1.1 A Short History of Static Program Analysis

Compilers for programming languages are expected to translate source-language programs correctly into efficient machine-language programs. Even the first FORTRAN compiler transformed programs to efficiently exploit the very modest machine resources of that time. This was the birth date of “optimizing compilers”. In FORTRAN these transformations mainly concerned the efficient access to arrays, a data structure of high relevance to the numerical analyst.

Even the first FORTRAN compiler mentioned above implemented several efficiency-improving program transformations, called *optimizing transformations*. They should, however, be carefully applied. Otherwise, they would change the semantics of the program. Most such transformations have *applicability conditions*, which when satisfied guarantee the preservation of the semantics. These conditions, in general, depend on non-local properties of the program, which have to be determined by a *static analysis* of the program performed by the compiler [17].

This led to the development of *data flow analysis*. This name was probably to express that it determines the flow of properties of program variables through programs. The underlying theory was only developed in the 70s when the semantics of programming languages had been put on a solid mathematical basis. Two doctoral dissertations had the greatest impact on this field. They were written

by Gary A. Killdall (1972) [11] and by Patrick Cousot (1978) [4]. Gary Killdall clarified the lattice-theoretic foundations of data-flow analysis. Patrick Cousot established the relation between the semantics of a programming language and static analyses of programs written in this language. He therefore called such a semantics-based program analysis *abstract interpretation*. This relation to the language semantics allowed for a correctness proof of static analyses and even the design of analyses that were correct by construction.

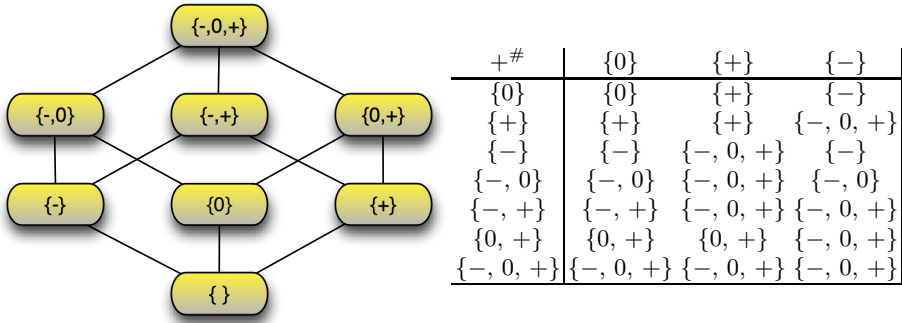
## 1.2 The Essence

The task of the designer of a semantics-based static program analysis is to find an abstraction of the programming-language semantics that is strong enough to prove the envisioned correctness property. Let us consider a very simple example to show the essential ingredients. The correctness properties we want to prove are that the program never divides by 0 and never takes the square root of a negative number. Sufficient for this proof would be that

- for every occurrence of an expression `sqrt(e)` in the program we know that the value of  $e$  is always nonnegative when execution reaches this occurrence, and
- for every occurrence of an expression `a/e` in the program we know that the value of  $e$  is never 0 when execution reaches this occurrence.

The abstraction we use to derive such proofs is quite simple: The semantics of the programming language, for instance an interpreter for programs in the language, records the actual value of each program variable in a data structure, which we may call a *value environment*. Our static analysis, an abstraction of the semantics, records the signs of variables of numeric type, i.e., + for positive, – for negative, and 0 for 0, in a data structure, called a *sign environment*. Sometimes the analysis knows only that the value is non-negative and will associate the variable therefore with  $\{+, 0\}$ , read as "the value is positive or 0". The fact that the analysis doesn't know anything about the sign is expressed by associating  $\{+, 0, -\}$  with the variable.

We arrange the sets of signs in an algebraic structure, a *lattice*, see Fig. 3. Going up in the lattice, i.e., following one of the edges upwards, means losing information; the node in the lattice at which we arrive admits more signs than the node we started from. So, we lost precision. The lattice represents a *partial order*; elements lower in the lattice are less precise than elements above them in the lattice, i.e. connected with them by a path going up. The analysis may sometimes arrive at a program point from two or more different predecessor points with different information, i.e., lattice elements. This is the case at the end of conditionals and at the beginning of loops, where two different control flows merge. The information that the analysis can safely guarantee is represented by the least common ancestor in the lattice of the incoming information. Going up to the least common ancestor means applying the *least upper bound operator* of the lattice.



**Fig. 3.** The lattice for rules-of-signs and the table for the abstract addition. Missing columns can be added by symmetry.

The association of sets of signs with program variables is called *sign environment*, as said above. In order for the program analysis to determine (new) signs of variables it needs to execute statements in sign environments instead of in value environments. These abstract versions of the semantics of statements are called *abstract transfer functions*. For an assignment statement  $x = e$ ; its abstract transfer function looks up the sign information for the variables in  $e$ , attempts to determine the possible signs of  $e$ , and associates these signs with  $x$  in the sign environment. As part of that it must be able to evaluate expressions in sign environments to obtain signs of the value of the expression.

The rules for this evaluation we know from our school days;  $- \times -$  gives  $+$ ,  $- \times +$  gives  $-$  and so on. We can easily extend this to sets of signs by computing the resulting signs for all combinations and collecting all results in a set. The rules for addition are given in the table in Fig. 3. The table defines an abstract addition operator  $\times \#$ .

We have met now the essential ingredients of a static program analysis, an *abstract domain*, a lattice, and a set of *abstract transfer functions*, one for each statement type. Following this principle we could design more complex static analyses. A very helpful static analysis, called *interval analysis*, would compute at each program point an enclosing interval for all possible values a numerical variable may take on. The results of this analysis allows one to exclude index-out-of-bounds errors for arrays; any expression used as an index into an array whose interval of possible values is completely contained in the corresponding index range of the array will never cause such an error. A run-time check for this error can thus be eliminated to gain efficiency.

Fig. 4 shows a simple program and an associated control-flow graph. We will demonstrate the working of the rules-of-sign analysis at this program. As a result we will obtain sets of possible signs for the values of program variables  $x$  and  $y$  at all program points.

The analysis goes iteratively through the program as shown in Fig. 5. It starts with initial assumptions at node 0 about the signs of variables, e.g. that these are unknown and could thus be any sign. It then evaluates the expressions it

```

1: x = 0;
2: y = 1;
3: while (y > 0) do
4:     y = y + x;
5:     x = x + (-1);

```

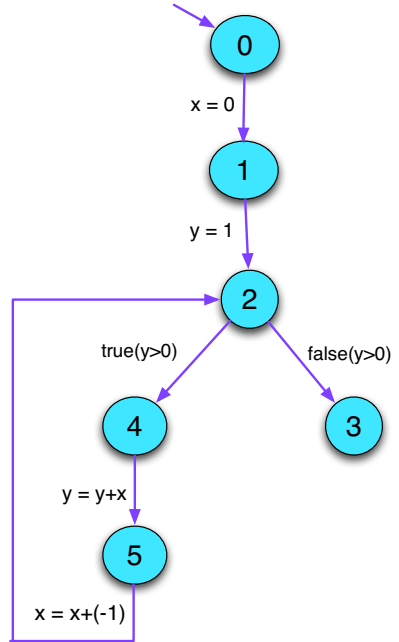


Fig. 4. A program and its control-flow graph

0		1		2		3		4		5	
$x$	$y$	$x$	$y$	$x$	$y$	$x$	$y$	$x$	$y$	$x$	$y$
$\{-,0,+\}$	$\{-,0,+\}$	$\{0\}$	$\{-,0,+\}$	$\{0\}$	$\{+\}$	$\{\}$	$\{\}$	$\{0\}$	$\{+\}$	$\{0\}$	$\{+\}$
$\{-,0,+\}$	$\{-,0,+\}$	$\{0\}$	$\{-,0,+\}$	$\{-,0\}$	$\{+\}$	$\{\}$	$\{\}$	$\{-,0\}$	$\{+\}$	$\{-,0\}$	$\{-,0,+\}$
$\{-,0,+\}$	$\{-,0,+\}$	$\{0\}$	$\{-,0,+\}$	$\{-,0\}$	$\{+\}$	$\{-,0\}$	$\{-,0,+\}$	$\{-,0\}$	$\{+\}$	$\{-,0\}$	$\{-,0,+\}$
$\{-,0,+\}$	$\{-,0,+\}$	$\{0\}$	$\{-,0,+\}$	$\{-,0\}$	$\{+\}$	$\{-,0\}$	$\{-,0,+\}$	$\{-,0\}$	$\{+\}$	$\{-,0\}$	$\{-,0,+\}$

Fig. 5. (Cleverly) iterating through the program. Column numbers denote program points, rows describe iterations.

finds in the actual sign environment, executes the assignments by associating the left-hand side variable with the set of signs computed for the right side. It stops when no set of signs for any variable changes any more. This can be seen in the last two rows of the table in Fig. 5.

You may be surprised to see that the set of signs  $\{-,0,+\}$  is associated with  $y$  at node 3. This results from the fact that our, not really precisely described analysis does not exploit conditions. It just checks whether the transitions along them are possible for a given a sign-environment. A different analysis would exploit the condition `false(y > 0)` and compute the set  $\{-,0\}$  for  $y$  at node 3.

By some nice mathematical properties, e.g. monotonicity of the abstract semantics and finite height of the lattice, this process is guaranteed to terminate.



One could be tempted to say that everything is fine, and on the correctness side, it actually is. If the analysis evaluates  $e$  to  $\{+, 0\}$  we are assured that executing `sqrt(e)` does never take the square root of a negative value. And if the analysis associates  $e$  with  $\{+, -\}$  we know that  $a/e$  never will attempt to divide by 0. On the precision side, the world may look different: A set of signs  $\{+, 0, -\}$  computed for an expression  $e$  in  $a/e$  will force the analysis to issue a warning "Possible division by 0", where this actually might be impossible.

## 2 Applications

Static program analysis is one of the verification techniques required by the transition from *process-based assurance* to *product-based assurance*. It is strongly recommended for high criticality levels by several international standards such as ISO 26262, DO178C, CENELEC EN-50128, and IEC 61508, see Fig. 6 and [8,9].

**Table 9 — Methods for the verification of software unit design and implementation**

Methods		ASIL			
		A	B	C	D
1a	Walk-through <sup>a</sup>	++	+	o	o
1b	Inspection <sup>a</sup>	+	++	++	++
1c	Semi-formal verification	+	+	++	++
1d	Formal verification	o	o	+	+
1e	Control flow analysis <sup>bc</sup>	+	+	++	++
1f	Data flow analysis <sup>bc</sup>	+	+	++	++
1g	Static code analysis	+	++	++	++
1h	Semantic code analysis <sup>d</sup>	+	+	+	+

<sup>a</sup> In the case of model-based software development the software unit specification design and implementation can be verified at the model level.

<sup>b</sup> Methods 1e and 1f can be applied at the source code level. These methods are applicable both to manual code development and to model-based development.

<sup>c</sup> Methods 1e and 1f can be part of methods 1d, 1g or 1h.

<sup>d</sup> Method 1h is used for mathematical analysis of source code by use of an abstract representation of possible values for the variables. For this it is not necessary to translate and execute the source code.

**Fig. 6.** ISO 26262 highly recommends static code analysis for ASIL level B -D

### 2.1 Proving the Absence of Run-Time Errors

Run-time errors will typically lead to systems crash. They should be avoided under all circumstances. In disciplined programming languages, the compiler needs to insert run-time checks to catch them and treat them properly if that is at all possible.

Several systems exist that attempt to prove the absence of run-time errors. ASTRÉE [3] tackles many of these run-time errors. It was designed to produce only very few false alarms for control/command programs written in C according to ISO/IEC 9899:1999 (E) (C99 standard). This is achieved by providing

many abstract domains for particular program features such as feedback control loops, digital filters and clocks. The ASTRÉE user is provided with options and directives to reduce the number of false alarms significantly. Besides the usual run-time errors such as index-out-of-bounds, division by 0, dereference of null pointers, it has strong analyses for the rounding errors of floating-point computations. These are particularly important for embedded control systems. The designers of these systems in general assume that the programs work with reals and base their stability considerations on this assumption. In fact, rounding errors may lead to divergence of a control loop with potentially disastrous consequences.

The POLYSPACE VERIFIER[15] aims at similar errors. However, ASTRÉE and the POLYSPACE VERIFIER have different philosophies with regard to the continuation of the analysis after an identified run-time error. VERIFIER’s philosophy is “grey follows red”, meaning that code following an identified run-time error is considered as non-reachable and will be disregarded by the analysis. ASTRÉE continues the analysis with whatever information it may safely assume. VERIFIER’s philosophy would make sense if the programmer would not remove the found run-time error. In practice, he will remove it. The code previously found to be unreachable thereby becomes reachable—assuming that the modification removed the error—and will be subject to analysis with possibly newly discovered errors and warnings. So, VERIFIER may need more iterations than ASTRÉE until all errors and warnings are found and removed. Given that analysis time is in the order of hours or even days for industrial size programs this increased number of iterations may be annoying.

POLYSPACE VERIFIER is able to check for the compatibility with several coding standards, such as MISRA C.

## 2.2 Determining Execution-Time Bounds for Real-Time Systems

The determination of reliable and precise execution-time bounds for real-time systems was a long open problem. Its difficulty came from the fact that modern high-performance processors achieve much of their performance from architectural components such as caches, deep pipelines, and branch speculation. These components introduce high variability of instruction execution-times. The execution times of instructions strongly depend on the execution state of the architecture, which results from the execution history. Static program analysis proved to be instrumental in solving this problem [5]. Dedicated analyses were able to derive strong invariants about all execution states at a program’s points [7]. These are used to bound the instructions’ execution times in a safe way.

aiT [1] of AbsInt is the leading tool for this task. It has been admitted for certification of several time-critical avionics systems by the European Aviation Safety Agency (EASA) and has been used in the certification of the Airbus A380, A350, and M400 airplanes [18]. Although the variability of execution times of machine instructions is high due to the influence of caches and pipelines, aiT was able to produce rather tight bounds, i.e., show small over-estimation [18].

Timing analysis is strongly dependent on the execution platform. So, the technology has to be instantiated for each newly employed execution platform by deriving an abstract model of this execution platform. This has been done for many different processor architectures that are used in embedded control systems.

### 2.3 Determining Upper Bounds on Space Usage

Exceeding the reserved stack space causes overwriting of other memory areas and will in most cases lead to unrecoverable crashes. It is therefore absolutely required to verify before run time that the reserved stack space is sufficient under all circumstances. Having reliable information about the maximal extension of stacks not only lets the developer sleep better; it has an economic side-effect. The space reserved for the stacks can be tightly bounded; no safety margin needs to be added. STACKANALYZER [2] computes upper bounds on the space required by all system and user stacks.

## 3 Integration into a Model-Based Design Flow

As stated above, additional information about the program to be analyzed may help the static analyzer to improve precision. This additional information can be given by the programmer, but it can also be transferred from the model level if the program is automatically generated by the code generator of some model-based design tool. One such example is information about ranges of input values. Sensor input often is constrained to ranges. These can be imported into a tool for the analysis of run-time errors and then exploited to improve precision. For real-life industrial projects, the number of automatically generated constraints for inputs and calibration variables can easily reach 1000, which shows how relevant automatic support is [10].

ASTRÉE is integrated with the TARGETLINK code generator, importing input ranges, output ranges and the top-level functions of the model. Traceability is ensured for ASTRÉE alarms for TARGETLINK and REAL TIME WORKSHOP EMBEDDED CODER; the tool guides the developer automatically from the code causing the alarm to the corresponding position in the model.

The PolySpace Verifier [15] is integrated with Simulink and the REAL TIME WORKSHOP EMBEDDED CODER in similar ways, e.g. in importing ranges for global variables from the model level.

Timing analysis for real-time systems need to know the maximal iteration counts for all loops in the program under analysis. These can be often determined by the value analysis, which is one phase in the timing analysis. However, they cannot be automatically determined for functions generated for look-up blocks. Fortunately, information about the maximal iteration counts is present in the data dictionary of the TARGETLINK code generator. Program annotations for loops can be automatically generated from this information [10].

The precision of execution-time bounds can often be improved by the elimination of infeasible paths since it could be that these paths contribute to the

determination of the bounds. [16] uses model-level information available in synchronous specifications to eliminate infeasible paths.

A smooth integration of the analysis and verification tools into the design flow is essential for the acceptance. One aspect is that analysis results should be exported back to the model level. Model-level objects are back-annotated with analysis information such as execution-time estimates. For traceability, the code generator needs to maintain the correspondence between the model-level input and the generated code. The integration of the model-based design tool SCADE with AbsInt's timing-analysis tool, aiT realizes such a coupling [6]. The developer immediately obtains an estimate of the time bound of the SCADE "symbols" he has entered into his specification.

## 4 Conclusion

Static program analysis is an automatic verification technique with a high potential for the development of correct safety-critical software. The method was explained on an intuitive level, and several applications and associated tools were presented, which are in routine use in industry, in particular in the embedded-systems industry. They have proved to be useful, usable, efficient, and precise enough. Several of the mentioned tools have been admitted to the certification of safety- and time-critical avionics subsystems. The last section has indicated how the integration with model-based design tools can improve precision and alleviate the user from manual and error-prone interaction with tools.

## References

1. AbsSint, <http://www.absint.com/ait>
2. AbsSint, <http://www.absint.com/stackanalyzer/index.htm>
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Riva, X.: A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI 2003, pp. 196–207. ACM, New York (2003)
4. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252. ACM, New York (1977)
5. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 469–485. Springer, Heidelberg (2001)
6. Ferdinand, C., Heckmann, R., Sergent, T.L., Lopes, D., Martin, B., Fornari, X., Martin, F.: Combining a high-level design tool for safety-critical systems with a tool for WCET analysis on executables. In: ERTS2 (2008)
7. Ferdinand, C., Wilhelm, R.: Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems* 17(2-3), 131–181 (1999)

8. Kästner, D., Ferdinand, C.: Efficient verification of non-functional safety properties by abstract interpretation: Timing, stack consumption, and absence of runtime errors. In: Proceedings of the 29th International System Safety Conference, ISSC 2011, Las Vegas (2011)
9. Kästner, D., Ferdinand, C.: Static verification of non-functional requirements in the ISO-26262. Embedded World Congress (2012)
10. Kästner, D., Kiffmeier, U., Fleischer, D., Nenova, S., Schlickling, M., Ferdinand, C.: Integrating model-based code generators with static program analyzers. In: Embedded World. Design & Elektronik (2013)
11. Kildall, G.A.: A unified approach to global program optimization. In: Fischer, P.C., Ullman, J.D. (eds.) POPL, pp. 194–206. ACM (1973)
12. Kreiker, J., Tarlecki, A., Vardi, M.Y., Wilhelm, R.: Modeling, Analysis, and Verification - the Formal Methods Manifesto 2010 (Dagstuhl Perspectives Workshop 10482). Dagstuhl Manifestos 1(1), 21–40 (2011)
13. Leroy, X.: Formally verifying a compiler: Why? how? how far? In: CGO. IEEE (2011)
14. Lev-Ami, T., Reps, T.W., Sagiv, S., Wilhelm, R.: Putting static analysis to work for verification: A case study. In: ISSTA, pp. 26–38 (2000)
15. PolySpace, <http://www.mathworks.de/products/polyspace/>
16. Raymond, P., Maiza, C., Parent-Vigouroux, C., Carrier, F.: Timing analysis enhancement for synchronous program. In: RNTS (2013)
17. Seidl, H., Wilhelm, R., Hack, S.: Compiler Design - Analysis and Transformation. Springer (2012)
18. Souyris, J., Pavec, E.L., Himbert, G., Jgu, V., Borios, G.: Computing the worst-case execution time of an avionics program by abstract interpretation. In: Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis, pp. 21–24 (2005)
19. Wilhelm, R.: Determining bounds on execution times. In: Zurawski, R. (ed.) Handbook on Embedded Systems, ch. 14. CRC Press (2006)
20. Yahav, E.: Verifying safety properties of concurrent java programs using 3-valued logic. In: Hankin, C., Schmidt, D. (eds.) POPL, pp. 27–40. ACM (2001)

# An Industry Ready Defect Causal Analysis Approach Exploring Bayesian Networks

Marcos Kalinowski<sup>1</sup>, Emilia Mendes<sup>2</sup>, and Guilherme Horta Travassos<sup>3</sup>

<sup>1</sup> UFJF – Federal University of Juiz de Fora,  
Rua José Kelmer, s/n - 36.036-330 - Juiz de Fora, Brazil

<sup>2</sup> BTH – Blekinge Institute of Technology,  
37179 Karlskrona, Sweden

<sup>3</sup> UFRJ – Federal University of Rio de Janeiro,  
68511 Rio de Janeiro, Brazil

kalinowski@ice.ufjf.br, emilia.mendes@bth.se, ght@cos.ufrj.br

**Abstract.** Defect causal analysis (DCA) has shown itself an efficient means to improve the quality of software processes and products. A DCA approach exploring Bayesian networks, called DPPI (Defect Prevention-Based Process Improvement), resulted from research following an experimental strategy. Its conceptual phase considered evidence-based guidelines acquired through systematic reviews and feedback from experts in the field. Afterwards, in order to move towards industry readiness the approach evolved based on results of an initial proof of concept and a set of primary studies. This paper describes the experimental strategy followed and provides an overview of the resulting DPPI approach. Moreover, it presents results from applying DPPI in industry in the context of a real software development lifecycle, which allowed further comprehension and insights into using the approach from an industrial perspective.

**Keywords:** Bayesian Networks, Defect Causal Analysis, Defect Prevention, Software Process Improvement, Software Quality, Experimental Software Engineering.

## 1 Introduction

Regardless of the notion of software quality that a project or organization adopts, most practitioners would agree that the presence of defects indicates lack of quality, as it had been said by Card [1]. Consequently, learning from defects is profoundly important to quality focused software practitioners.

Defect Causal Analysis (DCA) [2] allows learning from defects in order to improve software processes and products. It encompasses the identification of causes of defects, and ways to prevent them from recurring in the future. Many popular process improvement approaches (e.g., Six Sigma, CMMI, and Lean) incorporate causal analysis activities. In industrial settings, effective DCA has helped to reduce defect rates by over 50 percent in organizations. For instance, companies such as IBM [3], Computer Science Corporation [4], and InfoSys [5] reported such achievements.

Therefore, since software projects usually spend about 40 to 50 percent of their effort on avoidable rework [6], DCA has the potential of, besides improving quality, also improving productivity by reducing rework significantly and thus reducing the total project effort.

In order to produce evidence-based DCA guidelines to facilitate its efficient adoption by software organizations, a systematic literature review (SLR) was conducted in 2006 and replicated (new trials) in 2007, 2009, and 2010 [7]. Besides enabling to collect evidence, the SLR trials also allowed the identification of opportunities for further investigation [8]. For instance, “the DCA state of the art did not seem to include any approach integrating learning mechanisms regarding cause-effect relations into DCA meetings”. This means that, in the approaches found, the knowledge on cause-effect relationships gathered during each DCA meeting was only used to initiate actions addressing the identified causes and discarded afterwards.

This scenario encouraged us to initiate research in order to propose an innovative DCA approach, carefully considering the guidelines and this particular research opportunity to integrate cause-effect learning mechanisms into DCA meetings. An experimental strategy was adopted in order to mitigate risks relating to the uptake of our proposed approach by industry.

This paper details the steps initially employed to build and evaluate the resulting DCA approach, called DPPI (Defect Prevention-Based Process Improvement), and extends our research by providing new results from applying it in industry. Therefore, we describe (i) the underlying experimental strategy applied aiming at moving the approach from conception towards industry readiness, (ii) the resulting DPPI approach, and (iii) the new results obtained from applying DPPI in industry in a real software development project lifecycle, which allow further comprehension into using the approach from an industrial perspective.

The remainder of this paper is organized as follows. In Section 2, the underlying experimental strategy used to tailor DPPI is described. In Section 3, an overview of the resulting DPPI approach is provided. In Section 4, the use of DPPI in industry and the corresponding results are presented. Final considerations are given in Section 5.

## 2 The Experimental Strategy

The experimental strategy adopted considered an evidence-based methodology for defining new software technologies, combining secondary and primary studies [9]. The objective of the secondary studies (systematic reviews) was grounding the initial concept of the approach with evidence. The primary studies, on the other hand, were used to tailor the approach and to allow further understanding on its usage feasibility and on hypotheses associated to possible benefits of using it.

An overview of the strategy is shown in Figure 1. It is possible to see that the strategy included four major activities: (i) conducting systematic reviews to gather and update DCA evidence [7]; (ii) building the initial concept of the approach based on such evidence and addressing the innovation of integrating cause-effect learning mechanisms into DCA meetings [10]; (iii) undertaking a proof of concept to evaluate

the feasibility of using the approach [11]; and (iv) conducting experimental studies to evaluate possible benefits of using it [12]. A summary on how each of those activities was accomplished is provided in the subsections hereafter. The new results obtained from applying DPPI in industry in a real software development project lifecycle will be provided in Section 4.

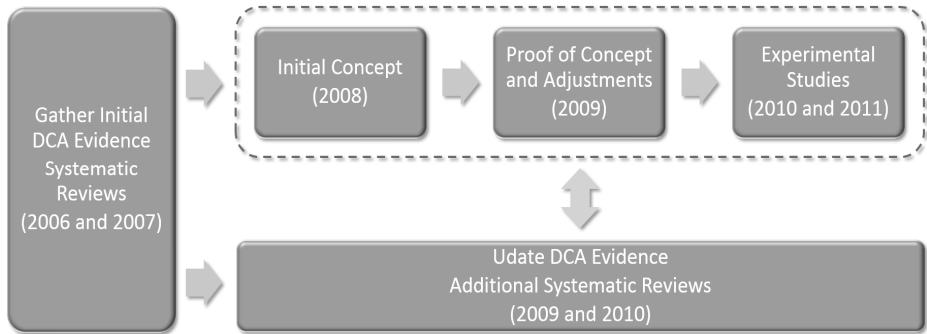


Fig. 1. Overview of the adopted experimental strategy

## 2.1 Systematic Reviews: Gathering DCA Evidence

The goal of our SLR was to conduct a broad review regarding the state of the art of DCA, more specifically aiming at: (i) summarizing the processes, approaches, and guidance described in the technical literature, and (ii) summarizing and analyzing the defect and cause classification schemes focused on those from sources concerned with DCA.

Based on this goal a research protocol was formulated following the PICO strategy [13] and reviewed by other researchers, as suggested by Kitchenham and Charters [14]. The details of the protocol can be found in the web extra of [7]. The protocol was planned so that its results could be updated by periodically conducting new SLR trials. Those trials were accomplished in August 2006, September 2007, January 2009 and July 2010. At all, 427 different papers were retrieved and 72 matched the protocol's inclusion criteria. As a result, information concerning the SLR goal could be grouped into summary tables and evidence-based DCA guidelines could be produced [7][8].

Moreover, as can be seen in Figure 1, the SLRs played a key role in building the approach, the evidence obtained from the first two trials (2006 and 2007) was used as the basis to propose the initial concept [10], while the remaining two (2009 and 2010) were used throughout the research period to allow further adjustments [11].

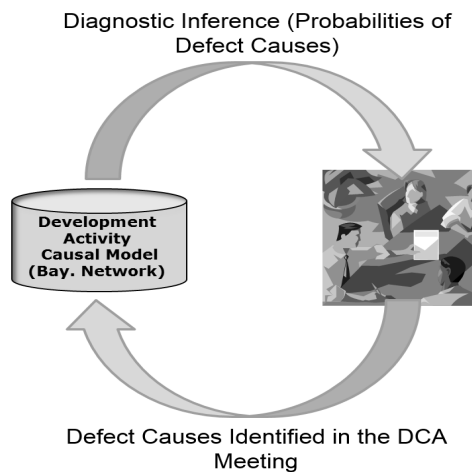
## 2.2 Initial Concept: Addressing the Identified Gap

The initial concept of the DCA approach was proposed after the first two SLRs (2006 and 2007) and is described in [10]. Its main innovation was addressing an opportunity for further investigation: the absence of learning mechanisms regarding cause-effect



relations identified in DCA meetings. Therefore, this initial concept suggested the use of Bayesian networks. Using such networks would allow the diagnostic inference to support defect cause identification in subsequent DCA meetings. The hypotheses were that this kind of inference could help answering questions during DCA meetings, such as: “Given the past projects within my organizational context, with which probability does a certain cause lead to a specific defect type?”. Additionally, in order to allow the usage of the Bayesian diagnostic inferences during DCA meetings, the traditional cause-effect diagram [15] was extended into a probabilistic cause effect diagram [10].

Figure 2 illustrates the approach’s proposed feedback and inference cycle, representing the cause-effect learning mechanism to support the identification of defect causes.



**Fig. 2.** The proposed feedback cycle. Building the Bayesian networks for each development activity upon results of the DCA meetings.

Later, after an additional SLR trial (conducted in 2009) and feedback gathered from experts in the field, this initial concept was evolved and tailored into the DPPI (Defect Prevention-Based Process Improvement) approach [11]. Besides using and feeding Bayesian networks to support DCA meetings with probabilistic cause-effect diagrams, DPPI also addresses the specific practices of the CMMI CAR (Causal Analysis and Resolution) process area, described in [16].

Once proposed there was a need to evaluate the feasibility of using the approach in the context of real development projects. Moreover, if the approach shows itself feasible there would be a need to evaluate the hypotheses related to benefits of using the approach’s main innovation: that the Bayesian inference could help in identifying causes in subsequent DCA meetings. The feasibility was addressed through a proof of concept (Section 2.3), while the hypotheses were evaluated through a set of experimental studies (Section 2.4).

### 2.3 Proof of Concept: Evaluating Feasibility

As a proof of concept, DPPI was applied retroactively to perform defect causal analysis on the functional specification activity of a real Web-based software project. The scope of this project was to develop a new information system to manage the activities of the COPPETEC Foundation. The system was modularized and developed following in an iterative and incremental lifecycle. Software inspections were performed on each of the modules' functional specifications [17], using a computational inspection framework [18]. In total, more than 10 iterations were performed and more than 200 use cases were specified, implemented and delivered to the customer over a three years development period. By the end of the project, all inspection data was available, including details on about 1000 defects found and removed from the functional specifications before the actual implementation.

In this context, the researchers applied DPPI retroactively to the functional specification activity of the fourth developed module. The details of performing each DPPI activity are described in [11]. However, since the application was retroactive, as a simple proof of concept, the identified causes could not be addressed in order to improve the development process (the product was already developed).

The experience of applying DPPI manually and retroactively to a real Web-based software project indicated its usage feasibility, minor adjustments to be done and insights into the required tool support [11]. Consequently, a prototype for providing such support was built [12]. Moreover, during this experience DPPI's Bayesian diagnostic inference predicted the main defect causes efficiently (according to the development team), motivating further investigation into the underlying hypotheses associated to benefits of the approach's innovation.

### 2.4 Experimental Studies: Evaluating the Innovation

Three hypotheses were stated in the experimental study plan to evaluate benefits (regarding effectiveness, effort and usability) of using probabilistic cause-effect diagrams obtained from automated Bayesian diagnostic inferences (DPPI's approach) to support the identification of causes of defects:

- **H1:** The use of DPPI's approach to identify causes of defects results in more effective cause identification, when compared to *ad-hoc* cause identification.
- **H2:** The use of DPPI's approach to identify causes of defects reduces the effort of cause identification, when compared to *ad-hoc* cause identification.
- **H3:** The use of DPPI's approach to identify causes of defects improves user satisfaction in identifying causes, when compared to *ad-hoc* cause identification.

The experimental study design had to handle several constraints, since it was planned to happen in the context of a real software project (the same used for the proof of concept) and its real development team. The decision to conduct the experimental study in a real context was to eliminate confounding factors and bias regarding defect causes that could easily happen in the context of toy problems with students.

To handle these constraints, the study design comprised a set of arrangements considering two treatments (applying DPPI and not applying it), two objects (defects from two different project modules) and four subjects, participating in three separate trials each. These trials allowed analyzing the hypotheses based on three different observation scenarios. More details on the experimental study design and its results can be found in [12].

The results indicated a significant improvement for the DPPI treatment regarding both the effectiveness in identifying the main defect causes (H1) and the cause identification effort (H2). In the last observation scenario, for instance, which involved comparing each participant applying both treatments on the same object, all participants were more effective (at least 40% higher) and spent less time (at least 20% less) when using the DPPI treatment. Regarding user satisfaction (H3), nothing could be observed based on the collected qualitative data.

In general, the study design and arrangements allowed providing consistent arguments with preliminary indications on concrete benefits associated to using DPPI's probabilistic cause-effect diagrams to improve the identification of causes in DCA meetings.

Having provided this overview of the adopted experimental strategy for building and evaluating the DPPI approach, the next section focuses on the resulting approach itself.

### **3 DPPI: Defect Prevention-Based Process Improvement**

DPPI represents a practical approach for defect prevention. As mentioned before, its main innovation is the integration of knowledge gathered in successive causal analysis events in order to provide a deeper understanding of the organization's defect cause-effect relations. Such integration allows establishing and maintaining common causal models (Bayesian networks) to support the identification of causes in each causal analysis event. Those causal models support diagnostic reasoning, helping to understand, for similar projects of the organization, which causes usually lead to which defect types.

Additionally, DPPI follows the evidence-based DCA guidelines [7] in order to tailor the defect prevention activities into specific tasks, providing further details on the techniques to accomplish these tasks efficiently. Moreover, it integrates defect prevention into the measurement and control strategy of the development activity for which defect prevention is being applied, allowing one to observe whether the implemented improvements to the development activity brought real benefits. This is done by defining defect-related metrics to be used to measure and control the development activity.

Given that DPPI aims at continuous improvement, it was designed to take place right after the inspection of each main development activity artifacts. Note that the inspection process expects the correction of defects by the author [19]. Thus, when the DPPI is launched the defects corresponding to the development activity have already been corrected.

DPPI includes four activities: (i) Development Activity Result Analysis; (ii) DCA Preparation; (iii) DCA Meeting; and (iv) Development Activity Improvement. The main tasks for each of these activities, as well as the roles involved in their execution, are depicted in Figure 3. Note that the software development activity itself and its inspection are out of DPPI's scope.

Figure 3 also shows the development activity’s causal model. DPPI considers those causal models to be dynamically established and maintained by feeding Bayesian networks. Pearl [20] suggests that probabilistic causal models can be built using Bayesian networks, if concrete examples of cause-effect relations can be gathered in order to feed the network. In the case of DPPI the examples to feed the Bayesian network can be directly taken from each DCA meeting results.

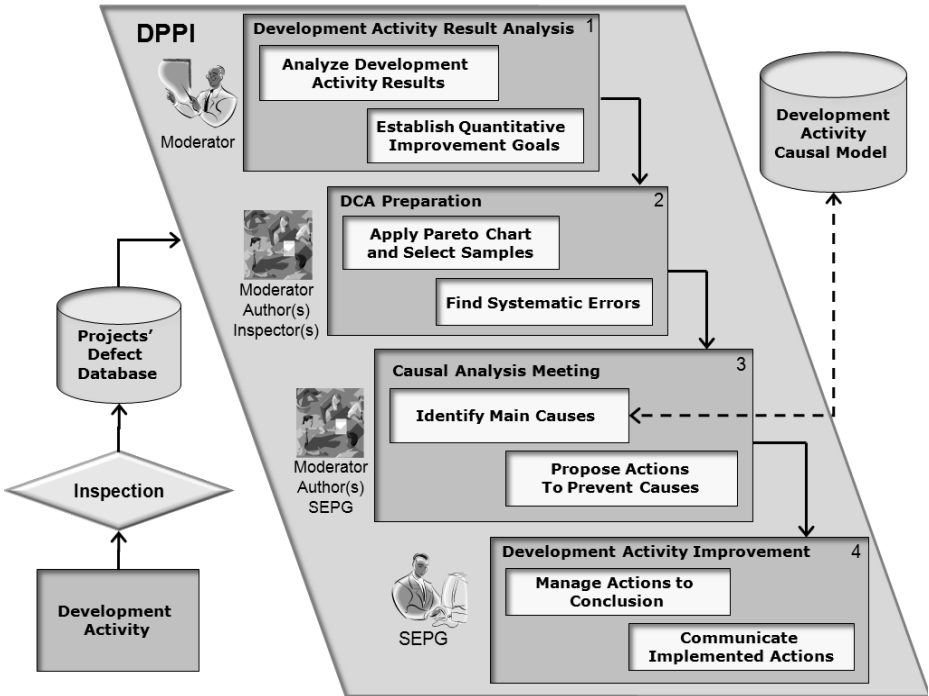


Fig. 3. DPPI approach overview, adapted from [12]

A brief description of the four DPPI activities and their tasks, with examples of how they can be applied considering real software project data taken from the proof of concept described in [11], is provided in the following subsections.

### 3.1 Development Activity Result Analysis

The Development Activity Result Analysis aims at quantitative measurement and control of the development activity. It comprises two tasks to be performed by the DCA moderator. Further details on these tasks follow.

**Analyze Development Activity Results.** This task aims at analyzing the development activity’s defect-related results by comparing them against historical defect-related results for the same development activity in similar projects (or previous iterations of the same project). Therefore, the number of defects per unit of size and per inspection

hour should be analyzed against historical data using a statistical process control chart. As suggested by [21], the type of the statistical process control chart for those metrics should be a U-chart, given that defects follow a Poisson distribution. Those charts can indicate if the defect metrics of the development activity are under control by applying basic statistical tests. An example U-chart based on real project data for defects per inspection hour is shown in Figure 4.

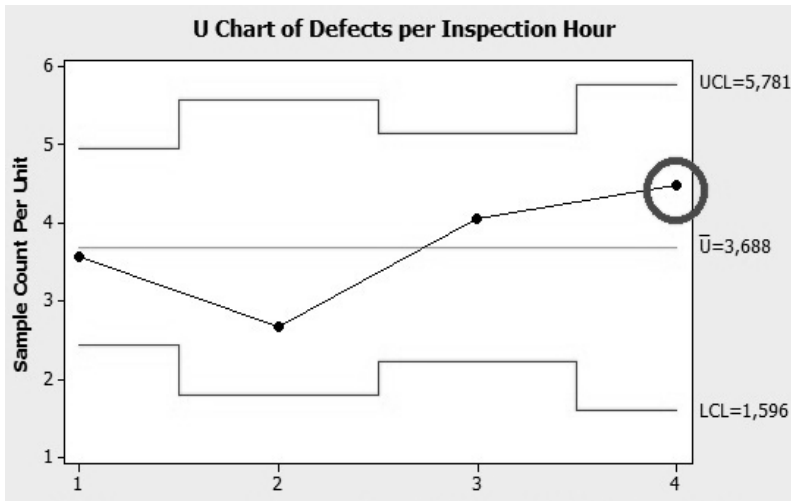


Fig. 4. Defects per inspection hour U-Chart [11]

**Establish Quantitative Improvement Goals.** This task involves establishing improvement goals for the development activity. A typical example of quantitative improvement goal is “reducing the defect rate per inspection hour (or per unit of size) by X percent”.

If the development activity is out of control, the focus of the causal analysis meeting becomes revealing assignable causes and the improvement goal should be stabilizing its behavior. If it is under control, the focus is on finding the common causes and the improvement goal should be improving its performance and capability.

### 3.2 DCA Preparation

This activity comprises the preparation for the DCA meeting by selecting the sample of defects and identifying the systematic errors leading to several of those defects.

**Apply Pareto Chart and Select Samples.** This task refers to finding the clusters of defects where systematic errors are more likely present. Systematic errors lead to several defects of the same type. Thus, Pareto charts can help to identify those clusters, by using the defect categories as the discriminating parameter. In DPPI, the samples should be related to the defect categories that contain most of the defects. An example of a real project module (identified by the acronym MPTV) Pareto chart is

shown in Figure 5. It shows that most defects were of type incorrect fact, and that the sum of incorrect facts and omissions represents about 60% of all defects found.

**Find Systematic Errors.** This task comprises analyzing the defect sample (reading the description of the sampled defects) in order to find its systematic errors. Only the defects related to those systematic errors should be considered in the DCA meeting. At this point the moderator could receive support from representatives of the document authors and the inspectors involved in finding the defects.

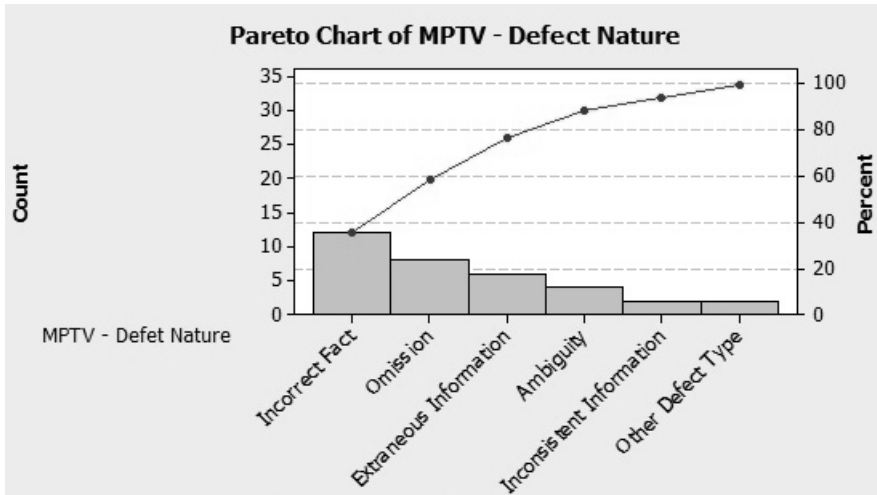


Fig. 5. Software defect Pareto chart [11]

### 3.3 DCA Meeting

In this activity, mandatory representatives of the authors and of the software engineering process group (SEPG) support the moderator. A description of the two tasks involved in this activity follows.

**Identifying Main Causes.** This task represents the core of the DPPI approach. Given the causal model elaborated based on prior DCA meeting results for the same development activity considering similar projects (by feeding the Bayesian network with the identified causes for the defect types), the probabilities for causes to lead to the defect types related to the systematic errors being analyzed can be calculated (using the Bayesian diagnostic inference).

Afterwards, those probabilities support the DCA meeting team in identifying the main causes. Therefore, probabilistic cause-effect diagrams for the defect types related to the analyzed systematic errors can be used. The probabilistic cause-effect diagram was proposed in [10]. It extends the traditional cause-effect diagram [15] by (i) showing the probabilities for each possible cause to lead to the analyzed defect type, and (ii) representing the causes using grey tones, where the darker tones represent causes with

higher probability. This representation can be easily interpreted by causal analysis teams and highlights the causes with greater probabilities of causing the analyzed defect type.

Figure 6 shows the probabilistic cause-effect diagram for defects of type “incorrect fact” in a real project. In this diagram it can be seen that, for this project (based on its prior iterations), typically the causes for incorrect facts are “lack of domain knowledge” (25%), “size and complexity of the problem” (18.7%), and “oversight” (15.6%). Figure 7 shows the underlying Bayesian network and its diagnostic inference for defects of type “incorrect fact” (built using the Netica Application software [22]).

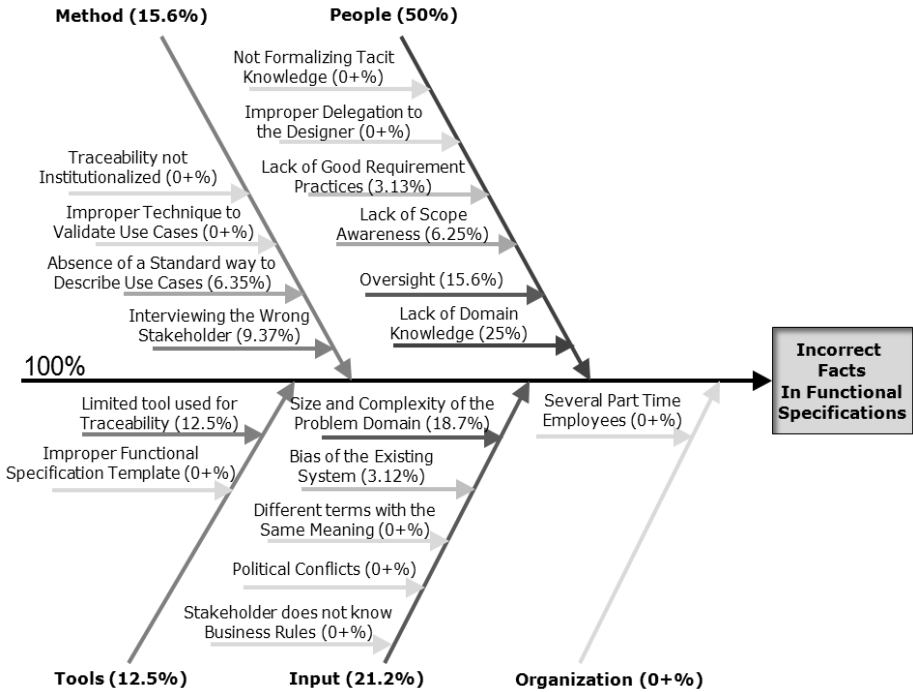


Fig. 6. DPPI’s probabilistic cause-effect diagram, adapted from [11]

The main hypotheses were related to the fact that showing such probabilistic cause-effect diagrams could help to effectively use the cause-effect knowledge stored in the Bayesian network to improve the identification of causes during DCA meetings. In fact, in the particular experience described in [11], the identified causes for the analyzed systematic errors were “lack of domain knowledge”, “size and complexity of the problem”, and “oversight”. Those causes correspond to the three main causes of the probabilistic cause-effect diagram and were identified by the author, inspectors, and SEPG members reading the defect descriptions.

Thus, the probabilistic cause-effect diagram showed itself helpful in this case. Afterwards, experimental investigations indicated that those probabilistic cause-effect diagrams may in fact increase the effectiveness and reduce the effort of identifying defect causes during DCA meetings [12].

According to DPPI, one of the outcomes of a meeting are resulting causes for the analyzed defect type that can be used to feed the Bayesian network, so that the probabilities of the causes can be dynamically updated, closing the feedback cycle for the next DCA event based on a consensus result of the team on the current DCA event. A prototype of tool support automating this dynamic feedback cycle was also built [12].

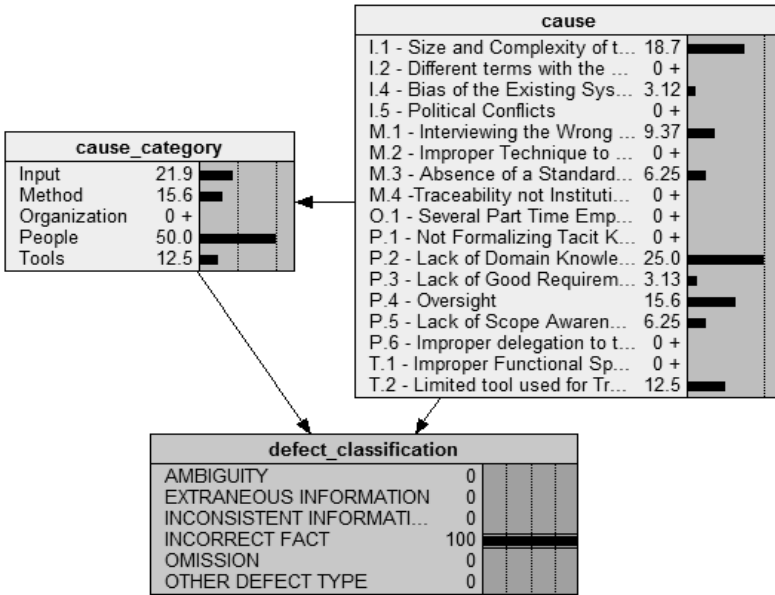


Fig. 7. DPPI’s probabilistic cause-effect diagram underlying Bayesian network inference for incorrect facts, adapted from [11]

**Propose Actions to Prevent Causes.** In this task, actions should be identified to improve the process assets in order to prevent the identified causes. Examples of actions proposed during the proof of concept, in which the DCA meeting was simulated retroactively based on real project data, addressing the causes “lack of domain knowledge” and “size and complexity of the problem”, were: (i) studying the domain and the pre-existing system; and (ii) modifying the functional specification template by creating a separate section for the business rules, listing them all together.

### 3.4 Development Activity Improvement

Finally, the action proposals should be implemented by a dedicated team and managed until their conclusion. After implementation, the changes to the process should be communicated to the development team. The conclusion of the actions and



the effort of implementing them should be recorded. The effect of such actions on improving the development activity will be objectively seen in DPPI's Development Activity Result Analysis activity (Section 3.1), once data on the defects for the same activity is collected in the next similar project (or in the next iteration of the same project) and DPPI is launched again.

## **4 Applying DPPI in Industry**

The results of the proof of concept and of the set of experimental studies indicated that DPPI could bring possible benefits to industrial software development practice. However, at this point there was no real industrial usage data (in the context of a real software development lifecycle) supporting such claims and allowing further understanding on the approach's industry readiness. According to Shull et al [23], applying a new proposed technology to an industrial setting of a real software development lifecycle is important to allow investigating any unforeseen negative/positive interactions with industrial settings.

Given the potential benefits, Kali Software decided to adopt the DPPI in May 2012 in the context of one of its biggest development projects. Kali Software is a small sized (~25 employees) Brazilian company that worked with software development projects for nearly a decade (November 2004 to March 2013), providing services to customers in different business areas in Brazil and abroad. The following subsections provide more information on the development project context, the preparation of the causal models, and the use of DPPI and its obtained results.

### **4.1 Project Context**

The development project was building a customized Web-based ERP system, designed to meet the specific business needs of Tranship, a medium size (~400 employees) naval company that provides marine support services. That system included several modules (e.g. user management, administration, operations and services, crew allocation, billing, and financial), and to date more than 80 use cases were implemented resulting in an ERP system with 82.931 lines of 434 Java classes manipulating 167 tables.

In May 2012, four modules were already delivered to the customer (user management, administration, operations and services, and crew allocation). Defects from the inspections performed on the functional specification (containing requirements, use case descriptions and prototypes) and technical specifications (containing UML package and domain class diagrams) of these modules were analyzed retrospectively by three representatives of the development team in order to feed the Bayesian network with causes for the different defect types. Later (from May to December 2012) the DPPI was applied to the functional and technical specification

documents of the fifth (billing) and sixth (financial) modules. Up to this point DPPI had only been applied to functional specifications; therefore the focus herein is on the technical specifications, although some quantitative results from applying it to the functional specifications are also provided. Details on these modules, including the number of use cases (#UC), the size in use case points (#UCP), the number of domain model classes (#DC), the total development effort in hours (Effort), the effort spent on inspecting the technical specification (Insp. Effort), and the number of defects found in the technical specification (#Defects) are shown in Table 1.

**Table 1.** Details on the project's modules

	User Mgt.	Adm.	Op. & Serv.	Allocation	Billing	Financial
#UCs	3	12	15	10	15	22
#UCP	25.2	124.9	154	102.7	165.6	242.2
#DC	4	26	29	24	21	52
Effort	176	980	1240	820	1340	1980
Insp. Effort	6	10	12	10.5	13.5	15
#Defects	4	12	16	13	23	17

As highlighted in [11], in order to apply DPPI, a defect type taxonomy should be in place. The defect type taxonomy used contained the following categories: ambiguity, extraneous information, GRASP design principle violation, inconsistency, incorrect fact, and omission. GRASP stands for General Responsibility Assignment Software Patterns, and the patterns describe basic design principles [25]. The idea of adding the GRASP related category to the taxonomy proposed by Shull [24] was to assure the use of good design practices (such as low coupling and high cohesion, among others) in the technical specifications.

## 4.2 Preparing the Causal Models

As mentioned before, functional and technical specification defects of the first four modules were analyzed retroactively with representatives of the development team in order to feed the Bayesian network with causes for the different defect types. Hereafter, our discussion will be scoped to the technical specifications. Table 2 shows the ten identified systematic errors for the most frequent technical specification defect types.

The causes identified for those systematic errors in the retroactive DCA meetings are shown in Table 3. This data was used to feed the Bayesian network so that it would be possible to use its inference when applying DPPI to the project (in the fifth module). Performing such retroactive analyses is not mandatory in order to use DPPI, but, otherwise, in the first sessions, DPPI's Bayesian network would only be able to learn from the DCA meeting results, not supporting the team in obtaining them.

**Table 2.** Systematic errors identified for the most frequent defect types

Module	Defect Type	Systematic Error
Module 1 (User Mgt.)	1.1: Inconsistency.	1.1.1: Inconsistency of standards (for instance, different conventions for the same type of elements).
	1.2: Incorrect fact.	1.2.1: Basic UML representation problems (for instance, using aggregation where a simple association is expected).
Module 2 (Adm.)	2.1: GRASP violation.	2.1.1: High coupling and low cohesion.
	2.2: Incorrect fact.	2.2.1: Incorrect facts in associations between classes.
	2.3: Omission.	2.3.1: Omitting classes and attributes.
Module 3 (Op. & Serv.)	3.1: GRASP violation.	3.1.1: high coupling and low cohesion. 3.1.2: not using inheritance where expected.
	3.2: Inconsistency.	3.2.1: Technical specification not matching current version of the requirements.
Module 4 (Allocation)	4.1: GRASP violation.	4.1.1: low cohesion.
	4.2: Omission.	4.2.1: Omitting classes and attributes.

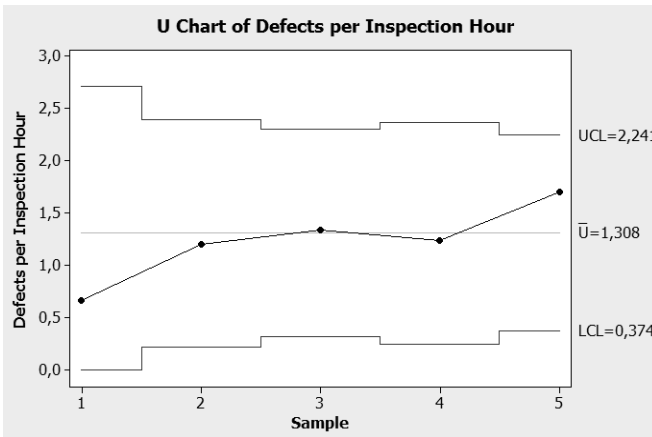
**Table 3.** Causes identified for the systematic errors of the first four modules

Category	Cause	Systematic Errors
Input	Requirements ambiguity	2.2.1.
	Requirements inconsistency	3.2.1.
	Requirements incorrect fact	2.2.1.
	Requirements omission	2.3.1, 4.2.1.
	Requirements volatility	3.2.1, 4.2.1.
	Size and Complexity of the Problem	2.1.1, 3.1.1, 4.1.1.
Method	Absence of a Standard for Models	1.1.1.
	Inefficient Req. Management	3.2.1, 4.2.1.
	Little time for Modelling	2.3.1.
	Starting Modelling to Soon	3.2.1.
Organization	Focus on Customer Deliveries	1.2.1.
	Unsustainable Pace	3.1.2.
People	Lack of Domain Knowledge	2.1.1, 2.2.1, 2.3.1, 4.1.1, 4.2.1.
	Lack of Good Design Practices	2.1.1, 3.1.1, 3.1.2, 4.1.1.
	Lack of UML Knowledge	1.2.1.
	Oversight	1.1.1, 2.2.1, 2.3.1.
Tools	Reverse Engineering Bias	2.1.1.
	Traceability Not Properly Explored	2.3.1, 4.2.1.

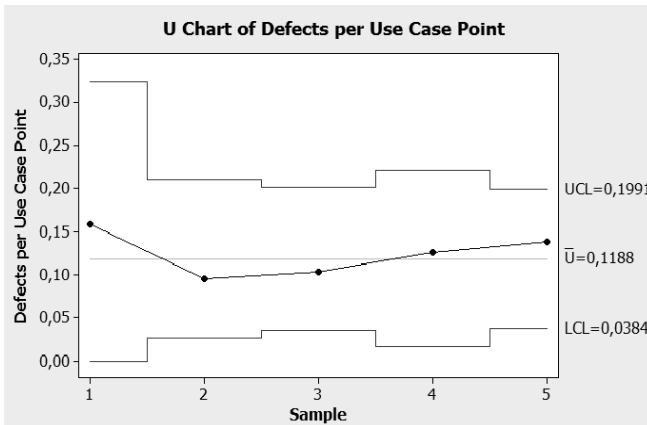
### 4.3 Use in Industry

Hereafter we describe the application of each DPPI activity to conduct DCA on technical specifications in an industrial setting and in the context of a real development lifecycle (the context described in Section 4.1).

**Development Activity Result Analysis.** Figures 8 and 9 present respectively the U-charts showing defects per inspection hour and defects per size for the data gathered for the first five modules. These Figures show a trend of an almost stable behavior. Based on this information, the quantitative improvement goal was set to reduce defect rates by 50 percent (from 0.14 defects per function point to 0.07 defects per function point), since this result had already been achieved in other organizational contexts [3][4][5].



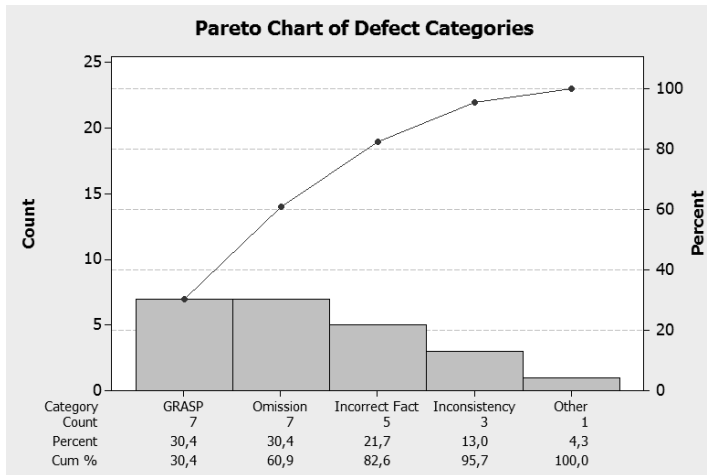
**Fig. 8.** Defects per inspection hour of the first five project modules



**Fig. 9.** Defects per use case point of the first five project modules

**DCA Preparation.** This activity relates to preparing for the DCA meeting by sampling defects (plotting a Pareto chart) and identifying the systematic errors leading to several defects of the same type by reading the sampled defects. Figure 10 shows the Pareto chart. In this chart, it is possible to observe that more than 60% of the

technical specification defects are related to violating GRASP design principles and to omission. Therefore, defects from these two categories were sampled in order to identify systematic errors; this was carried out by the moderator during a one-hour meeting with the first author and one of the two inspectors who found the defects.



**Fig. 10.** Pareto chart of the defect categories

The systematic errors related to the GRASP category were high coupling (3 defects) and using inheritance improperly (2 defects). The systematic error related to the Omission category was omitting classes and attributes (5 defects). Since the total amount of defects was low during the meeting the team also read the incorrect facts, in which another systematic error could be identified: misunderstanding several parts of the functional specification.

**DCA Meeting.** In this meeting, the moderator, the first author and a software engineering process group member identified the main causes for each of the systematic errors. The probabilistic cause-effect diagram for each defect type was used to support this task. The Bayesian network built (using the Netica tool) based on the data provided in Table 3 is shown in Figure 11.

The diagnostic inferences of this network based on the defect types (GRASP violation and Omission) of the systematic errors to be analyzed are shown in Figure 12. Based on these inferences it was possible to obtain the probabilistic cause-effect diagrams. For instance, the diagram to support analyzing the systematic errors associated to the GRASP violation defects is shown in Figure 13. Hence, the probabilistic cause-effect diagrams were used to support the DCA meeting team in identifying the causes.

Of course the data obtained from only four modules is still initial, to be completed with the results of each new DCA meeting. In this particular case, the diagram of Figure 13 shows that, in prior projects, Lack of Good Design Practices (36.4%), the Size and Complexity of the Problem Domain (27.3%), and the Lack of Domain Knowledge (18.2%) usually caused the GRASP violation defects.

In fact, the identified causes where Lack of Good Design Practices and the Size and Complexity of the Problem Domain, the latter led to high coupling and inheritance problems due to difficulties in understanding the entities and their responsibilities.

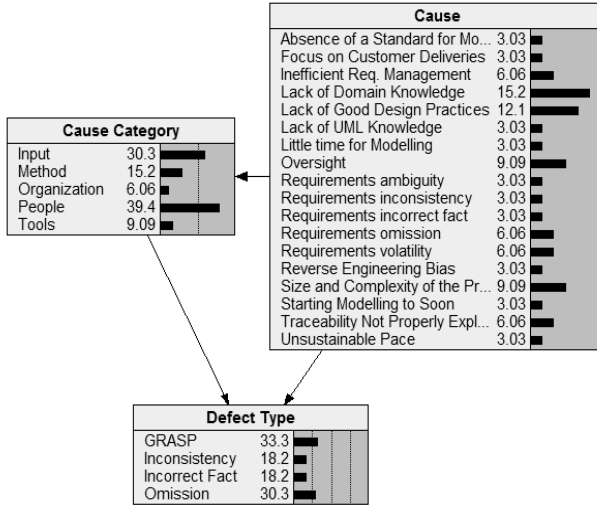


Fig. 11. The Bayesian network built based on the DCA meetings of the four prior modules

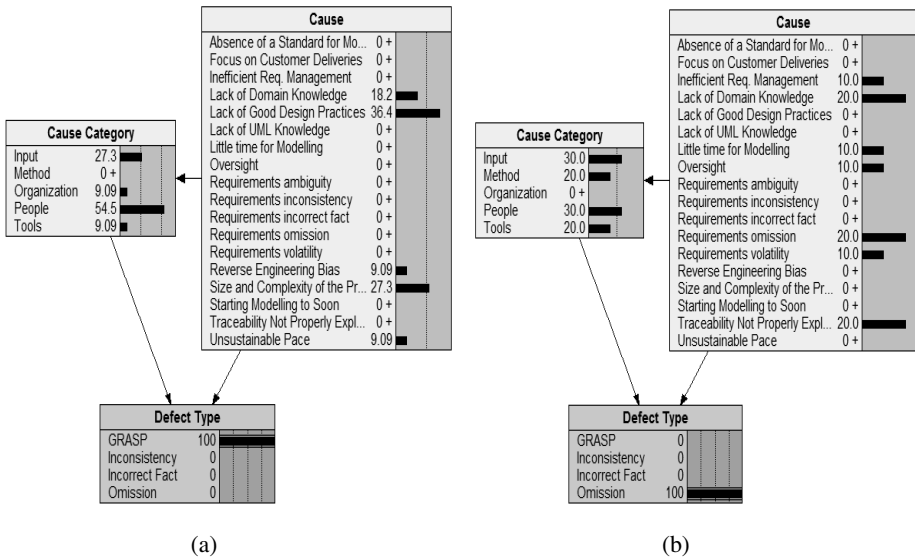


Fig. 12. Bayesian diagnostic inference for defect types GRASP (a) and Omission (b)

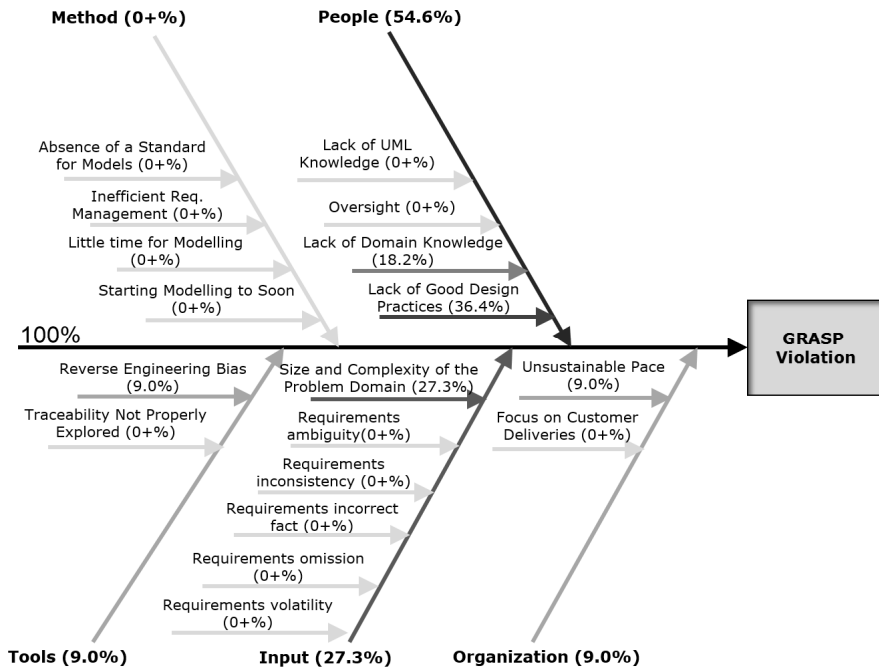


Fig. 13. Probabilistic cause-effect diagram for GRASP violation defects

Regarding the systematic error of omitting classes and attributes, the causes were Requirements Omissions, in the initial versions of the functional specification, and Inefficient Requirements Management, not communicating changes in the requirements properly to the designer.

Once the causes were identified, the meeting continued with action proposals addressing them. Five actions were proposed to treat the four identified causes: (i) providing training on GRASP design principles, (ii) publishing a list with common design defects (iii) providing training on the problem domain, with an overview of the whole system and the relation between its concepts, (iv) reinforcing the importance of avoiding omissions in the functional specifications, and (v) adjusting the requirements management process to make sure that changes to the requirements are always communicated to the designer.

After the meeting was finished, identified causes for the defect types were fed into the Bayesian network, updating the causal model for the development activity and closing the knowledge learning feedback cycle.

**Development Activity Improvement.** Finally, DPPI's last activity comprises implementing the proposed actions and communicating the changes to the development team. All five proposals were implemented. The effect of implementing them could be seen when DPPI was launched in order to analyze the defects of the sixth module's technical specification.

The U-charts plotted during DPPI's Development Activity Result Analysis (the first activity) of the sixth module are shown in Figure 14 and Figure 15.

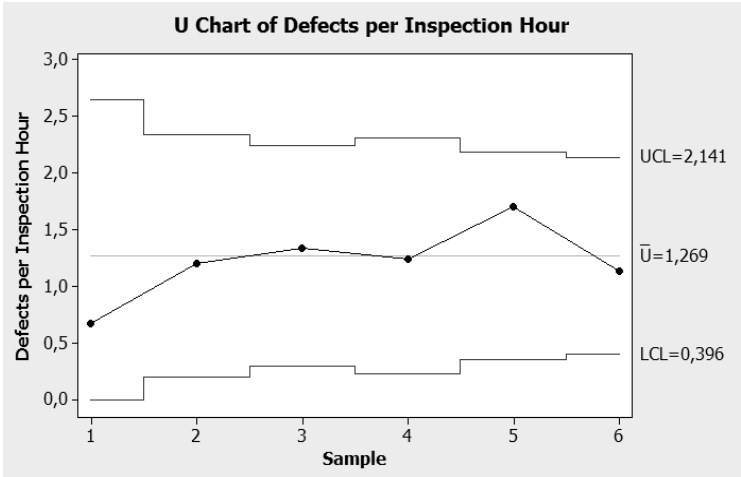


Fig. 14. Defects per inspection hour, including the sixth module

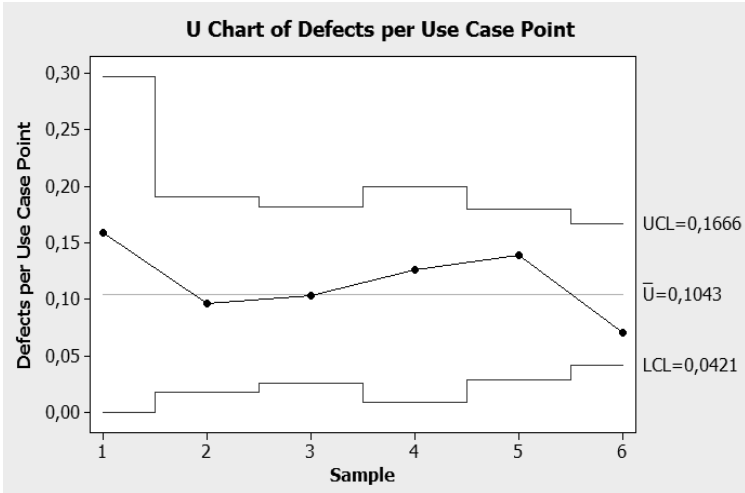


Fig. 15. Defects per use case point, including the sixth module

Those charts can be compared to the quantitative goal established in the prior DPPI enactment. As expected [7], defect rates reduced in about 50 percent (from 0.14 to 0.07). This expected defect rate reduction behavior of about 50 percent (in fact, 46%) also happened to the functional specification activity. Moreover, for the functional specifications, analyzing the defects allowed producing a set of nine “improvement factors” that can be used by the employees for writing better use cases.

The fifth module of this project represented the first time that DPPI was applied end-to-end to a real project, by the project team and allowing to present defect rate reductions observed in a subsequent iteration.



The total effort of applying DPPI was about 15.5 hours, 1 hour spent by the moderator in the Development Result Analysis activity, 2 hours spent by each team member in the DCA Preparation activity (6 hours at all), and 2.5 hours spent by each team member in the DCA Meeting (7.5 hours at all). Of course this effort does not consider the time spent in implementing the action proposals, otherwise the GRASP training (8 hours), publishing the list of design problems (2 hour), the domain training (4 hours) and the adjustments to the requirements management process (1 hour) would have to be accounted. Nevertheless, comparing such effort to defect rate reductions of 50 percent the trade-off is clearly favorable.

## 5 Conclusions

The DPPI approach resulted from research following an experimental strategy. Its conceptual phase considered evidence-based guidelines acquired through systematic reviews and feedback from experts in the field. Afterwards the approach was evolved based on results of an initial proof of concept and a set of experimental studies.

The results of the proof of concept and of the set of experimental studies indicated that DPPI could bring possible benefits to industrial software development practice. However, even though the proof of concept showed the feasibility of using the approach, it was applied retroactively by the researcher. Hence, not allowing observing some interesting aspects, such as application effort and results on defect rate reductions. The set of experimental studies, on the other hand, allowed evaluating the usefulness of the probabilistic cause-effect diagrams in identifying causes. Nevertheless, by not being applied as part of a real development life cycle they were not able of providing further information on the results of addressing those causes.

In this paper, we extended our research by outlining the complete adopted experimental strategy, providing an overview of DPPI and presenting additional results from applying it to a real industrial software development lifecycle. In this experience, DPPI was successfully applied to different development activities (functional specification and technical specification), allowing further comprehension on its industry readiness and objectively measuring the effort and the obtained benefits. The total application effort was reasonably low (on average 15.5 hours) when compared to the obtained benefits (reducing defect rates in 46 percent in functional specifications and in 50 percent for technical specifications).

Although these results were obtained in a specific context, not allowing any external validity inferences on other types of projects or other industrial contexts, we believe that applying DPPI to a real software project lifecycle helped to understand possible benefits and constraints of using the approach from an industrial perspective.

**Acknowledgments.** We thank David N. Card for his contributions to our research, the subjects involved in the experimental study, the COPPETEC Foundation, and Tranship. Without their support this paper would not have been possible. Thanks also to CAPES and CNPq for financial support.

## References

1. Card, D.N.: Defect Analysis: Basic Techniques for Management and Learning. In: *Advances in Computers*, ch. 7, vol. 65, pp. 259–295 (2005)
2. Card, D.N.: Defect Causal Analysis Drives Down Error Rates. *IEEE Software* 10(4), 98–99 (1993)
3. Mays, R.G., Jones, C.L., Holloway, G.J., Studinski, D.P.: Experiences with Defect Prevention. *IBM Systems Journal* 29(1), 4–32 (1990)
4. Dangerfield, O., Ambardekar, P., Paluzzi, P., Card, D., Giblin, D.: Defect Causal Analysis: A Report from the Field. In: *Proceedings of International Conference of Software Quality*, American Society for Quality Control (1992)
5. Jalote, P., Agrawal, N.: Using Defect Analysis Feedback for Improving Quality and Productivity in Iterative Software Development. In: *3rd ICICT*, Cairo, pp. 701–713 (2005)
6. Boehm, B., Basili, V.R.: Software Defect Reduction Top 10 List. *IEEE Computer* 34(1), 135–137 (2001)
7. Kalinowski, M., Card, D.N., Travassos, G.H.: Evidence-Based Guidelines to Defect Causal Analysis. *IEEE Software* 29(4), 16–18 (2012)
8. Kalinowski, M., Travassos, G.H., Card, D.N.: Guidance for Efficiently Implementing Defect Causal Analysis. In: *VII Brazilian Symposium on Software Quality (SBQS)*, Florianopolis, Brazil, pp. 139–156 (2008)
9. Mafra, S.N., Barcelos, R.F., Travassos, G.H.: Aplicando uma Metodologia Baseada em Evidência na Definição de Novas Tecnologias de Software. In: *Proc. of the XX Brazilian Symposium on Software Engineering (SBES)*, Florianopolis, Brazil, pp. 239–254 (2006)
10. Kalinowski, M., Travassos, G.H., Card, D.N.: Towards a Defect Prevention Based Process Improvement Approach. In: *34th Euromicro Conference on Software Engineering and Advanced Applications*, Parma, Italy, pp. 199–206 (2008)
11. Kalinowski, M., Mendes, E., Card, D.N., Travassos, G.H.: Applying DPPI: A Defect Causal Analysis Approach Using Bayesian Networks. In: Ali Babar, M., Vierimaa, M., Oivo, M. (eds.) *PROFES 2010. LNCS*, vol. 6156, pp. 92–106. Springer, Heidelberg (2010)
12. Kalinowski, M., Mendes, E., Travassos, G.H.: Automating and Evaluating Probabilistic Cause-Effect Diagrams to Improve Defect Causal Analysis. In: Caivano, D., Oivo, M., Baldassarre, M.T., Visaggio, G. (eds.) *PROFES 2011. LNCS*, vol. 6759, pp. 232–246. Springer, Heidelberg (2011)
13. Pai, M., McCulloch, M., Gorman, J.D.: Systematic reviews and meta-analyses: An illustrated step-by-step guide. *National Medical Journal of India* 17(2) (2004)
14. Kitchenham, B.A., Charters, S.: Guidelines for Performing Systematic Literature Reviews in Software Engineering. Technical Report (version 2.3), Keele University (2007)
15. Ishikawa, K.: *Guide to Quality Control*. Asian Productivity Organization, Tokyo (1976)
16. SEI: *CMMI for Development (CMMI-DEV), Version 1.3. CMU/SEI-2010*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University (2010)
17. Kalinowski, M., Spínola, R.O., Dias Neto, A.C., Bott, A., Travassos, G.H.: Inspeções de Requisitos de Software em Desenvolvimento Incremental: Uma Experiência Prática. In: *VI Brazilian Symposium on Software Quality (SBQS)*, Porto de Galinhas, Brazil (2007)
18. Kalinowski, M., Travassos, G.H.: A Computational Framework for Supporting Software Inspections. In: *International Conference on Automated Software Engineering (ASE 2004)*, Linz, Austria, pp. 46–55 (2004)
19. Fagan, M.E.: Design and Code Inspection to Reduce Errors in Program Development. *IBM Systems Journal* 15(3), 182–211 (1976)
20. Pearl, J.: *Causality Reasoning, Models and Inference*. Cambridge University Press (2000)

21. Hong, G., Xie, M., Shanmugan, P.: A Statistical Method for Controlling Software Defect Detection Process. *Computers and Industrial Engineering* 37(1-2), 137–140 (1999)
22. Netica Application, <http://www.norsys.com/netica.html>
23. Shull, F., Carver, J., Travassos, G.H.: An Empirical Methodology for Introducing Software Processes. In: *European Software Engineering Conference, Vienna, Austria*, pp. 288–296 (2001)
24. Shull, F.: *Developing Techniques for Using Software Documents: A Series of Empirical Studies*. Ph.D. thesis, University of Maryland, College Park (1998)
25. Larman, G.: *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development*. Prentice Hall (2008)

# Business Intelligence in Software Quality Monitoring: Experiences and Lessons Learnt from an Industrial Case Study

Alexander Kalchauer<sup>2</sup>, Sandra Lang<sup>1</sup>, Bernhard Peischl<sup>1</sup>,  
and Vanesa Rodela Torrents<sup>2</sup>

<sup>1</sup> Softnet Austria, Inffeldgasse 16b/II, 8010 Graz, Austria  
{bernhard.peischl,sandra.lang}@soft-net.at  
<http://www.soft-net.at>

<sup>2</sup> Technische Universität Graz, Institut für Softwaretechnologie,  
Inffeldgasse 16b/II, 8010 Graz, Austria  
{kalchauer,torrents}@student.tuGraz.at

**Abstract.** In this industrial experience article we briefly motivate continuous, tool-supported monitoring of quality indicators to facilitate decision making across the software life cycle. We carried out an industrial case study making use of a web-enabled, OLAP-based dashboard to measure and analyse quality indicators across various dimensions. We report on lessons learnt and present empirical results on execution times regarding different groups of queries implementing the desired quality metrics. We conclude that today's business intelligence (BI) solutions can be employed for continuous monitoring of software quality in practice. Furthermore, BI solutions can act as an enabler for any kind of data-driven research activities within companies. Finally we point out some issues, that need to be addressed by future data-driven research in the area of software measurement.

**Keywords:** business intelligence, lessons learned, industrial case study, web-enabled software dashboard.

## 1 Motivation

The new generation of interconnected and open IT systems in areas like health-care, telematics services for traffic monitoring, or energy management are evolving dynamically, for example, software updates are carried out in rather short cycles under presence of a complex technology stack. Tomorrow's IT systems are liable to stringent quality requirements (e.g. safety-, security or compliance issues) and evolve continuously, that is, the software habitually is subject of change. Given this context, monitoring of quality attributes becomes a necessity. Efficient and effective management of a software product or service thus requires continuous, tool-supported monitoring of well-defined quality attributes.

The role of software as a driver of innovation in business and society leads to an increasing industrialization of the entire software life cycle. Thus, the development and operation of software nowadays is perceived as mature engineering

discipline (even in the secondary sector) and the division of labor is driven by coordinated tools (in the sense of an integrated tool chain). As a consequence, monitoring of quality attributes has to be carried out holistically, i.e. taking into account various quality dimensions. This includes the continuous monitoring of process- (e.g. processing time of for bugs or change requests), resource- (e.g. alignment of bugs to software engineers) and product-metrics (e.g. code metrics like test coverage as well as metrics regarding abstract models of the software, if in place). However, to establish a holistic view on software quality we lack methods and tools that allow one for an integration of the relevant dimensions and (key) quality indicators.

To address these issues, as part of the research programme Softnet Austria II, a web-enabled software dashboard making use of powerful OLAP technology has been developed. This cockpit strives to provide the technological underpinning to enable the integration of the various dimensions. In this article we briefly describe the software architecture of our dashboard and point out experiences and lessons learnt in the course of a pilot project. We considered issue management, that is the treatment of defects and change requests (CRQs) alongside with related parts regarding project management. In this industrial experience paper we present exhaustive empirical results regarding the running time of the queries implementing the various quality indicators. We contribute to the state of the art in the field of quality monitoring in terms of (1) providing experiences on setting up a software dashboard relying on an open-source BI solution and (2) a performance analysis regarding the execution times.

In Section 2 we discuss the importance of metrics and quality models and point out the need for integration of the different views on software quality. In Section 3 we subsume the main ideas behind OLAP, typical usage scenarios and operations and the overall architecture of our web-enabled dashboard solution. In Section 4 we present our industrial case study conducted with a company from the secondary sector<sup>1</sup>. In the context of the prevailing processes around CRQs we outline (1) examples of quality metrics, (2) show how to use the OLAP technology for ad-hoc analysis of an industrial software repository, and (3) provide empirical insights on the execution time of the various multi-dimensional queries. In Section 5 we report on lessons learned from our industrial case study, Section 6 discusses related work and Section 7 points out open issues and concludes our industrial experience paper.

## 2 Measuring Software Quality

In general software quality is nowadays perceived as the degree of fulfilment of either explicitly specified or tacit requirements of the stakeholder (e.g. the ISO/IEC 25010:2011 standard [1] follows this notion). A specific quality model makes this abstract definition applicable. Usually a quality model defines certain quality attributes (e.g. test coverage on the level of code, test coverage on

---

<sup>1</sup> Due to a non-disclosure and confidentiality agreements we do not mention our cooperation partner and made relevant data anonymous.

the level of requirements, processing time of defects or CRQs, testability of the product, ...) and allows one for refining these attributes in terms of specific, measurable characteristics of the resources, the product or the underlying processes. The quality model thereby specifies a hierarchy of measurable characteristics and ideally allows one for definition, assessment and prediction of software quality [7]. Although definition, assessment and prediction of quality are different purposes, the underlying tasks are not independent of each other. Obviously, it is hard to assess quality without knowing what it actually constitutes. Likewise one cannot predict quality without knowing how to assess it [7]. Quality models may serve as a central repository of information regarding software quality. Ideally the different tasks in software quality engineering, e.g. definition, assessment, and prediction of quality attributes should rely on the same model. However, in practice different (often implicit and incomplete) models are used for these tasks. Therefore a common infrastructure that supports definition, assessment and prediction of software quality using a common representation of the underlying quality-relevant data and metrics is highly desirable. As mentioned previously, our dashboard solution focuses on continuous monitoring of the software life cycle particularly addressing the need for a smooth integration of various quality dimensions. This allows one for analysing the relevant aspects regarding quality attributes, including the early detection of erroneous trends. The early detection of trends in turn is the foundation of setting remedial countermeasure in motion.

### 3 Exploiting Business Intelligence in Software Engineering

The main idea behind BI is to transform simple data into useful information which enhances the decision making process by basing it on a wide knowledge about itself and the environment. That minimizes risks and uncertainties derived from any decision that a company has to take [17].

Moreover, business intelligence contributes to translate the defined objectives of a company into indicators with the possibility of analysing them from different points of view. That transforms the data into information that, not only is able to answer questions about current or past events but it also makes possible building models to predict future events [6].

#### 3.1 Architecture of the Web-Enabled Dashboard

The concepts outline in the previous sections are typically used in various business areas for measurement, analysis and mining of specific data pools. Our dashboard primary serves the purpose of carrying out data-driven research particularly addressing the integration of various views on software quality (process-, product- and resources view). Figure 1 outlines the architecture of the web-enabled dashboard making use of BI technology. The dashboard uses the open source tool JPivot [10] as front end. It operates on the open source OLAP

server Mondrian [4]. JPivot allows the interactive composition of MDX (Multi-Dimensional eXpressions) queries via a Web interface that also displays the results in tabular and graphical form. Predefined queries are stored on the server and accessible from the web interface. JPivot loads these queries from the query files or takes them from the MDX editor provided through the web interface, than OLAP parses those queries through the multidimensional cubes and converts them to SQL queries which are sent to a MySQL database.

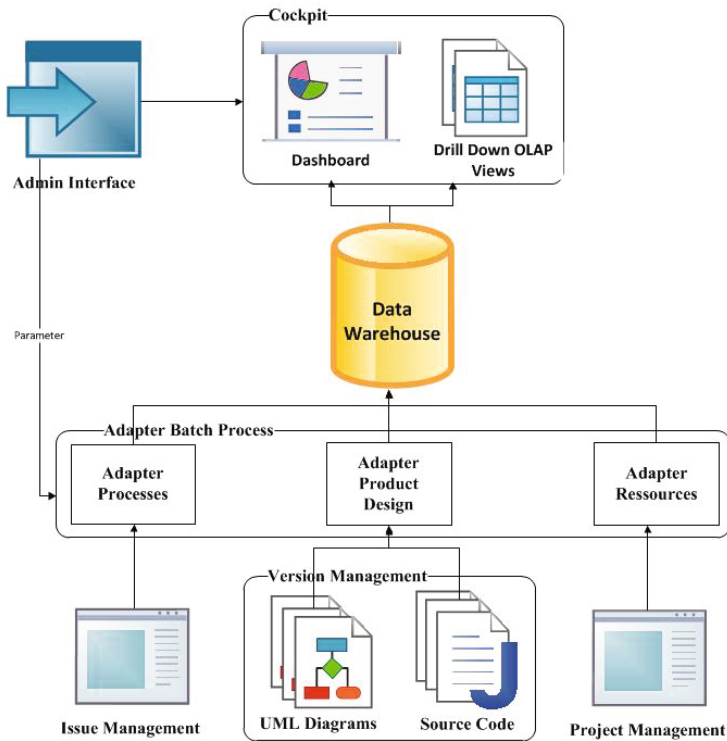


Fig. 1. Software architecture of the BI-based dashboard [11]

#### 4 Industrial Case Study: Quality Indicators for Managing Defects and Change Requests

Our cooperation partner is developing a software product and has well-defined work flows for development and maintenance in place. For our pilot project we addressed the work flows around CRQs. Our cooperation partner classifies issues according to three types: Defects, Enhancements (CRQs) and Tickets. In general issues pass through various phases: The decision phase (Decide), the implementation phase (Implement), the assessment phase (Review) and the test phase (Test). According to these categorization, the work flow around CRQs is partitioned into these four areas as illustrated in Figure 2.

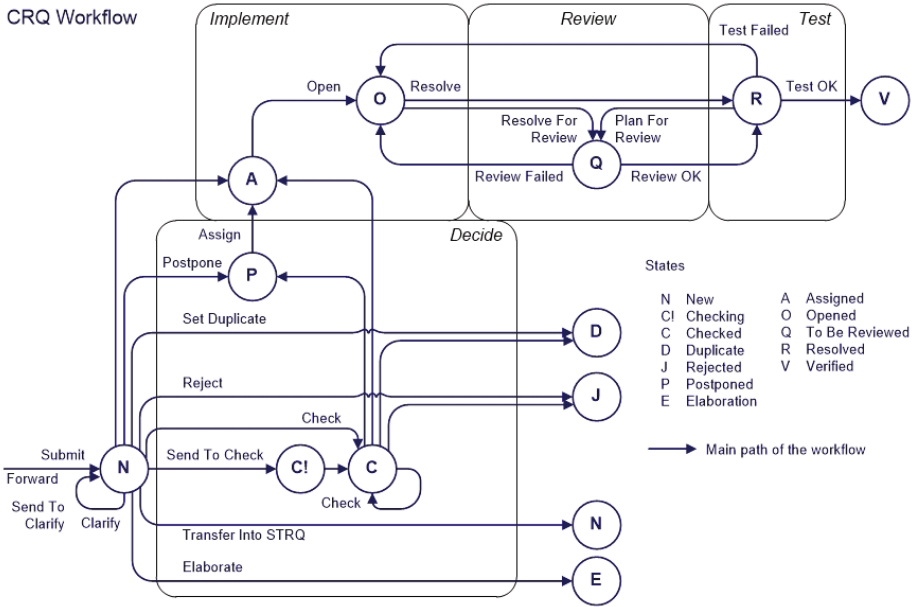


Fig. 2. Simplified workflow, states and state transitions

#### 4.1 Measuring the Management of Defects and Change Requests

Regarding the monitoring of quality indicators we distinguish between metrics with a target state from the set of inflow states  $\{A, C, C!, D, E, N, J, P\}$  and metrics with target state from the set of outflow states  $\{O, Q, R, V\}$ . Relying on this categorization, metrics have been grouped into 6 groups:

- **Group 1:** Metrics regarding the inflow states and the elapsed time from submission of an issue to a particular state. The around 20 performance indicators from this group affect the decision phase and the implementation phase. Depending on the concrete metric, drill-down and slicing into 15 to 20 different dimensions (version, milestone, product, status, priority, error severity, person, role, age etc.) is required.
- **Group 2:** Metrics regarding the outflow state and the elapsed time from submission to a particular state. The around 15 performance indicators concern the review- and testing phase and have similar requirements wrt. ad hoc analysis as group 1.
- **Group 3:** Metrics regarding the inflow states and the elapsed time on that state. The 11 performance indicators are dealing with the elapsed time of a CRQ in the decision- and implementation phase. Some of the performance indicators are drilled-down to around 15 dimensions, often standard aggregates (maximum, minimum, average) are used.



- **Group 4:** Metrics regarding outflow states and the elapsed time on that state. The 14 performance indicators require ad-hoc analysis in up to 20 dimensions.
- **Group 5 and Group 6:** These metrics have high level of complexity as these metrics relate states that are not consecutive. The complexity is caused due to the fact, that it is necessary to keep track of an issue from its submission through all its changes in order to compute the desired metrics. Nevertheless, these performance indicators need to be drilled down to a couple of dimensions. However, these metrics are less time critical as the Group 1-4 as the quality indicators regarding group 5 and 6 are used occasionally only.

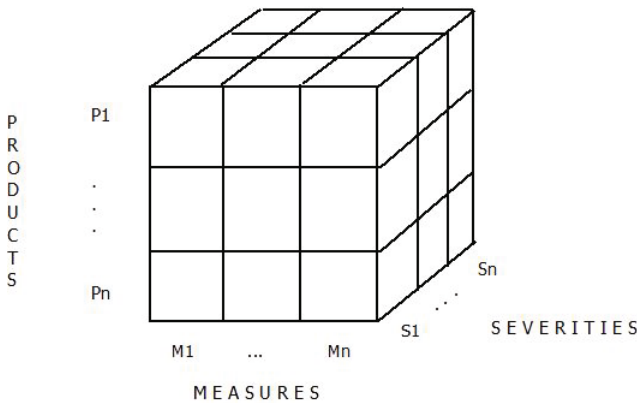
### 4.2 Queries for Ad-Hoc Analysis

The MDX language has become the standard defined by Microsoft to query OLAP systems and provide a specialized syntax for querying and manipulating the multidimensional data stored in OLAP cubes.

Table 1 shows the example of a query and the explanation of every statement following by its graphical representation. As one can see in the query four dimensions are selected:

- Time and measures on the columns.
- Products and severities on the rows.

Figure 3 and 4 illustrates the basic idea.



**Fig. 3.** The three queried dimensions form a cube

Using the *where* clause, one has the possibility to slice the represented cube. That means that not all the data of the four dimensions will be shown, but only those specified through the slicer defined in the *where* clause. In this case, these issues which go to the state *Resolved (R)* and were successfully tested in the previous step (*Verified (V)*).

**Table 1.** Query Example

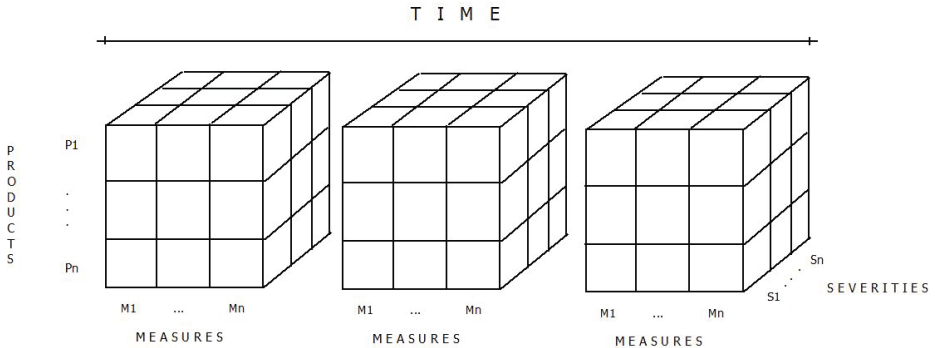
<i>Functions</i>	<i>Handled data</i>	<i>Description</i>
1 Select NON EMPTY Crossjoin	(([LastChangedDate].[All date].children, [Measures].[IssuesStatesCount], [Measures].[MinAgeInDays], [Measures].[MaxAgeInDays], [Measures].[AvgAgeInDays])	The crossjoin function returns the cross product of two sets. It is necessary because LastChangedDate and Measures are two different dimensions. Since the type given to crossjoin has to be a set, one converts the members to a set by enclosing them in curly brackets.
2 ON COLUMNS,		All the information selected before will be shown through the x axis.
3 NON EMPTY Crossjoin	(([Product].[All product].Children, [Severity].[All severity])	
4 ON ROWS		All the information selected before will be shown through the y axis.
5 From [SoftCockpit]		Cube from where one wants to extract the data
6 where	(([PrevStatus].[All status].[Verified], [Activity].[All activities].[resolved])	Slicer: Tuple which contains the members that specify the conditions that the data has to fit to be shown.

The output given by the application is a table showing the required information on the desired rows (Figure 5). Products and Milestone can be drilled down, since they compose a hierarchy with several levels.

### 4.3 Performance Analysis

During the implementation of the different sets of quality indicators, differences in the performance regarding the amount of data have been observed. To analyse this, different sets of data of the fact table have been taken into account. The following graphs and explanations show the conclusions obtained from this performance analysis. Table 2 summarizes the resources of the server on which we conducted our experiments. The test has been executed as follows: Every developed query has been tested for a different amount of years and consequently for a different amount of data.

Moreover, a relation between some queries has been noticed during the experiment as queries belonging to the same group have similar execution time



**Fig. 4.** The three previous dimensions are queried along the years which is the fourth dimension

		LastChangedDate							
		*2011				*2012			
		Kennzahlen				Kennzahlen			
Product	Milestone	IssuesStatesCount	MinAgeInDays	MaxAgeInDays	AvgAgeInDays	IssuesStatesCount	MinAgeInDays	MaxAgeInDays	AvgAgeInDays
*Product A	*All milestone	8	25,04	1.008	291,77	2	13	205	109
*Product B	*All milestone					4	20	73	33,25
*Product C	*All milestone	1	1	1	1				
*Product D	*All milestone	1	28	28	28	1	32	32	32
*Product E	*All milestone					1	94,96	94,96	94,96

**Fig. 5.** Results produced by the query of Table 1

**Table 2.** Used resources

<i>Physical Memory: 32GB</i>
<i>JVM Memory: 2GB</i>
<i>Processor: Intel Xeon CPU ES606@ 2.13 GHz, 2.13GHz (2 Processors)</i>
<i>OS: Windows Server 2008 R2 Standard</i>

characteristics. The reason why this happens can be found in the complexity of the queries which is caused by the where clause (slicer) of the query.

For the different groups of queries the following execution times have been collected:

Figure 6 shows the execution times for the queries of Group 1 with different sets of data. The average response time of all the considered metrics of Group 1 is shown in Figure 7.

Figure 8 shows the execution times for the queries of Group 2 with different sets of data. Queries 2.7 and 2.8 are excluded since they prove the correct use of the open state and have been implemented for the purpose of verification. These queries are not used for continuous monitoring of the process. To ease the illustration, queries 2.10 and 2.13 are also excluded as they can be computed

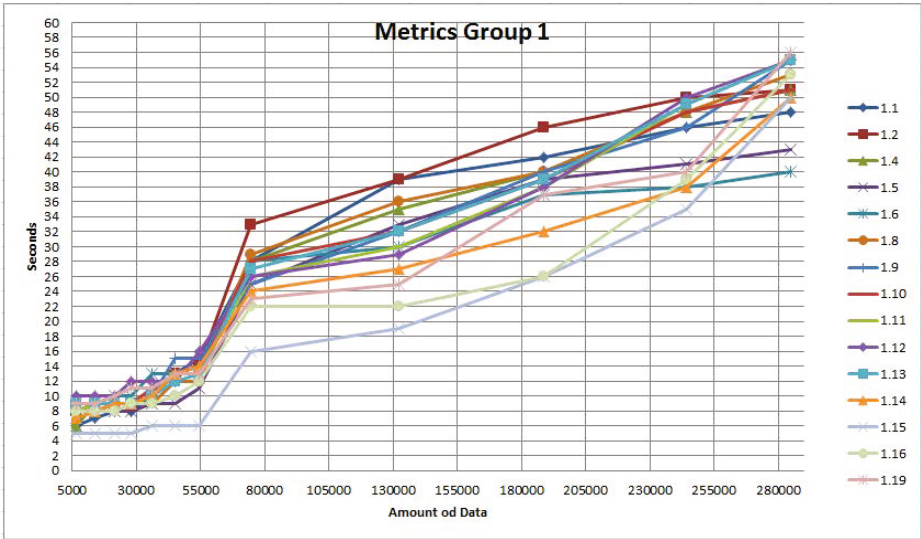


Fig. 6. Response time regarding the amount of the data for Group 1

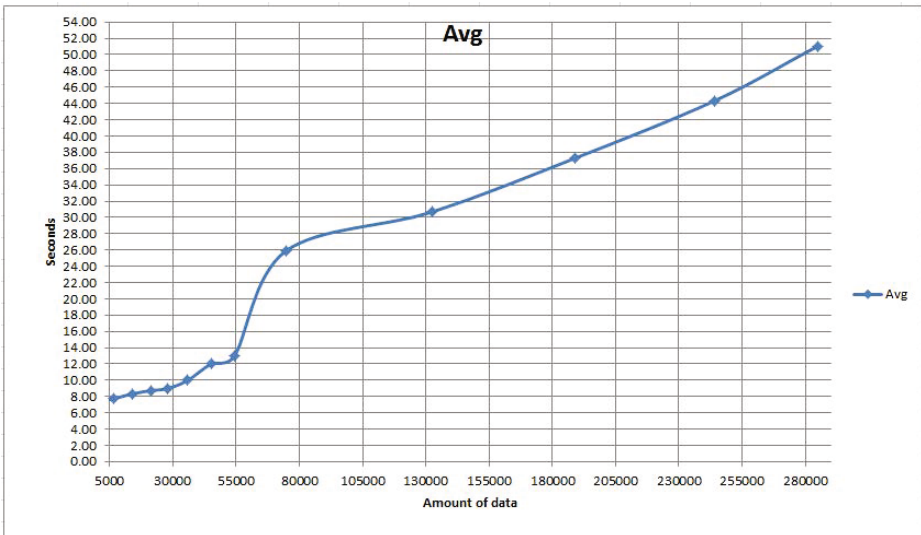


Fig. 7. Average over the response time of Group 1

within a second even for a big amount of data. The average of all the quality indicators of Group 2 is shown in Figure 9.

Execution times for the queries of Group 3 and 4 look very similar. For Group 5 and Group 6 queries we did not evaluate the response times, because of the complexity of these queries. However, as mentioned before, these queries are not used for continuous quality monitoring and are executed only occasionally.

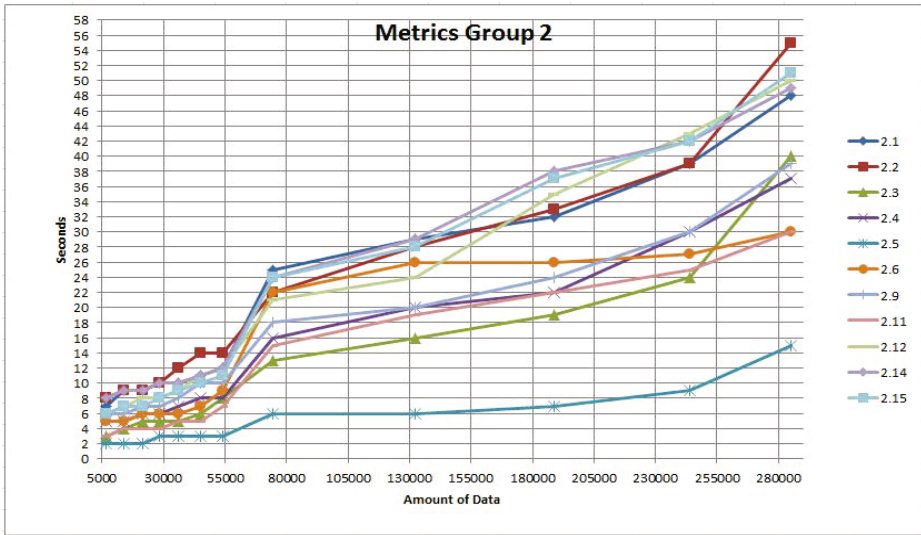


Fig. 8. Response time regarding the amount of the data for Group 2

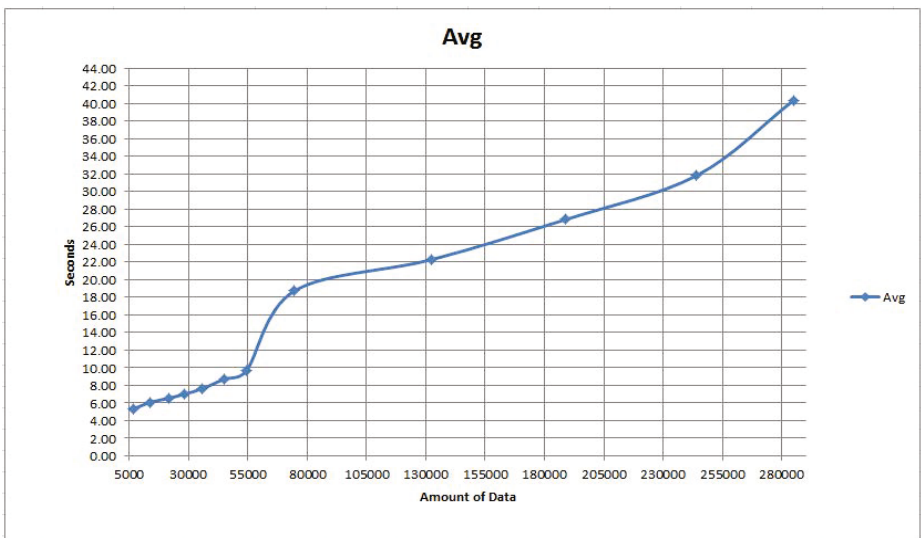
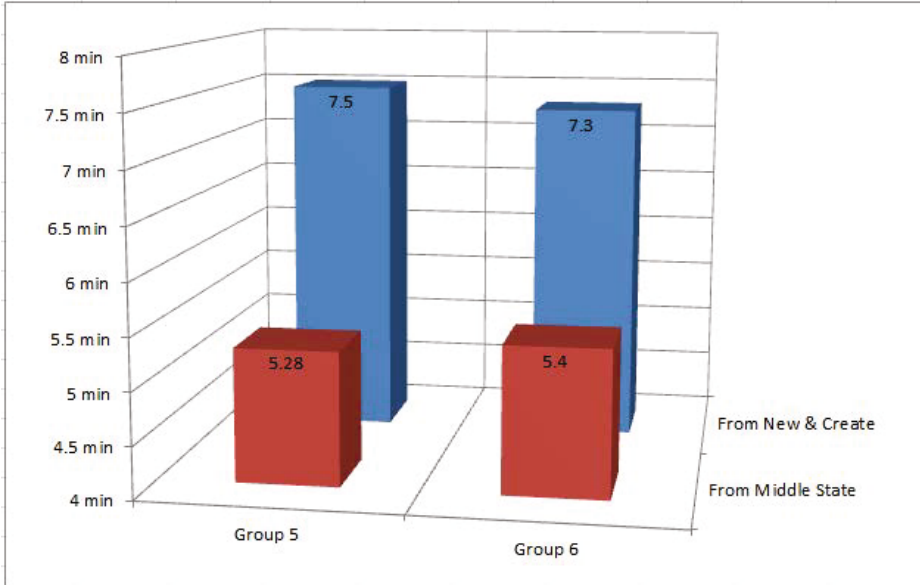


Fig. 9. Average over the response time of Group 2

However, inside these two groups one can also differentiate between the metrics which start in state New ('N', creation of an issue) and go to a non-terminating state (we refer to a middle state in the following, see Figure 2) and the metrics which start in a middle state and end in another middle state. Since the way through the hierarchy is longer for those metrics starting in state *New* ('N'), the complexity of these metrics is higher. Furthermore the number of OR

operations in their definition is higher, which increases the execution times further. Therefore a comparison between metrics from state new ('N') to a middle state and the metrics which goes from a middle state to another middle state can be of interest.

Figure 10 shows the differences in performance between those metrics.



**Fig. 10.** Performance comparison between initial and middle states queries

## 5 Discussion and Lessons Learned

As one can observe, in all the metrics groups (although the execution times are different) the shape of the graph is the same. The execution time increases in a mostly linear way up to approx. 55000 rows in the fact table. For bigger fact tables, there is a dramatic increase in the running time which is shown by the almost vertical line. From this point, the execution time is increasing following an almost linear shape. This fact shows that, with the current resources, if the amount of data to be queried is above 55000 rows for the fact table, a generalized tuning process should be carried out.

For very large cubes, the performance issues are driven mostly by the hardware, operating system and database tuning, than anything Mondrian can do.

It has been observed that for this amount of data, increasing memory resources means no change, since the execution times keep on being the same. 2GB are enough to carry out any operation with this data. Therefore a tuning for the database and Mondrian is suggested in [19].

## 6 Related Work

Software cockpits for interpretation and visualization of data were already conceptually prepared ten years ago. A reference model for concepts and definitions around software cockpits is presented in [15]. Concepts and research prototypes have been developed in research projects like Soft-pit [14] and Q-Bench [9]. Whereas Soft-pit has the goal of monitoring process metrics [14], the Q-Bench project is aimed at developing and using an approach to assure the internal quality of object oriented software systems by detecting quality defects in the code and adjusting them automatically.

The authors of [2] report that the integration of data from different analysis tools gives a comprehensive view of the quality of a project. As one of the few publications in this field, the authors also deal with the possible impact of introducing a software dashboard. The authors of [3], [5], [13], [12] report on experiences and lessons learnt regarding software dashboards .

According to [8], in the context of explicit quality models, a viable quality model has to be adaptable and the handling of the obtained values (i.e. a table result) should be easily understandable. Business intelligence systems with OLAP functionalities support these requirements and are used nowadays in several fields [20]. Therefore using BI technology for quality monitoring offers an excellent foundation for data-driven research in collaboration with companies [16].

Related to the work presented herein are industrial-strength tools, e.g. Bugzilla (<http://www.bugzilla.org>), JIRA (<http://www.atlassian.com/software/jira>), Polarion (<http://www.polarion.com>) and Codebeamer (<http://www.intland.com>), Swat4j (<http://www.codeswat.com>), Rational Logiscope (<http://www-01.ibm.com/software/awdtools/logiscope/>) or Sonar (<http://www.sonarsource.org>) strive to integrate quality metrics in an environment of heterogeneous data sources [18]. Some tools (Swat4j or Rational Logiscope) support the evaluation of quality metrics explicitly taking account a quality model. Whereas these tools are very flexible in data storing and representation, they offer few possibilities for the integration and analysis of different quality dimensions.

## 7 Conclusion

In this article we motivate that continuous, tool-supported quality monitoring is gaining more and more importance due to a new generation of interconnected and open IT systems that evolve dynamically under presence of a complex technology stack. We briefly summarize the challenges in operationalising quality indicators and point out the necessity to integrate the different quality perspectives (e.g. process-entered view, product-centred view and resource-centred view). Having a multi-dimensional view in mind, we argue that OLAP technology, as it is successfully applied in various business areas - provides a solid foundation for data-driven research in the field of software quality. We provide

an overview of the software architecture of our software dashboard which makes use of open-source OLAP technology. Afterwards we present an industrial case study carried out with a company developing a software product. The case study particularly deals with quality indicators for managing issues (change requests and defects) and comprises over 43.000 issues over a time period of 13 years. We implemented over 100 metrics that are dissolved in up to 20 different dimensions. Our exhaustive analysis of the running time for the most important quality indicators shows that today's BI technology is good enough to support ad-hoc analysis for most of the relevant quality indicators. Further we report on challenges and lessons learnt. The work presented in this industrial experience paper shows that - by relying on today's BI technology - measurements and ad-hoc analysis of relevant data is feasible, however, there is an increased need for further research dealing with the impact of measurement tools in industrial practice. Yet it is an open issue, how continuous monitoring of quality indicators effects the overall software quality. Further research should take advantage of past experiences and lessons learnt but specifically address the impact of continuous quality monitoring on software quality.

**Acknowledgement.** The work presented herein has been partially carried out within the competence network Softnet Austria II ([www.soft-net.at](http://www.soft-net.at), COMET K-Projekt) and funded by the Austrian Federal Ministry of Economy, Family and Youth (bmwfj), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in support of the Center for Innovation and Technology (ZIT). We listed the authors in alphabetical order.

## References

1. ISO/IEC 25010:2011 software engineering - software product quality requirements and evaluation (SQuaRE) - quality model. International Organization for Standardization (2011)
2. Bennicke, M., Steinbrückner, F., Radicke, M., Richter, J.-P.: Das sd&m software cockpit: Architektur und erfahrungen. In: INFORMATIK, pp. 254–260 (2007)
3. Biehl, J.T., Czerwinski, M., Smith, G., Robertson, G.G.: Fastdash: a visual dashboard for fostering awareness in software teams. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 1313–1322. ACM (2007)
4. Bouman, R., van Dongen, J.: Pentaho Solutions: Business Intelligence and Data Warehousing with Pentaho and MySQL. Wiley Publishing (2009)
5. Ciolkowski, M., Heidrich, J., Simon, F., Radicke, M.: Empirical results from using custom-made software project control centers in industrial environments. In: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 243–252. ACM (2008)
6. Dario, B.R.: DATA WAREHOUSING: Investigacin y Sistematizacin de Conceptos. PhD thesis, Universidad nacional de Crdoba (2009)
7. Deissenboeck, F., Juergens, E., Lochmann, K., Wagner, S.: Software quality models: Purposes, usage scenarios and requirements. In: ICSE Workshop on Software Quality, WOSQ 2009, pp. 9–14 (2009)



8. Deissenboeck, F., Juergens, E., Lochmann, K., Wagner, S.: Software quality models: Purposes, usage scenarios and requirements. In: ICSE Workshop on Software Quality, WOSQ 2009, pp. 9–14. IEEE (2009)
9. <http://www.qbench.de/>
10. JPivot, <http://jpivot.sourceforge.net/>
11. Lang, S.M., Peischl, B.: Nachhaltiges software management durch lebenszyklusübergreifende überwachung von qualitätskennzahlen. In: Tagungsband der Fachtagung Software Management, Nachhaltiges Software Management. Deutsche Gesellschaft für Informatik (November 2012)
12. Larndorfer, S., Ramler, R., Buchwiser: Dashboards, cockpits und projekt-leitstände: Herausforderung messsysteme für die softwareentwicklung. OBJEKTSpektrum 4, 72–77 (2009)
13. Larndorfer, S., Ramler, R., Buchwiser, C.: Experiences and results from establishing a software cockpit at bmd systemhaus. In: 35th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2009, pp. 188–194 (2009)
14. Münch, J., Heidrich, J., Simon, F., Lewerentz, C., Siegmund, B., Bloch, R., Kurpicz, B., Dehn, M.: Soft-pit - ganzheitliche projekt-leitstände zur ingenieurmässigen software-projektdurchführung. In: Proceedings of the Status Conference of the German Research Program Software Engineering, vol. 70 (2006)
15. Münch, J., Heidrich, J.: Software project control centers: concepts and approaches. Journal of Systems and Software 70(1), 3–19 (2004)
16. Peischl, B., Lang, S.M.: What can we learn from in-process metrics on issue management? Testing: Academic and Industrial Conferencem Practice and Research Techniques, page IEEE Digial Library (2013)
17. Raisinghani, M.S.: Business intelligence in the digital economy: opportunities, limitations and risks. Idea Group Pub. (2004)
18. Staron, M., Meding, W., Nilsson, C.: A framework for developing measurement systems and its industrial evaluation. Information and Software Technology 51(4), 721–737 (2009)
19. Torrents, V.R.: Development and optimization of a web-enabled OLAP-based software dashboard, Master thesis, Universidad de Alcalá (2013)
20. Watson, H.J., Wixom, B.H.: The current state of business intelligence. Computer 40(9), 96–99 (2007)

# Dealing with Technical Debt in Agile Development Projects

Harry M. Sneed

ANECON GmbH, Vienna, Austria  
Fachhochschule Hagenberg, Upper Austria  
Harry.Sneed@T-Online.de

**Abstract.** Technical Debt is a term coined by Ward Cunningham to denote the amount of rework required to put a piece of software into that state which it should have had from the beginning. The term outdates the agile revolution. Technical debt can occur using any development approach, but since agile development has become wide spread the notion of “technical debt” has gained much more attention. That is because if agile development is not done properly, technical debt accrues very fast and threatens to strangle the project. In this paper the author describes how technical debt comes to being, how it can be measured and what can be done to prevent it in agile development. The emphasis of the paper is on how to prevent it. What measures are necessary in an agile development project to keep technical debt from accruing over time? The paper lists out a number of measures which can be taken – organizational, procedural and technical. Automated tools play an important role in the struggle against technical debt. Samples are given how a tool can be helpful in identifying and removing problems before they get out of hand. Also important is an external auditing agency which monitors projects in progress with the aide of automated tools. In the end a case study is presented which illustrates the monitoring of technical debt within an agile development and what counter measures are required to stop it.

**Keywords:** Technical debt, code quality, product documentation, software defects, refactoring, software metrics, agile team organization, agile testing.

## 1 Technical Debt

“*Technical Debt*“, is a term for the work required to put a piece of software in the state that it should be in. It may be that this is never done and the software remains in the substandard state forever, but still the debt remains. In other words, technical debt is a term for substandard software. It was coined to describe poor quality software in terms that business managers can relate to, namely in terms of money, money required to fix a problem [1]. Managers should become aware of the fact the neglect of software quality costs them money and that these costs can be calculated. The notion of “debt” should remind them that someday they have to pay it back or at least try to reduce it, just like a country should reduce the national debt. The size of the national debt is an indicator that a national economy is spending more than what it is

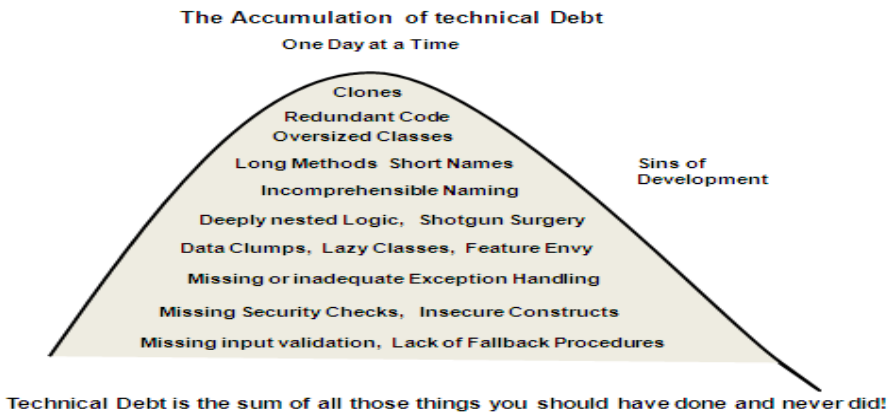
producing. It has a negative balance. The size of the technical debt is an indicator that an organization is producing more software than it can make correctly as it should be. The amount of the national debt can be measured absolutely in terms of Dollars or Euros and relatively in relation to the gross national product. The same applies to the technical debt. It too can be measured absolutely in terms of money required to renovate the software and relatively in terms of the costs of renovation relative to the development costs. Just as a country whose national debt exceeds its annual gross national product is in danger of bankruptcy, a software producer whose technical debt exceeds its annual development budget is in danger of collapsing. Something must be done to eliminate or at least to reduce the size of the debt.

The notion of “*technical debt*” was coined by Ward Cunningham at the OOPSLA conference in 1992. The original meaning as used by Cunningham was “all the not quite right code which we postpone making it right.” [2]. With this statement he was referring to the inner quality of the code. Later the term was extended to imply all that should belong to a properly developed software system, but which was purposely left out to remain in time or in budget, system features such as error handling routines, exception conditions, security checks and backup and recovery procedures and essential documents such as the architecture design, the user guide, the data model and the updated requirement specification. All of the many security and emergency features can be left out by the developer and the users will never notice it until a problem comes up. Should someone inquire about them, the developers can always say that these features were postponed to a later release. In an agile development they would say that they were put in the backlog. In his book on “Managing Software Debt - Building for Inevitable Change“, Sterling describes how this debt accumulates – one postponement at a time, one compromise after another [3].

There are many compromises made in the course of a software development. Each one is in itself not such a problem, but in sum the compromises add up to a tremendous burden on the product. The longer the missing or poorly implemented features are pushed off, the less likely it is that they will ever be added or corrected. Someone has to be responsible for the quality of the product under construction. It would be the job of the testers in the team to insist that these problems be tended to before they get out of hand. That means that the testers not only test but also inspect the code and review the documents. Missing security checks and exception handling are just as important as incorrect results. For that the testers must be in a position to recognize what is missing in the code and what has been coded poorly. This will not come out of the test. The test will only show what has been implemented but not what should have been implemented. What is not there cannot be tested. Nor will it show how the code has been implemented. Testers must have the possibility and also the ability to look into the code and see what is missing. Otherwise they would have to test each and every exception condition, security threat and incorrect data state. That would require much too much testing effort. For this reason testers should also have good knowledge of the programming language and be able to recognize what is missing in the code. Missing technical features make up a good part of the technical debt [4].

The other part of the technical debt is the poor quality of the code. This is what Ward Cunningham meant when he coined the term. In the heat of development developers make compromises in regard to the architecture and the implementation of

the code in order to get on with the job or simply because they don't know better. Instead of adding new classes they embed additional code into existing classes because that is simpler. Instead of calling new methods they nest the code deeper with more if statements, instead of using polymorphic methods, they use switch statements to distinguish between variations of the same functions, instead of thinking though the class hierarchy, they tend to clone classes thus creating redundant code. There is no end to the possibilities that developers have to ruin the code and they all too often leave none out, when they are under pressure as is the case with sprints in Scrum. In their book on "Improving the Design of existing Code", Fowler and Beck identify 22 code deficiencies, which they term as bad smells, or candidates for refactoring [5]. These smells such bad practices as duplicated code, long methods, large classes, long parameter lists, divergent change, shotgun surgery and feature envy. In a contribution to the journal of software maintenance and evolution with the title "Code Bad Smells – a Review of Current Knowledge", Zhang, Hall and Baddoo review the literature on bad smells and their effects on maintenance costs [6]. The effects of bad smells range from decreasing readability to causing an entire system to be rewritten. In any case they cause additional costs and are often the source of runtime errors which cost even more. The fact is undisputed that certain bad coding habits drive up maintenance costs. Unfortunately these costs are not obvious to the user. They do not immediately affect runtime behavior. The naïve product owner in an agile development project will not recognize what is taking place in the code. However, if nothing is done to clean up the code, the technical debt is growing from release to release (see Figure 1: accumulating technical debt)



**Fig. 1.**

It has always been known that poor coding practices and missing features cause higher maintenance costs but no one has ever calculated the exact costs. This has now been done by Bill Curtis and his colleagues from CAST Software Limited in Texas. Their suggestion or computing the costs of bad quality joins each deficiency type with the effort required to remove that deficiency. The basis for their calculation is an experience database from

which the average cost of removing each deficiency type is calculated. The data from scores of refactoring projects has been accumulated in this database [7]. The refactoring and retesting of a single overly complex method can cost a half day. Considering an hourly rate of US \$70, that leads to a total cost of \$280. The cost of a missing exception handling may cost only an hour but if five hundred such handlings are missing the cost will amount to some \$35,000. The cost of adding a missing security can cost up to three person days or \$1600. The main contribution of Curtis and his colleagues is that they have identified and classified more than a hundred software problem types and put a price tag on them. Samples of their problem types are:

- Builtin SQL queries (subject to hacking)
- Empty Catch Blocks (no exception handling)
- Overly complex conditions (error prone)
- Missing comments (detracts from comprehension)
- Redundant Code (swells the code amount and leads to maintenance errors)
- Non-uniform variable naming (making the code hard to understand)
- Loops without an emergency brake (can cause system crash)
- Too deeply nested statements (are difficult to change).

The costs of such deficiencies are the sum of the costs for each deficiency type. The cost of each deficiency type is the number of occurrences of that deficiency times the average effort required to remove and retest that deficiency as taken from the experience database. (see Figure 2: Code Deficiencies)

#### Technical Debt in Terms of Code Deficiencies

Major deficiencies sorted by number of occurrences	Correction Effort (Person Hours)	
(10) Return Value is not controlled after Method Call 2435	x 0,5	= 1217
(22) Public Variables should be avoided in Classes 2280	x 1	= 2280
(01) IO-Operations are not in a try block 913	x 1,5	= 1370
(25) Class is not derived from a Superordinate Class 219	x ?	= ?
(14) Control Logic exceeds maximum allowed nesting level 962	x 2	= 1924
(15) Missing Final clause in method declaration 856	x 0,5	= 428
(04) Data Casting should be avoided 692	x 2	= 1384
(18) Check on incoming public request is missing 635	x 4	= 2140
(26) Nested Classes are not allowed 376	x 4	= 1504
(27) Returning a Function may cause an endless loop 239	x 6	= 1434
(11) Conditions should not contain an Assignment 150	x 1	= 150
(17) Try and Catch clauses do not match 112	x 1	= 112
(13) Default is missing in Switch Statement 85	x 1	= 85
(12) Case block should contain a Break statement 77	x 2	= 154
(08) Method Invocation with Array is not in a try Block 31	x 3	= 93
(07) External Variables are not allowed 29	x 2	= 58
(06) Standard IO Functions are prohibited 21	x 4	= 84
(09) There should be no global Data Definitions in C# 5	x 2	= 10
(02) Two Dimensional Arrays violate 1. Normal Form 2	x 8	= 16
<b>Total Correction Effort</b>		<b>= 14.443</b>

Fig. 2.

To the costs of these statically recognizable deficiencies must be added the costs of removing errors which come up during operations. Due to time constraints not all errors will be fixed before release. These errors may lead later to interruptions in production if they are not corrected. Other errors may lead to false results which must be manually corrected. Bad code can lead to bad performance and that too will

become annoying to the users. The users may accept living with these inconveniences for a while but eventually they will insist that they be fixed. The cost of fixing them is not trivial. Together with the code smells and the missing features this adds up to a significant debt. Bill Curtis estimates that the median debt of an agile project is \$ 3.61 per statement [8].

That is the absolute measure for debt. It is also possible to compute the relative debt. That is the cost of renovating the software relative to the cost of development.

$$\text{Relative Debt} = \text{Renovation Cost} / \text{Development Cost}$$

It could be that the cost of renovation is almost as high as the development cost. In that case the software should be abandoned and rewritten.

## 2 Accumulating Technical Debt

In their endeavor to give the user a functioning software product in the shortest possible time, developers are inclined to take short cuts. They leave the one or the other feature out with the good intention of adding it later. A typical example is error handling. If a foreign function is called on another server, the developer should know that he will not always get an answer. It can be that the remote server is overloaded or that it is out of operation. This applies particularly to web services. For this reason, he should always include an exception handler to check the time and when a given time is exceeded to invoke an error handling routine. The same applies to accessing files and databases as well as to all external devices like printers and displays. In addition, a cautious developer will always check the returned results to make sure that they are at least plausible. The result of this defensive programming is a mass of additional code. Experts estimate that at least 50% of the code should be for handling exception conditions and erroneous states if the developer is really concerned about reliability [9].

By disregarding reliability problems the developer can save half of his effort for coding and testing, since the error handling he does not code will also not have to be tested. The temptation is very great to do this. The agile developer knows that the user representative will not notice what goes on behind the user interface. The code does not interest him. Often it will have to come to a tremendous load or to extreme situations before an exception condition is triggered. Other exceptions come up only in certain cases. Even those exceptions which do come up in test, leading to short interruptions, will not be immediately registered by the user. He is too busy dealing with other things. Therefore, if a punctual release is the overriding goal, it is all too tempting to leave the exception handling out. As John Shore, the manager of software development at the Naval Research Lab once pointed out, programmers are only poor sinners, who, under pressure, cannot withstand the temptation to cheat, especially if they know they will probably not be caught [10]. The developer has the good intention of building in the missing code later when he has more time. Unfortunately, this “later” never comes. In the end it is forgotten until one day a big crash takes place in production. The system goes down because an exception occurs and is not handled properly. The path to hell is paved with good intentions.

Tilo Linz and his coauthors also comment on this phenomena in their book on testing in Scrum projects [11]. There they note that the goal of finishing a release in a fully transparent environment with an unmovable deadline in a prescribed time box can lead to tremendous mental burden on the developers. Theoretically, according to the Scrum philosophy they should be able to adjust the scope of the functionality to their time and their abilities. In practice, this is most often not possible. Instead of cutting back on functionality, the team cuts back on quality. Discipline is sacrificed to the benefit of velocity. The functionality is not properly specified, test cases that should be automated are executed manually, necessary refactoring measures are pushed off. The software degrades to a point where it can no longer be evolved. The team is sucked under by a downwards spiral and in the end the project is discarded.

The situation with security is similar. Secure code is code that protects itself against all potential intrusions. But to achieve that means more code. A secure method or function controls all incoming calls before it accepts them. Every parameter value should be checked to make sure it is not corrupt. The developer should also confirm that the numbers and types of arguments matches that what is expected, so that no additional parameters are added to the list. These could be hooks to get access to internal data. The values of the incoming parameters should be checked against predefined value ranges. This is to prevent corrupted data from getting into the software component. To make sure that only authorized clients are calling, the function called can require an identification key to each call. In Java the incoming objects should be cloned, otherwise it can be mutated by the caller to exploit race conditions in the method. Methods which are declared public for testing purposes should later be changed to private or protected to restrict access rights, and classes should be finalized when they are finished to protect their byte code from being overwritten [12]. Embedded SQL statements are particularly vulnerable to attack since they can be easily manipulated at runtime. They should be outsourced into a separate access shell, but that entails having extra classes which also have to be tested [13]. Thus, there are many things a developer can do to make his code more secure, if he takes the time to do it. But here too, the developer can rest assure that nobody will notice the missing security checks unless they make an explicit security test. So security is something that the developer can easily postpone. The users will only become aware of the problem when one day their customer data is stolen. Then it will be too late.

That is the problem with software. Users cannot easily distinguish between prototypes and products. A piece of software appears to be complete and functioning, even when it is not. As long as the user avoids the pitfalls everything seems to be ok. They have no way of perceiving what is missing. For that they would either have to test every possible usage or they have to dig into the code. The question of whether a piece of software is done or not, cannot be answered by the user, since he has no clue of what “done” really means. The many potential dangers lurking in the code cannot be recognized by a naive user, therefore it is absurd to expect that from him. He needs an agent to act in his behalf who can tell him if the software is really done or not. This agent is the tester, or testers, in the development team [14].

### 3 Role of Automated Static Analysis in Detecting Technical Debt

Static analysis is one of many approaches aimed at detecting defects in software. It is a process of evaluating a system or component based on its form, structure, content or documentation [15] which does not require program execution. Code inspections are one example of a classic static analysis technique relying on manual examination of the code to detect errors, violations of coding conventions and other potential problems. Another type of static analysis is the use of automated tools to identify anomalies in the code such as non-compliance with coding standards, uncaught runtime exceptions, redundant code, incorrect use of variables, division by zero and potential memory leaks. This is referred to as automated static analysis.

There are now many tools available to support automated static analysis – tools like Software Sonar, Conformity and SoftAudit. In selecting a proper tool, the main question is what types of deficiencies are reported by the tool. Another question has to do with the number of false positives the tool produces that is how many defects or deficiencies are by closer examination not really true. When tools produce a great number of false positives, this only distracts from locating the real problems. As to the types of true problems reported, it is important to have a problem classification schema. From the viewpoint of technical debt there are three major classes of code problems:

- missing code = statements that should be there and or not
- redundant code = statements that are there and should not be
- erroneous code = statements that may cause an error when the code is executed
- non-conform code = statements which violate a local standard or coding convention
- deficient code = statements that reduce the overall quality of the code, maintainability, portability, testability, etc.

Samples of missing code are:

- a missing break in a switch statement
- a missing try clause in the call of a foreign method
- a missing check of return value after a function call

Samples of redundant code are:

- cloned codes, i.e. code sections which are almost the same except for minor variances
- variables declared but not used
- methods implemented but never called
- dead code or statements which can never be reached



Samples of erroneous code are:

- missing loop termination conditions
- pointers or object references that are used before they are set, i.e. null-pointers
- variables that are used before they are initialized
- casting of variables, i.e. changing the data type at run time.

Samples of non-conform code are:

- missing prescribed comments
- data names which do not adhere to the naming convention
- procedures which are too long or too complex, i.e. they exceed the maximum number of statements or the maximum nesting level
- statements which exceed the maximum line length

Samples of deficient code:

- nested statements which are not intended
- conditions which are too complex
- non portable statements, i.e. statements which may work in one environment but not in another
- statements which deduct from the readability such as expressions in conditions

Also many of the security threats in the code can be detected through automated static analysis. *Jslint* from Citital is representative of tools which perform such a security analysis. It enforces 12 rules for secure Java code. These rules are:

- 1) Guard against the allocation of uninitialized objects
- 2) Limit access to classes and their members
- 3) Declare all classes and methods as final
- 4) Prevent new classes from being added to finished packages
- 5) Forbid the nesting of classes
- 6) Avoid the signing of code
- 7) Put all of the code in one jar or archive file
- 8) Define the classes as uncloneable
- 9) Make the classes unserializable
- 10) Make the classes undeserializable
- 11) Forbid company classes by name
- 12) Avoid the hardwiring of sensitive data in the code

*Jslint* scans the code for infringement of these rules. All of the rule violations can be recognized except for the last one. Here it is not possible to distinguish between sensitive and non-sensitive data. Therefore it is better not to use hard-wired text in the code at all. If it is, then it can be easily recognized. In some cases *Jslint* can even correct the code, for instance by overwriting the access class, by adding the final

clause and removing code signatures. The automated correction of code can however be dangerous and cause undesired side effects. Therefore, it is better to use automated tools to detect the potential problems and to manually eliminate them [16].

A study was performed at the North Carolina State University on over 3 million lines of C++ Code from the Nortel Networks Corporation to demonstrate how effective automated static analysis can be. Using the tool “FlexLint” the researchers there were able to detect more than 136,000 problems or one problem for every 22 code statements. Even though this code had been in operation for over three years there remained a significant technical debt. The problems uncovered by the automated static analysis also correlated well to the defects discovered during the prerelease testing with an accuracy of 83%. That implies that 83% of the erroneous modules could have been identified already through static analysis.

Of the defects uncovered in the static analysis, 20% were critical, 39% were major and 41% were minor. The coding standard violations were not included in this count. Based on their findings, the authors of this study came to the following conclusions:

- the defect detection rate of automated static analysis is not significantly different than that of manual inspections
- the defect density recorded by automatic static analysis has a high correlation to the defect density recorded in prerelease testing
- the costs of automated static analysis is a magnitude less – under 10% - of the costs occurred by manual inspection and prerelease testing

Thus, automated static analysis can be considered an effective and economic means of detecting problematic code modules [17].

One of the areas addressed by the static analysis study at North Carolina State was that of security vulnerabilities. The authors of the study paid particular attention to code constructs that have the potential to cause security vulnerabilities if proper protection mechanisms are not in place. Altogether 10 security vulnerabilities were identified by the automated analysis tool:

- Use of Null pointers
- Access out of bound references, i.e. potential buffer overflow
- Suspicious use of malformed data
- Type mismatch in which statements
- Passing a null pointer to another function
- Failure to check return results
- Division by zero
- Null pointer dereference
- Unrecognizable formats due to malformed data
- Wrong output messages

The number of false positives in the check of these potential security violations was higher than average, however the results indicate that automated static analysis can also be used to find coding deficiencies that have the potential to cause security vulnerabilities. The conclusion of this research is that static analysis is the best instrument for detecting and measuring technical debt [18].

## 4 Early Recognition of Mounting Technical Debt

The main contribution of agile testing is to establish a rapid feedback test to development. This is the rationale for having the testers in the team. When it comes to preventing quality degradation and increasing debt, the testers in an agile project have more to do than just test. They are there to assure the quality of the product at all levels during construction. This should be accomplished through a number of control measures, measures such as reviewing the stories, transforming the stories into testable specifications, reviewing the architectural design, inspecting the code, validating the unit tests, performing continuous integration testing and running an acceptance test with the user. On top of that, there still have to be special performance, load and security tests, as well as the usability and reliability tests. If the product under construction is not just a trivial temporary solution, then those special tests should be made by a professional testing team, which works independently of the agile developing teams. This solution is referred to by Linz and his coauthors as “System Test Team” whose job it is to shake down the systems delivered by the Scrum teams independently of the schedule imposed on the project. That means that the releases of the agile development team are really only prototypes and should be treated as such. The final product comes only after the prototypes have gone through all of the special tests. To these special tests belongs also the final auditing of the code [19]. (see Figure 3: The role of the Tester)

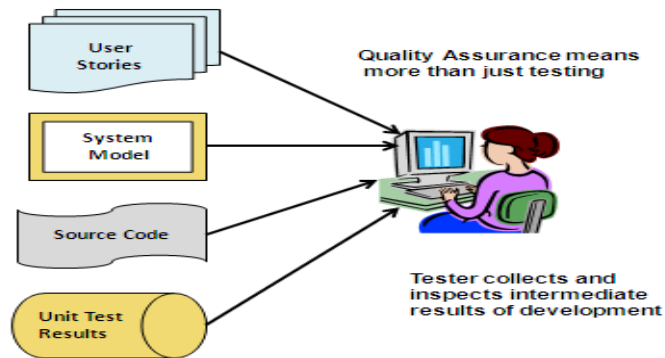


Fig. 3.

The review of the stories is concerned with discussing and analyzing the stories with the view of enhancing and improving them. They should also be checked for testability. The user representative might overlook something or fail to cover all describe a function adequately. It is up to the testers in the team to point this out and to clear the missing and unclear issues with him. Special attention should be paid by the testers to the non-functional aspects of the stories like security and usability. In any case the stories need to be purified before the developers start to implement them. The testers need stories they can interpret and test.

In order to establish a testing baseline, the testers should convert the informally defined user stories into an at least semi-formal requirement specification. As noted

above a significant portion of the technical debt is caused by missing functions, but who is to know that they are missing when they are not specified anywhere. All of the exception conditions, security checks and other quality assuring functions of a system should be noted down as non-functional requirements or as business rules. They should be specified in such a way as to be recognizable, for instance with key words and to be convertible into test cases. For every business rule and non-functional requirement at least one test case should be generated to ensure that the possible exceptions and security threats are tested. In this way the requirements can be used as an oracle for measuring test coverage [20].

In checking the code for conformity with the coding rules testers should also check the format and the readability. Most all of the missing exception handling, security measures and data validation checks can be recognized in the code. This is the purpose of an automated code audit. It only takes a few minutes of the tester's time to set up and run such an audit. More time is needed to explain the reported deficiencies to the developers and to convince them that they should clean them up. Of course the tester cannot force the developers to do anything, but he can post the deficiencies on the project white board and try to convince the team members that it is to their benefit to have these coding issues settled, if not in the next release then sometime in a future release. In no case should he let the team forget them. A good way of reminding the team is to post a chart on the amount of technical debt accrued in the project room for everyone to see, similar to the clock in Times Square which reminds Americans constantly of the current state of their national debt. Just as in New York where many American citizens simply shrug their shoulders and say "so what", project members who could care less about the final quality of their product will ignore the issue of technical debt, but at least it will remind them of their sins. .

By controlling the results of the unit tests, the testers can point out to the developers what they have missed and what they can do to improve their test. In checking through the trace and coverage reports, the tester can identify what functions have not been adequately tested and can suggest to the developer additional test cases required to traverse those hidden corners of the code. Here too, he must convince the developer that it is to his benefit to filter out the defects as early as possible. Pushing them off to the acceptance test only increases the cost of correcting them. Every defect removed in unit testing is one defect less to deal with in integration and acceptance testing [21].

The testers in an agile development team must strive to be only one or maximal two days behind the developers [22]. No later than two days after turning over a component or class, the developer should know what he has done wrong. "Continuous Integration" makes this possible [23]. The tester should maintain an automated integration test frame into which he can insert the new components. The existing components will already be there. Every day new or corrected components are taken over into the build. In the case of new functions, the test scripts and test data have to be extended. Not only will the new functions be tested, but the test of the old functions will be repeated. The setting up of such an automated regression test may take days, but after that the execution of each additional test should take only few hours. The goal is to keep a steady flow of testing. The actual data results should be

constantly compared with the specified results by means of an automated data comparator. Should new input data be required, it should be generated by an automated test data generator. The two tools should be joined by a common test language which ensures that the test outputs match with the test inputs. It is imperative that all deviations from the expected behavior of the software be automatically registered and documented. It should go so far that the defect reports are also automatically generated. The tester should only have to enhance them. In agile development time is of essence and anything which can save time should be introduced (see Figure 4: Testing in an agile Development Project).

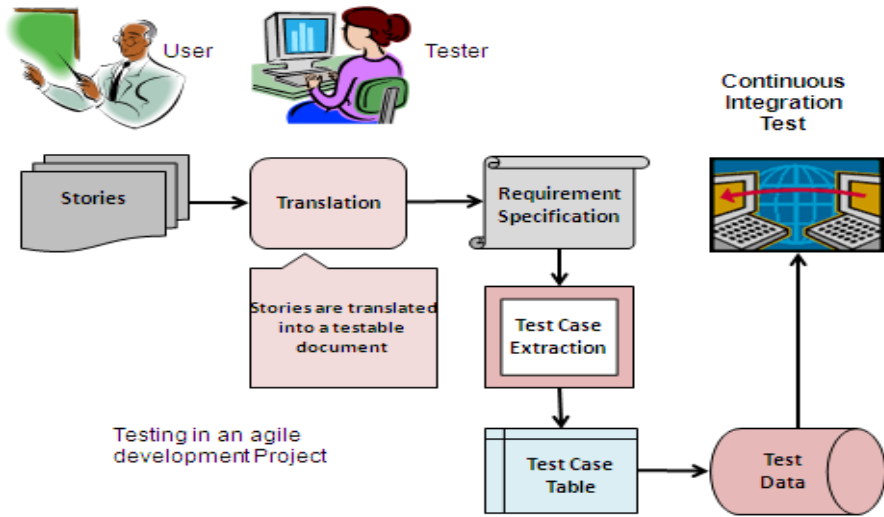


Fig. 4.

The main goal of agile development is rapid feedback. Problems should be reported back immediately to the developer to be resolved before they become acute. This is also the great advantage of agile development vis-a-vis classical phase-oriented software development where it often took weeks if not months before the problems were detected. In the meantime, the developers assumed that all was honky dory and went on developing new erroneous components or, in the worst case, they did nothing and just waited on the feedback from the testers. In this way many valuable hours of tester time were lost. Thus, it is truly beneficial to have component integration testing running parallel to component development [24].

The early recognition of defects and the quick non-bureaucratic handling of defect reports is one of the main advantages of the agile development process over the classic waterfall process [25]. To function well the testers should be in constant contact with the developers. Whether this contact must be physical is disputed. The authors of the agile manifesto put great emphasis on "face to face" communication. Many proponents of agile development agree with that and insist that the testers be physically together with the developers. Others argue that it is enough for the testers

to be in virtual contact with the developers. Physically they might be in another continent. The proponents of the virtual team claim that it suffices to communicate daily with a social networking system and video conferences via the internet. Whether this really is feasible or not remains to be seen. For the moment there remains two schools of thought [26].

## 5 What Is “Done”?

At the Belgian “Testing Days” conference in 2012 Johanna Rothman und Lisa Crispin, two recognized experts on agile testing, discussed what is meant by “done” [27]. In the opinion of Rothman this is something that must be decided upon by the team as a whole. It is, however, up to the testers to trigger the discussion and to lead it a conclusion. The testers should feed the discussion with their experience from previous projects and to steer it in a positive direction. Rothman proclaims “you have to get the team thinking about what is done. Does it mean partially done, as in it is ready for testing or fully done, as in it is ready for release?” A minimum quality line is required in order to make an intermediate release. If the users know they are working with a prototype they are more likely to be error tolerant. If they know, this release will be the final product they will be stricter. In the end the users must decide what is done. But, since users only have a restricted view of the system, they need others to help them in passing final judgment on the state of the software. Those others are the testers who are prepared to assess the system on behalf of the potential users. It is all too easy, just to declare a product as being “done”. The lead tester should, in any case, be involved in the discussion. The decision as to whether something is “done” or not is, in the end, a political decision, which must be considered from several points of view.

Developers tend to become impatient and to push the product thru in any case, even if it is falling apart. They will always argue that the quality is sufficient. Testers will, on the other hand, claim that the quality level is not high enough. They will argue for more time to test. Otherwise the quality problems will only be pushed off on the maintenance team. Rothman suggests using the KANBAN progress charts to show the current quality state of each and every component as well as the product under development. This way everyone in the project can see where they are, relative to the goals they have set for themselves. In effect, the project should produce two quality reports, one on the functionality and one on the quality. (see Figure 5: Measuring Technical Debt).

The functional state of a project is easier to assess than the qualitative state. It is visible whether a function is performing or not. The quality state is not so visible. To see what security and data checks are missing, the tester has to dig into the code. To see how many rules are violated he has to have the code audited. There is no easy way to determine how many defects remain. This can only be known after every last function has been tested for all variations. That is why, one must often guess when deciding on the state of product quality. Perhaps the best indicators of product quality are the number of code deficiencies relative to the number of code statements and the number of defects detected so far relative to the test coverage. For both metrics there are benchmark levels which the testers can discuss with the user and the other team members [28].

**Measuring Technical Debt  
relative to Development Costs**

Accumulated debt should be reduced from Release to Release

Component	1. Release	2. Release	3. Release	4. Release	5. Release
Client-GUI	0,30	0,20	0,15	0,10	0,05
Input-Validation	0,25	0,15	0,10	0,05	0,03
Output-marshall	0,20	0,20	0,15	0,10	0,05
Database-Access	0,15	0,15	0,10	0,05	0,01
Total Debt	0,225	0,175	0,125	0,075	0,03

**Debt = (Renovation Costs / Construction Costs)**

**Fig. 5.**

Johanna Rothman argues that testers must be involved in the discussion as to what is “done“ from the beginning of the project on. ”To be done also means that the quality criteria set by the team are met“. For that this criteria has to be accepted and practiced by all team members. Everyone in the team must be fully aware of his or her responsibility for quality and must be dedicated to improving it. Rothman summarizes “Everybody in the team needs to take responsibility for quality and for keeping technical debt at a manageable level. The whole team has to make a meaningful commitment to quality“. It is true that quality is a team responsibility but the testers have a particular role to play. They should keep track of the technical debt and keep it visible to the other team members [29].

Lisa Crispin points out that software quality is the ultimate measure of the success of agile development [30]. Functional progress should not be made by sacrificing quality. After each release, i.e. every 2 to 4 weeks, the quality should be assessed. If it falls below the quality line, then there has to be a special release devoted strictly to the improvement of quality, even if it means putting important functions on ice. In this release the code is refactored and the defects and deficiencies removed. Crispin even suggests having a separate quality assurance team working parallel to the development team to monitor the quality of the software and to report to the developers. This would mean going back to the old division of labor between development and test, and to the problems associated with that. The fact is that there is no one way of ensuring quality and preventing the accumulation of technical debt. Those responsible for software development projects must decide on a case by case basis which way to go. The right solution is as always context dependent.

## References

1. Kruchten, P., Nord, R.: Technical Debt – from Metaphor to Theory and Practice. IEEE Software, S.18 (December 2012)
2. Cunningham, W.: The Wgcash Portfolio Management System. In: Proc. of ACM Object-Oriented Programming Systems, Languages and Applications, OOPSLA, New Orleans, p. S.29 (1992)

3. Sterling, C.: *Managing Software Debt – Building for inevitable Change*. Addison-Wesley (2011)
4. Lim, E., Taksande, N., Seaman, C.: *A Balancing Act – What Practitioners say about Technical Debt*. *IEEE Software*, S.22 (December 2012)
5. Fowler, M., Beck, K.: *Improving the Design of existing Code*. Addison-Wesley, Boston (2011)
6. Zhang, M., Hall, T., Baddoo, M.: *Code Smells – A Review of Current Knowledge*. *Journal of Software Maintenance and Evolution* 23(3), 179 (2011)
7. Curtis, B., Sappidi, J., Szykarski, A.: *Estimating the Principle of an Application’s Technical Debt*. *IEEE Software*, S. 34 (December 2012)
8. Wendehost, T.: *Der Source Code birgt eine Kostenfalle*. *Computerwoche* 10, S.34 (2013)
9. Nakajo, T.: *A Case History Analysis of Software Error Cause and Effect Relationships*. *IEEE Trans. on S.E.* 17(8), S.830 (1991)
10. Shore, J.: *Why I never met a Programmer that I could trust*. *ACM Software Engineering Notes* 5(1) (January 1979)
11. Linz, T.A.O.: *Testing in Scrum Projects*, p. 177. Dpunkt Verlag, Heidelberg
12. Lai, C.: *Java Insecurity – accounting for Sublities that can compromise Code*. *IEEE Software Magazine* 13 (January 2008)
13. Merlo, E., Letrte, D., Antonioli, G.: *Automated Protection of PHP Applications against SQL Injection Attacks*. In: *IEEE Proc. 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, Amsterdam, p. 191 (March 2007)
14. Crispin, L., Gregory, J.: *Agile Testing – A practical Guide for Testers and agile Teams*. Addison-Wesley-Longman, Amsterdam (2009)
15. Ayewah, N., Pugh, W.: *Using Static Analysis to find Bugs*. *IEEE Software Magazine*, 22 (September 2008)
16. Vieg, J., McGraw, G., Felten, E.: *Statically Scanning Java Code – Finding Security Vulnerabilities*. *IEEE Software Magazine*, 68 (September 2000)
17. Zheng, J., Williams, L., Nagappan, N., Snipes, W.: *On the Value of Static Analysis for Fault Detection in Software*. *IEEE Trans. on S.E.* 32(4), 240 (2006)
18. Gueheneu, Y.-G., Moha, N., Duchien, L.: *DÉCOR – A Method for the Detection of Code and Design Smells*. *IEEE Trans. on S.E* 36(1), 20 (2010)
19. Linz, T.A.O.: *Testing in Scrum Projects*, p. 179. Dpunkt Verlag, Heidelberg
20. Sneed, H., Baumgartner, M.: *Value-driven Testing – The economics of software testing*. In: *Proc. of 7th Conquest Conference, Nürnberg*, p. 17 (September 2009)
21. Marre, M., Bertolini, A.: *Using Spanning Sets for Coverage Measurement*. *IEEE Trns. on S.E.* 29(11), 974 (2003)
22. Bloch, U.: *Wenn Integration mit Agilität nicht Schritt hält*. *Computerwoche* 24, S. 22 (2011)
23. Duvall, P., Matyas, S., Glover, A.: *Continuous Integration – Improving Software Quality and reducing Risk*. Addison-Wesley, Reading (2007)
24. Humble, J., Fairley, D.: *Continuous Delivery*. Addison-Wesley, Boston (2011)
25. Cockburn, A.: *Agile Software Development*. Addison-Wesley, Reading (2002)
26. Bavani, R.: *Distributed Agile Testing and Technical Debt*. *IEEE Software*, S.28 (December 2012)
27. Rothman, J.: *Do you know what fracture-it is?* Blog (June 2, 2011), <http://www.jrothman.com>
28. Letouzey, J.L., Ilkiewicz, M.: *Managing Technical Debt*. *IEEE Software Magazine*, 44 (December 2012)
29. Rothman, J.: *Buy Now, Pay Later*. In: *Proc. of Belgium Testing Days, Brussels* (March 2012)
30. Crispin, L., House, T.: *Testing Extreme Programming*. Addison-Wesley-Longmann, Reading (2002)



# Statistical Analysis of Requirements Prioritization for Transition to Web Technologies: A Case Study in an Electric Power Organization

Panagiota Chatzipetrou, Christos Karapiperis, Chrysa Palampouiki,  
and Lefteris Angelis

Department of Informatics, Aristotle University of Thessaloniki, Greece  
{pchatzip,c.karapiperis,chrisapalam}@gmail.com  
lef@csd.auth.gr

**Abstract.** Transition from an existing IT system to modern Web technologies provides multiple benefits to an organization and its customers. Such a transition in a large organization involves various groups of stakeholders who may prioritize differently the requirements of the software under development. In our case study, the organization is a leading domestic company in the field of electricity power. The existing online system supports the customer service along with the technical activities and has more than 1,500 registered users, while simultaneous access can be reached by 300 users. The paper presents an empirical study where 51 employees in different roles prioritize 18 software requirements using hierarchical cumulative voting. The goal of this study is to test significant differences in prioritization between groups of stakeholders. Statistical methods involving data transformation, ANOVA and Discriminant Analysis were applied to data. The results showed significant differences between roles of the stakeholders in certain requirements.

**Keywords:** Requirement prioritization, Software quality, Hierarchical Cumulative Voting, Statistical Analysis.

## 1 Introduction

The paper presents a quantitative case study related to requirements prioritization by various stakeholders in different roles. Although the study concerns a specific organization, the problem of transition of an existing critical system to modern Web technologies is general and quite common, while the research questions and the methodologies applied in the paper can be used to address a wide range of similar problems in other domains. In the following subsections we provide a detailed introduction to the background and the objectives of the paper, we state the research questions and we briefly mention the methods followed (these are detailed in later sections).

### 1.1 Background

Nowadays, the energy environment is rapidly changing. Even for monopoly activities, like transmission and distribution of electric power, an Information Technology (IT)

infrastructure is a competitive advantage in order to improve the organization efficiency. Because of the financial crisis, it is the right time to reduce costs and improve the official operational processes through the exploitation of the benefits that technology has to offer. The authors in [1] refer to the role of technology and in particular, to the several problems associated with the use of technological achievements. These problems are well known, however they continue to handicap the accomplishments of expected returns from technology investments in organizations.

IT projects which involve modernization of existing systems, aim to optimize the functions of organizations and the quality of services provided to customers. Of course, transition to new Web technologies has several problems which may have various sources like financial, technical, operational etc. Since the final result of such projects concerns a large number of people, and in a way affects their everyday professional life and performance inside the organization, several problems are related to the different and often conflicting opinions regarding the requirements of the final product. Transition to new technologies has various peculiarities, since the project essentially aims to substitute an existing system which has been used for years by a large number of employees. In general, employees are too familiar with certain functionalities and options offered by the old system and they are often reluctant to accept changes that may affect their everyday way of work and can cause delays and mistakes.

## 1.2 Objectives of the Case Study

Considering all the aforementioned issues, it appears necessary to take into account the opinions of stakeholders and to understand how they perceive the requirements of a project leading to an upgrade of the current system through transition to Web technologies. Since opinions on requirements are hardly analyzed when expressed in natural language, it is imperative to have them coded explicitly, and moreover, to study them systematically with sound scientific methodologies. In this regard, the research area of requirements prioritization has recently offered tools and methodologies towards the systematic recording and study of how different people in different roles perceive and prioritize various requirements.

This paper is the result of our involvement in such a project, where the existing system of a large organization is going to be replaced by a new one, based on Web technologies. The idea of the paper started from discussions with various employees in the organization, who have stated different opinions on certain functions and qualities the new system should have. These discussions led us to consider the possibility of recording a number of key requirements and then to use statistical analysis in order to study in depth the prioritization attitudes of individuals in different roles inside the organization. The statistical analysis can give numerous interesting results. In this paper we specifically investigate how different roles affect the prioritization perspective of stakeholders.

The case study of this paper concerns a domestic organization, leading in the field of electricity power. The company currently uses an online system covering the customer service and the technical support of the electric power network. The IT system has more than 1,500 registered users, with requirement for simultaneous access (concurrency) of 300 users on average. The current system is critical for the functionalities of the organization related to response rates and quality of services in order to address customers' needs.

The new ongoing project, aims to convert the front end application to modern Web technologies for the parts of the application related to customer service and network operation. The modernization of application architecture to advanced Web technologies facilitates system's integration with other available systems. Such a transition was decided on the basis of a benefit-cost analysis as especially advantageous for the organization and the customers, due to the reasonable cost, the low risk of implementation and also the easiness of training.

This improvement in IT processes optimizes the planning of the work effort and improves the system quality, the quality of services and the user satisfaction. The ultimate goal of the company is to proceed in enterprise system integration as soon as possible. Another advantage of this project is the fact that it is of medium scale and a small, skillful and flexible team is capable to complete it in scheduled time. Furthermore, the generated knowledge remains in house and the company retains its independence from external partners specialized in IT systems.

The main goals for the new deployment are:

- High quality services;
- reduction of process complexity;
- improvement of the communication between departments;
- reduction of human effort;
- improvement of data storage and management;
- minimization of failure rates due to overtime;
- interoperability;
- scalability.

To reach these goals, the deployment is based on the existing IT infrastructure, using a carefully designed enterprise system integration plan. As we already mentioned, such a transition involves various groups of employees with different relations with the company. Discussions with these stakeholders indicated that prioritizations of requirements regarding the software under development and the services supported are different. However, the origins of different views are not always clear. Divergences may originate from their different responsibilities and authorities, the working environment, even from individual cultural, educational and personality characteristics. On the other hand, decision makers have to make strategic decisions based not only on technical specifications, but also on the need to achieve the maximum possible stakeholders' satisfaction.

### **1.3 Research Questions and Methods**

The above objectives, apart from their practical importance, raised a number of research questions which led us to conduct the current study. The basic motivation of the current research was the need to provide to IT decision makers a sound methodology which can facilitate them to explore the various opinions about prioritization of requirements and understand the origins of divergent prioritizations. In our previous research work on similar problems, we employed systematic ways for recording the various prioritization views and to discover agreements or trends that can provide useful information for decision making. In this paper we apply and extend these methodologies to the current situation.

The specific research questions posed in this paper were related to the study of prioritizations of stakeholders in different roles. Hence, the first question was:

*RQ1: How can we capture in a quantitative form, appropriate for statistical analysis the requirements prioritization of stakeholders?*

For this purpose, we used the methodology of Hierarchical Cumulative Voting (HCV) in order to elicit the priorities recorded by four different categories of employees: senior executives, executives (local managers), software developer and users (company employees with continuous interaction with the system). HCV essentially quantifies the opinions of stakeholders in a meaningful and easy-to-analyze manner. Prioritization data were collected from 51 stakeholders and were statistically analyzed by univariate and multivariate statistical methodologies (like ANOVA and Discriminant Analysis) in order to address the following research questions:

*RQ2: Are there significant differences in the prioritization of requirements between the different groups of stakeholders?*

*RQ3: Is there a combination of requirements which are able to discriminate the various groups of stakeholders according to their prioritization attitude?*

The findings are interesting since they revealed that prioritization attitudes are able to discriminate major groups of stakeholders. Furthermore, the overall approach followed in the specific case study can be applied as methodology to similar situations.

The rest of the paper is structured as follows: Section 2 provides an outline of the related work. Section 3 describes the survey procedure and the resulted dataset. Section 4 presents the basic principles of the statistical tools used in the analysis. Section 5 presents the results of statistical analysis. In Section 6 we conclude by discussing the findings of the paper, the threats to the validity of the study and we provide directions for future work.

## **2 Related Work and Contribution**

Requirement prioritization is a crucial field in the development of information systems. It is also an emerging field of research since there are various ways to record the opinions of individuals involved in the development process and the use of the system. Since our study is empirical, i.e. is based on data, we focus on methods for collecting data from surveys and analyzing them with statistical techniques.

In [2] the results of a user survey were used to identify key quality characteristics and sub-characteristics of intranet applications. Another survey [3] was used for the prioritization of quality requirements that commonly affect software architecture with respect to cost and lead-time impact. The analysis concerned differences between the importance of requirements and between the views of different stakeholders. J. Offut [4] discusses various aspects of quality requirements for Web applications.

The value of data collected through surveys is clearly portrayed in recent studies. In [5] an interview study regarding quality requirements in industrial practice was performed. Data were collected through interviews with product managers and project leaders from different software companies. Quality requirements were analyzed with

respect to the type of companies, the roles of managers and their interdependencies. In [6] an online survey was used for collecting quantitative and qualitative data regarding quality requirements in Australian organizations.

The data of the present paper were obtained by the Hierarchical Cumulative Voting (HCV) method [7], [8]. HCV is a very effective way to elicit the priorities given to the various requirements of software quality. The methodology is essentially an extension of the Cumulative Voting (CV) procedure on two levels. Each stakeholder first uses CV to prioritize issues of the upper level and then uses the same technique in the lower level. The original CV method (also known as hundred-dollar test) is described in [9]. CV is a simple, straightforward and intuitively appealing voting scheme where each stakeholder is given a fixed amount (e.g. 100, 1000 or 10000) of imaginary units (for example monetary) that can be used for voting in favor of the most important items (issues, aspects, requirements etc). In this way, the amount of money assigned to an item represents the respondent's relative preference (and therefore prioritization) in relation to the other items. The imaginary money can be distributed according to the stakeholder's opinion without restrictions. Each stakeholder is free to put the whole amount on only one item of dominating importance or to distribute equally the amount to several or even to all of the issues.

In software engineering, CV and HCV have been used as prioritization tools in various areas such as requirements engineering, impact analysis or process improvement ([10], [11], [12], [13]). Prioritization is performed by stakeholders (users, developers, consultants, marketing representatives or customers), under different perspectives or positions, who respond in questionnaires appropriately designed.

Recently, HCV was used in an empirical study [14] undertaken in a global software development (GSD) of a software product. The 65 stakeholders, in different roles with the company, prioritized 24 software quality aspects using HCV in two levels.

The multivariate data obtained from CV or HCV are of a special type, since the values of all variables have a constant sum. An exploratory methodology for studying data obtained from CV-based methods is Compositional Data Analysis (CoDA). CoDA has been widely used in the methodological analysis of materials composition in various scientific fields like chemistry, geology and archaeology [15].

Authors of the present paper in a former study [16] derived data concerning prioritization of requirements and issues from questionnaires using the Cumulative Voting (CV) method and used CoDA to analyze them. Compositional Data Analysis (CoDA) has been used also for studying different types of data. For example, in [17] the authors used CoDA for obtaining useful information on allocated software effort within project phases and in relation to external or internal project characteristics. CoDA apart from being a method of analysis provides useful tools for transformation of the original data and further analysis by other methods such as cluster analysis [14].

Finally, a recent paper [18], provides a systematic review of issues related to CV and a method for detecting prioritization items with equal priority.

In the present paper we used the experience from our former studies on requirements prioritization in order to collect data from the organization under study and then to apply statistical techniques in order to study divergences between different types of stakeholders in prioritization. The data were collected using a 2-level HCV procedure and then we applied specific CoDA techniques for transforming the data. The transformed data were subsequently analyzed with univariate and multivariate statistical methodologies in order

to test the significance of differences in prioritization between the groups of stakeholders. Therefore the contribution of the paper is twofold: the type of analysis we use here is novel regarding the methods applied to the specific type of data while the case study, which refers to a real situation of a large electricity power organization which modernizes its IT system can be considered as a novel application domain.

### 3 Description of the Data Set

The case study presented in this paper is based on data obtained from a group of employees of a leading company in the field of electric power. The employees who belong to different roles within the company were asked to prioritize different issues of the new system. All issues concern the transition from an existing IT system to a new one. All the employees have been working for long time in the organization and they know very well the functionalities of the system since their interaction with the system is everyday and continuous.

The method of selecting these individuals was not random; it can be described as “convenience sampling” [19], [20]. That means that due to confidentiality issues and the sensitivity of the data, only employees that were accessible by the interviewer were approached. The respondents filled a spreadsheet that was developed for that reason, preventing the respondent to fill invalid numbers or numbers that were not summed up to 100. The spreadsheet was given to respondents during person-to-person interviews where the interviewer could provide explanations and answer clarifying questions regarding the procedure and the requirements that should be prioritized. The whole study was performed and completed during December of 2012 and January of 2013.

There are two different perspectives that can describe the term “stakeholder”. From the strategic management point of view “a stakeholder in an organization is (by definition) any group or individual who can affect or is affected by the achievement of the organization’s objectives.” [21]. On the other hand, from the software engineering point of view “system stakeholders are people or organizations who will be affected by the system and who have a direct or indirect influence on the system requirements” [22]. There are several paradigms for approaching stakeholder's identification process [23] [24]. Our approach based on Eason [25] which categorized the system users to primary, secondary and tertiary. Moreover, Sharp et al. [26] identified four different stakeholders’ categories: users, developers, legislators and decision makers.

The stakeholders’ groups who responded and filled the requirement prioritization questionnaire are classified in 4 categories:

*Senior Executives:* Their opinion reflects the standpoint of administration (the Board).

*Executives:* These are local frontline managers and their experience of the system is significant.

*Software Developers:* Employees who have the responsibility to develop the software of the new application

*Users:* Employees who have daily working contact with the present system and their opinion is of high importance.

The distribution of the stakeholders within the inherent identified grouping is shown in Table 1.

Each stakeholder was asked to prioritize 18 requirements which were decided after long interaction with members of the organization. It is important to emphasize that the authors of this paper are actively involved in the procedures of this project so the defined requirements are realistic demands expressed by the personnel. Furthermore, it is necessary to highlight that the requirements examined here are not all quality requirements. Some of them can be considered as functional requirements while others are simply requirements for properties that were considered practically important for the transition to Web technologies of the specific organization.

**Table 1.** Groups of Stakeholders

Role	Total	Percent (%)
Senior Executives	3	5,9%
Executives	9	17,6%
Software Developers	11	21,6%
Users	28	54,9%
<b>TOTAL</b>	<b>51</b>	<b>100%</b>

The requirements were prioritized using Hierarchical Cumulative Voting (HCV) in two levels. The higher level consists of four requirements while each one of them contains a number of requirements in the lower level. Specifically, the four higher level requirements are:

- A. *Features*, which are related to the general profile of the system (contains 5 requirements of the lower level),
- B. *System Properties*, which reflect the desired practical extensions of the new developed system (contains 5 requirements of the lower level),
- C. *Project Management*, which is related to the software development process of the new system (contains 4 requirements of the lower level),
- D. *New Technology Adaptation*, which is related to the application of the new technologies (contains 4 requirements of the lower level).

As already mentioned, the lower level consists of 18 requirements. Specifically, their description is:

- A1. *Usability*, which is related to the effort needed for use and learning the new system,
- A2. *(uninterrupted) Operability*, is related to the degree the system can be used (without interruptions due to technical reasons),
- A3. *System Response Time*, is related to the response time of the new system and to the avoidance of delays due to cumbersome procedures or to network delays,
- A4. *Data Security*, which is referred to the protection of sensitive data from malicious attacks,
- A5. *Adaptability*, which is related to the system's ability of adjusting to the new business processes.
- B1. *Ability of Statistical Services*, which is related with the ability of the system to automatically issue statistics (statistics reports, KPIs),
- B2. *Configurability*, which refers to the potential configuration of some features of the system by the user (e.g. customization of the main menu),
- B3. *Publishing-printing Forms/Limiting Bureaucracy*, which is related to publishing documents and official forms,

- B4. *Procedures Implementation*, which refers to the full integration of all business processes within the system,
- B5. *Prevention of duplicate data entries (systems' interoperability)*, which refers to the reduction of the primary data entry due to the interoperability with other IT systems.
- C1. *Time*, which refers to the time that will be needed to develop the system software,
- C2. *Costs*, which refers to the total cost of the development and the transition, but also other costs too.
- C3. *Insource Development*, which refers to the work which is undertaken by employees internal to the organization,
- C4. *Outsource Development*, which refers to the work which is undertaken by employees from a consultancy company or any other subcontractor working for the organization.
- D1. *Mobile Technologies*, which stands out for the ability of storing data via mobile devices or tablet PCs ,
- D2. *Internet Publications*, which refer to the possibility of publishing online information
- D3. *Barcode/RFID Implementation*, which is related to managing and monitoring materials via barcodes or RFID technology,
- D4. *GIS Interconnection of Distribution Network*, which refers to Geographical mapping of the distribution network through GIS technology

A summary description of the requirements in both levels (Higher and Lower) is shown in Table 2.

**Table 2.** Requirements in Two Levels

<b>Higher Level</b>	<b>A: Features</b>		<b>B: System Properties</b>	
<b>Lower Level</b>	A1	Usability	B1	Ability of Statistic Services
	A2	Operability	B2	Configurability
	A3	System Response Time	B3	Publishing-printing Forms/ Limiting Bureaucracy
	A4	Data Security	B4	Procedures Implementation
	A5	Adaptability	B5	Prevention of duplicate data entries
<b>Higher Level</b>	<b>C: Project Management</b>		<b>D: New Technology Adaptation</b>	
<b>Lower Level</b>	C1	Time	D1	Mobile Technologies
	C2	Cost	D2	Internet Publications
	C3	Insource Development	D3	Barcode/RFID Implementation
	C4	Outsource Development	D4	GIS Interconnection of Distribution Network

In order to apply specific statistical techniques to the data regarding the 18 lower-level requirements considering all of them together, we applied a weighting conversion schema. The data obtained by Hierarchical Cumulative Voting (HCV) were transformed to simple Cumulative Voting (CV) results by a procedure described by



Berander et Jönsson in [7]. The basic idea of this procedure is to convert the amount assigned to each one of the requirements of the lower level by taking into account the amount assigned to the high-level requirement where they belong and also the number of the low-level requirements belonging in the requirement of the higher level. The data obtained after this transformation are weighted on a common base and they are of a specific kind since they are summed up to 1 (actually their sum is 100, but by a simple division with 100, we can consider that they sum up to 1). This methodology essentially addresses RQ1.

## 4 The Statistical Methodology

In this section we briefly discuss the statistical methods that were used in our analysis. As already mentioned the data collected through the questionnaires are of a specific form with a certain dependence structure known in bibliography as Compositional Data [15].

### 4.1 Descriptive Statistics

Descriptive statistics involve the computation of simple summary statistics like minimum and maximum values, the mean, standard deviation and the median of the data. Descriptive statistics are computed separately for higher and lower level and also for different roles of stakeholders. The statistics are accompanied by graphical representations like bar charts and box plots.

### 4.2 Compositional Data Techniques

Regarding the general theory, the term compositional data refers to vectors of variables having a certain dependence structure. The values of the variables in each vector are nonnegative and their sum is equal to one. Thus, our dataset consists of vectors of proportions or percentages  $(\mathbf{p}_1, \dots, \mathbf{p}_k)$  in the following form:

$$\sum_{i=1}^k p_i = 1 \quad p_i \geq 0 \quad (1)$$

Since the sum is fixed, there is a problem of interdependence of the proportions and therefore they cannot be treated as independent variables. Moreover, normality assumptions are invalid since the values are restricted in the  $[0,1]$  interval and finally the relative values of proportions are of particular interest instead of their absolute values. So, our dataset needs to be transformed in order to address a problem of analysis and interpretation of ratios of proportions. Therefore, in order to apply various classical univariate and multivariate statistical methods like ANOVA and Discriminant Analysis, the proportions are transformed to ratios by appropriate formulas like the clr transformation we describe below.

Since we are interested in ratios, the problem of zeros in these data is of principal importance since division by zero is prohibited. According to the general theory, the

zeros can be essential (i.e. complete absence of a component) or rounded (i.e. the instrument used for the measurements cannot detect the component). However, in our context, the importance of a requirement is expressed as a measurement according to the human judgment. So we can assume that a low prioritization is actually measured by a very low value in the 100\$ scale which is rounded to zero. Of course, when all respondents allocate zero to an issue, this can be excluded from the entire analysis and considered as essential zero.

Due to the problems of zeros, the various ratios needed for the analysis are impossible to be computed. It is therefore necessary to find first a way of dealing with the zeros. In our case we used a simple method proposed by [27], known as multiplicative zero replacement strategy. According to this method, every vector  $p = (p_1, \dots, p_k)$  consisting of proportions or percentages as described in (1) and having  $c$  number of zeros can be replaced by a vector  $r = (r_1, \dots, r_k)$  where

$$r_j = \delta_j \text{ (if } p_j = 0) \text{ or } r_j = p_j \left( 1 - \sum_{l: p_l = 0} \delta_l \right) \text{ (if } p_j > 0) \quad (2)$$

and where  $\delta_j$ s a very small imputed value for  $p_j$ .

As already mentioned, the transformation of the proportions is essential so as to convert them to functions of relative proportions. Aitchison [15] proposed the centered log-ratio (clr) transformation for transforming the compositional data to a form that can be analyzed by traditional statistical methods. The clr transformation requires the division of each component of a vector of proportions by their corresponding geometric mean. Since in our context the data contain zeros, the clr transformation can be applied after the replacement of zeros. The formula for the clr transformation performed in the compositional vector  $\mathbf{r}$  is as follows:

$$\mathbf{y} = \text{clr}(\mathbf{r}) = \left[ \log \frac{\mathbf{r}}{g(\mathbf{r})} \right] \quad (3)$$

where the geometric mean is given by

$$g(\mathbf{r}) = \left( \prod_{i=1}^k r_i \right)^{1/k} \quad (4)$$

The zero replacements as well as CLR transformations were carried out with the CoDAPack 3D [28].

### 4.3 Analysis of Variance (ANOVA)

In general, the purpose of analysis of variance (ANOVA) is to test for significant differences between means of different groups. In our study we used one-way ANOVA and post hoc tests in order to examine if there are any requirements which receive significantly different prioritization with respect to a categorical grouping variable. In our case, this grouping variable contains the different roles of the stakeholders. The analysis of the differences between groups is graphically depicted

by box-plots. In total, we performed 18 one-way ANOVA tests, one for each of the transformed lower level requirements. It should be emphasized that the initial transformation described in [7] essentially weights the lower-level requirements using the values and the structure of the higher-level requirements, i.e. the new transformed values contain information of both levels. That is why we do not perform ANOVA for the higher level (A-D). With this method we address RQ2.

#### 4.4 Discriminant Analysis (DA)

Discriminant Analysis (DA) is a multivariate statistical method used to explain the relationship between a dependent categorical variable and a set of quantitative independent variables. In our case, DA builds a linear model with the role grouping as the dependent variable, and the 18 software requirements as the independent variables. Due to the fact that not all of the independent variables are necessary for an efficient model, we used a stepwise procedure which selects the best variables for the model.

Our aim with stepwise DA is to discover which of the requirement prioritizations are most significant for discriminating the groups of stakeholders, i.e. to address RQ3. DA was first applied to the initial 4 inherent groups, but since the model was not satisfactory (due to high misclassification), we applied it to a new grouping with only two categories.

Discriminant Analysis (DA) is more informative than the simple one-way ANOVA, since it considers all the requirements together as a set and not each one separately. Although multiple ANOVA tests can provide indications of significant differences for each individual variable, there is a problem with the statistical inference from all of them since there is an inflation of the Type I error. A solution in this regard is the use of DA which considers all variables together and takes into account their correlations [29].

Furthermore, the stepwise approach accounts for the possible correlations between the independent variables.

The accuracy of DA can be evaluated by the Receiver Operating Characteristic (ROC) curve [30] and the related Area Under the Curve (AUC) statistic. ROC is a graphical tool which represents the performance of a binary classifier regarding the true positives rate, called sensitivity, versus the false positive rate of the prediction, known as one minus specificity. Values of AUC very close to 1 show high accuracy.

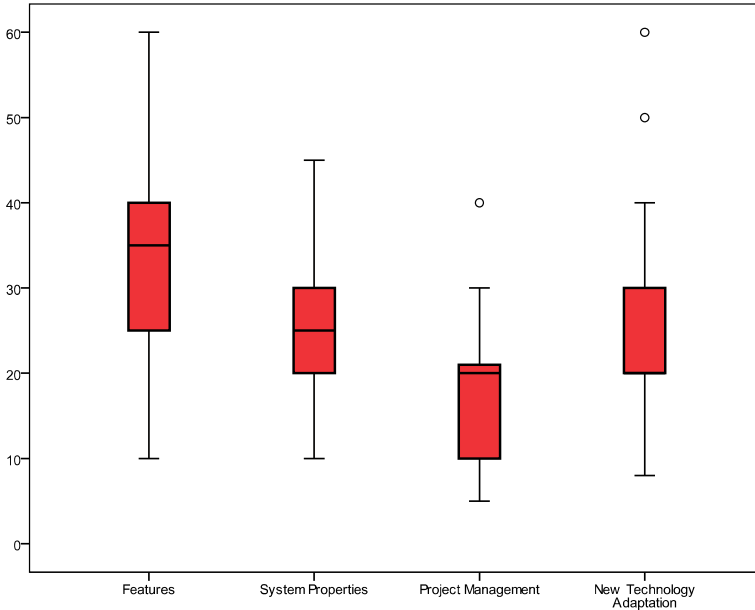
ANOVA, box plots and DA results were carried out with the PASW/SPSS statistical software [29].

## 5 Results

The results from an initial descriptive analysis (Table 3 & Fig 1) revealed that regarding the higher level, all the stakeholders prioritize *Features* (A) first, assigning in general higher values (mean value 34.5, median 35). On the other hand, the requirements related to *Project Management* (C) seem to receive the lowest values (mean value 17.5, median 20).

**Table 3.** Results of Descriptive Statistics of Higher Level requirements

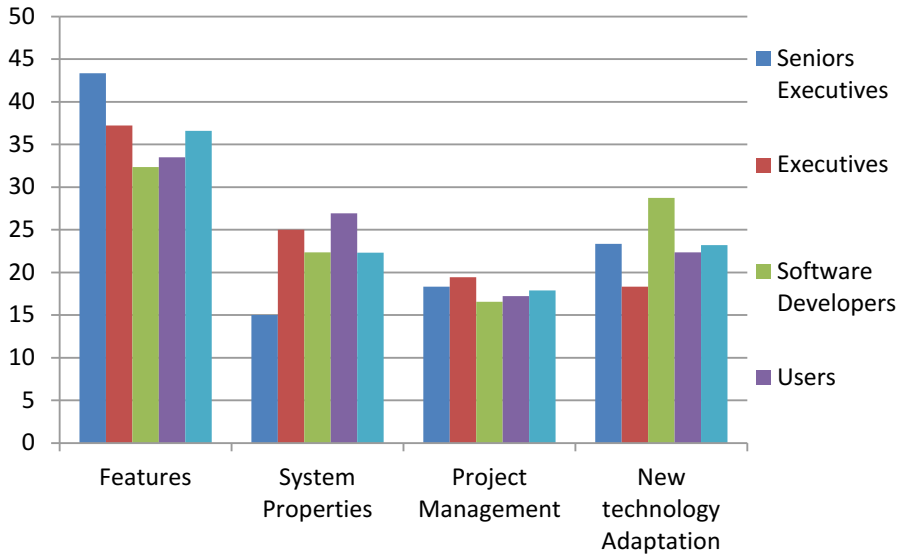
	<b>Features (A)</b>	<b>System Properties (B)</b>	<b>Project Management (C)</b>	<b>New Technology Adaptation (D)</b>
Mean	34.5	24.9	17.5	23
Median	35	25	20	20
Minimum	10	10	5	8
Maximum	60	45	40	60
Std Deviation	10.85	7.72	7.42	9.86

**Fig. 1.** Box Plot for the requirements prioritizations at the Higher Level (original values)

Taking into consideration the prioritization according to the different groups of stakeholders, the results from the descriptive analysis regarding the higher level revealed that all groups assign at the average the highest values to *Features (A)*. The largest value of all is assigned by *Senior Executives* stakeholders (Table 4 and Fig 2). On the other hand, the requirements related to *System Properties (B)* seem to receive the lowest values from this group of stakeholders. Seniors seem to emphasize more on a powerful IT system, with strong feature characteristics, and less on practical issues like the ability of statistic services or the implementation procedures. Other interesting findings are: requirements related to *New Technology Adaptation* issues seem to be more important for the *Software Developers* than the other groups of stakeholders, *Systems Properties* are more important to simple *Users* and *Project Management* requirements seem to be more important for *Executives*.

**Table 4.** Mean values of high-level requirements in the different groups of stakeholders

	Features (A)	System Properties (B)	Project Management (C)	New Technology Adaptation (D)
Senior Executives	43.33	15.00	18.33	23.33
Executives	37.22	25.00	19.44	18.33
Software Developers	32.36	22.36	16.55	28.73
Users	33.50	26.93	17.21	22.36



**Fig. 2.** Bar chart of means of all high-level requirements in the groups of stakeholders

The results obtained at the lower level are presented in Table 5. Here we report the statistics of the weighted percentages taking into account the number of lower level issues within each issue of the higher level [7]. Again, requirements related to *Features*, are prioritized with the highest values (*Usability (A1)*, *Operability (A2)* and *System Response Time (A3)*). The requirement with the lowest values is *Outsource Development (C3)*. It is also remarkable that *Prevention of duplicate data entries (B5)* receives in general high values of prioritization.

Similar results are obtained regarding the different groups of stakeholders (Table 6). It is clear (Table 6) that requirements *Usability (A1)* and *Operability (A2)* are prioritized high and especially by senior stakeholders. On the contrary, it is noticeable that *Users* seem to assign low amounts on *Data security (A4)* and *Adaptability (A5)*, which deserves further investigation. Hence *Users* seem to attribute high values to the *Ability of Statistical Services (B1)*, *Bureaucracy issues (B3)* as well as to the *Prevention of Duplicate Data (B5)*.

Taking into consideration the *Project Management* requirements, the results were more or less expected. Senior stakeholders prioritize higher issues related with *Cost (C2)*, where on the other hand *Software Developers* assign higher values to *Time (C1)*. Finally, *Software Developers* prioritize with high values all the requirements related to *New Technology Adaptation (D)* issues.

**Table 5.** Results of Descriptive Statistics of Lower Level requirements

	Mean	Median	Min	Max	Std Deviation
<b>A1:</b> Usability	9.80	9.51	1.14	31.91	6.01
<b>A2:</b> Operability	8.09	6.67	1.14	17.39	4.17
<b>A3:</b> System Response Time	9.13	8.51	1.11	26.6	4.35
<b>A4:</b> Data Security	4.89	4.79	0	9.78	2.37
<b>A5:</b> Adaptability	5.44	4.89	0	21.28	3.56
<b>B1:</b> Ability of Statistic Services	6.79	6.62	0	16.48	3.78
<b>B2:</b> Configurability	3.57	3.30	0	15.63	2.74
<b>B3:</b> Publishing-printing Forms/ Limiting Bureaucracy	5.74	5.62	0.31	21.54	3.40
<b>B4:</b> Procedures Implementation	3.86	3.33	0	10.77	2.36
<b>B5:</b> Prevention of duplicate data entries	7.09	6.59	2.13	13.64	3.27
<b>C1:</b> Time	4.88	4.44	0.64	10.67	2.57
<b>C2:</b> Cost	4.07	3.52	0.08	10.55	2.45
<b>C3:</b> Insource Development	4.09	3.64	0.08	8.89	2.07
<b>C4:</b> Outsource Development	2.30	1.98	0	8.89	1.69
<b>D1:</b> Mobile Technologies	6.10	6.02	1.70	16.74	3.21
<b>D2:</b> Internet Publications	4.87	4.35	0	18.18	3.37
<b>D3:</b> Barcode/RFID Implementation	3.82	3.40	0	10.21	2.21
<b>D4:</b> GIS Interconnection of Distribution Network	5.47	5.22	0.42	16.74	2.99

**Table 6.** Results of Descriptive Statistics regarding the different groups of stakeholders: Lower Level

	Senior Executives	Executives	Software Developers	Users
<b>A1:</b> Usability	13.95	9.86	7.84	10.12
<b>A2:</b> Operability	11.98	8.19	7.65	7.81
<b>A3:</b> System Response Time	7.10	9.14	6.67	10.31
<b>A4:</b> Data Security	7.84	6.85	5.92	3.54
<b>A5:</b> Adaptability	6.17	6.15	7.32	4.39
<b>B1:</b> Ability of Statistic Services	4.21	7.69	5.04	7.46
<b>B2:</b> Configurability	2.02	3.09	3.96	3.74
<b>B3:</b> Publishing-printing Forms/ Limiting Bureaucracy	2.29	4.95	4.55	6.82
<b>B4:</b> Procedures Implementation	3.72	4.86	4.35	3.37
<b>B5:</b> Prevention of duplicate data entries	4.21	6.39	6.54	7.85
<b>C1:</b> Time	4.85	5.10	5.21	4.68
<b>C2:</b> Cost	4.99	4.70	4.10	3.77
<b>C3:</b> Insource Development	4.26	4.41	3.10	4.37
<b>C4:</b> Outsource Development	1.99	2.68	2.22	2.24
<b>D1:</b> Mobile Technologies	5.29	5.07	7.33	6.02
<b>D2:</b> Internet Publications	3.52	3.54	6.49	4.80
<b>D3:</b> Barcode/RFID Implementation	4.36	3.66	4.38	3.59
<b>D4:</b> GIS Interconnection of Distribution Network	7.26	3.68	7.34	5.13

The prioritized data of the lower level were transformed and analyzed separately with the methods of CoDa described in the previous section (replacement of zeros and clr transformation). One-way ANOVA and DA were applied to the obtained data.

The results from one-way ANOVA revealed that there is a statistically significant difference between the way the different groups of stakeholders prioritize some of the 18 requirements of the lower level. Specifically, the requirements with significant differences between the different roles of the stakeholders ( $p < 0.05$ ) are the following five: *Data Security (A4)*, *Adaptability (A5)*, *Publishing-printing Forms/Limiting Bureaucracy (B3)*, *Insource Development (C3)* and *GIS Interconnection of Distribution Network (D4)*.

Furthermore, for those requirements that ANOVA showed significant difference, we used the Tukey's post-hoc test (for equal variances) and the Games-Howell (for unequal variances) in order to see which of the groups are different. For each of these five requirements the appropriate post-hoc test gave the following results:

*Data security (A4)*: There is statistically significant difference only between *Users* and all the other roles. The important finding here is that *Users* prioritize this requirement with significantly lower values than all the other groups (Fig. 3). It is also notable that A4 is very important issue for the *Senior Executive* stakeholders.

*Adaptability (A5)*: The only statistically significant difference here is between *Users* and *Software Developers*. Again it is remarkable that *Users* prioritize low adaptability issues (Fig. 4). This is not so unexpected result since system's ability of adjusting to the new processes or changes of the existing framework is not of the main concern of a simple user.

*Publishing-printing Forms/Limiting Bureaucracy (B3)*: (Fig. 5 ). The significant differences are between *Users* and *Senior Executives*, and between *Users* and *Software Developers*. Here, it seems that simple users are more concerned about issues related to the workload due to the bureaucracy that can be limited by the new system.

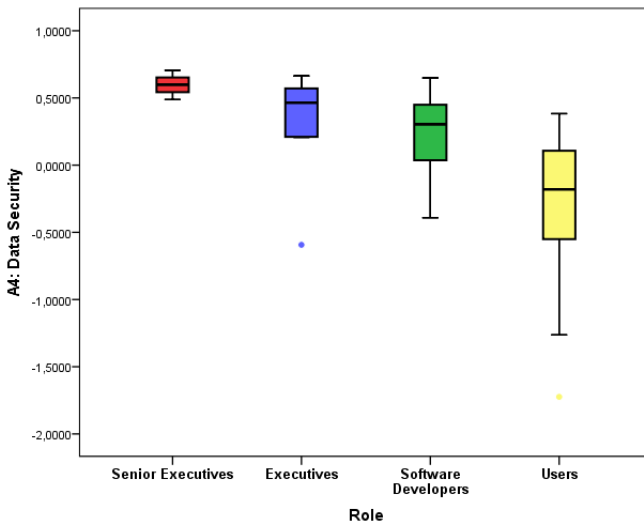


Fig. 3. A4: Data Security (clr transformed values)

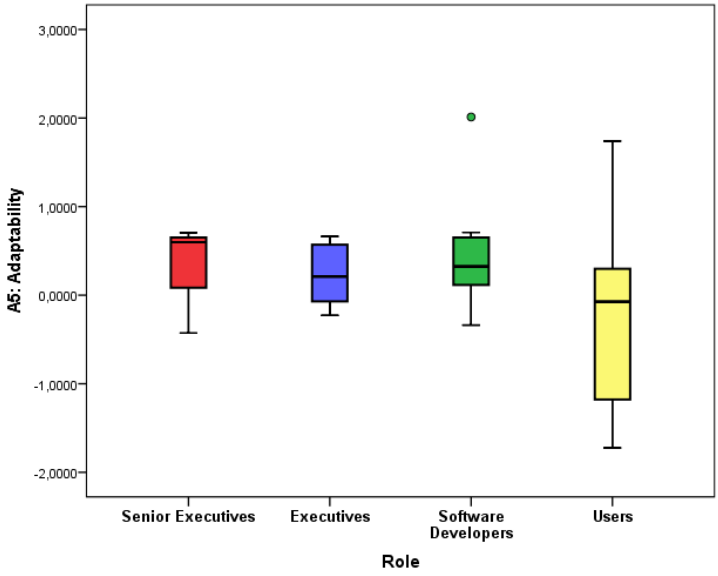


Fig. 4. A5: Adaptability (clr transformed values)

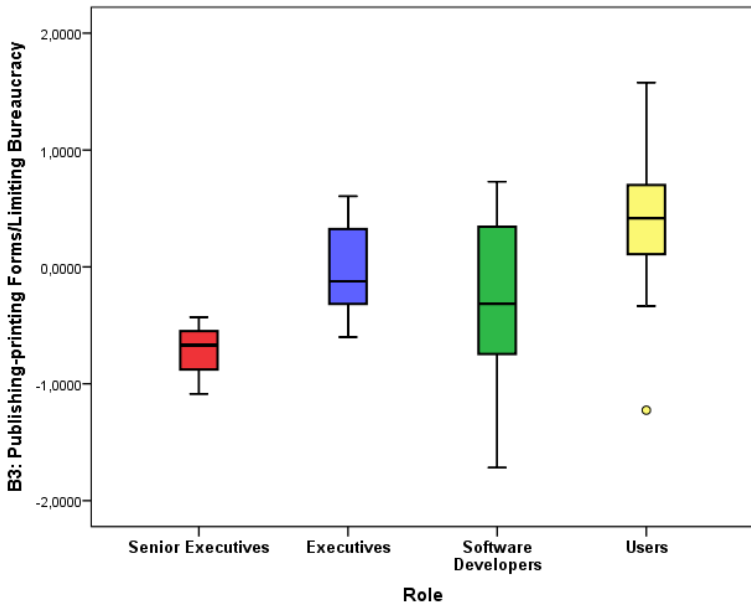


Fig. 5. B3: Publishing-printing Forms/Limiting Bureaucracy (clr transformed values)



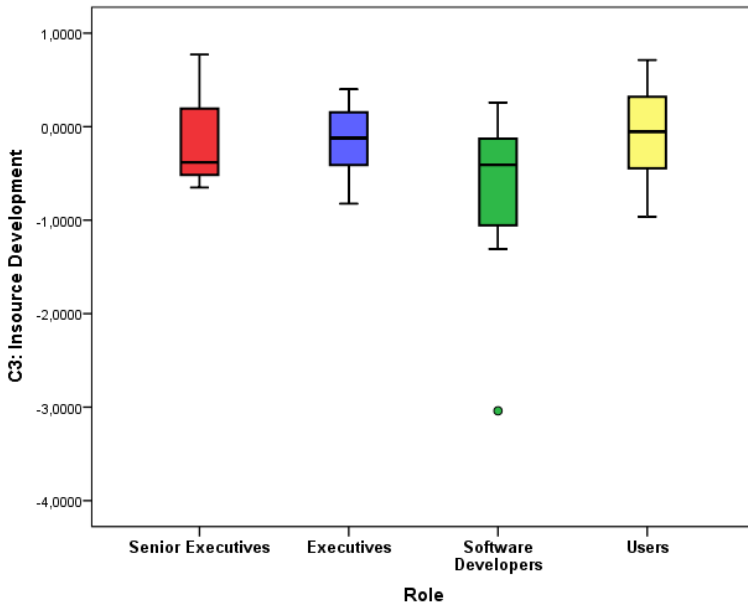


Fig. 6. C3: Insource Development (clr transformed values)

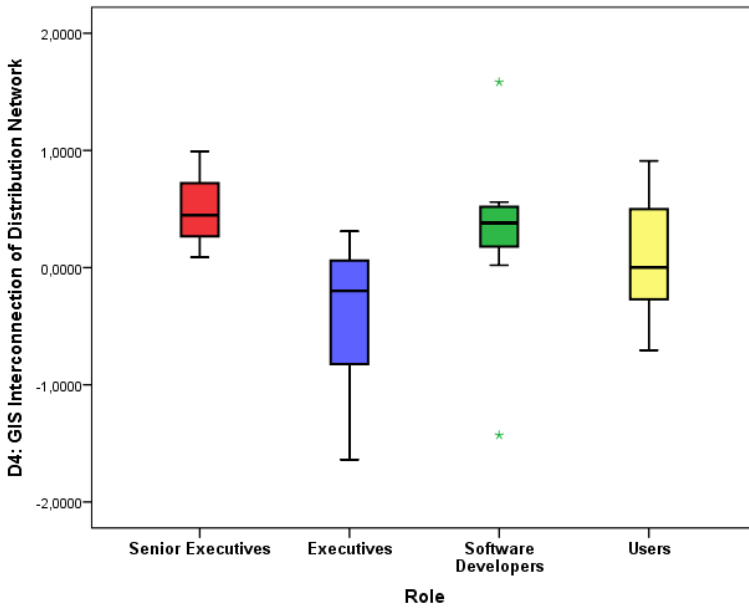


Fig. 7. D4: GIS Interconnection of Distribution Network (clr transformed values)

*Insource Development (C3)*: The only significant difference here is between *Software Developers* and *Users*. The unexpected finding is that *Users* are more concerned about the insource development than the developers themselves (Fig. 6). It is true that users count on the company's development team. The developers are able to understand user's daily routine and can deliver a better application.

*GIS Interconnection of Distribution Network (D4)*: There was only one significant difference, between *Executives* and *Software Developers*. It seems here that the group of executives prioritizes low the issue of geographical mapping of the distribution network through GIS technology (Fig. 7).

At the next step in our analysis we perform stepwise Discriminant Analysis (DA). DA is used to address the following research question: Is there a combination of requirements which are able to discriminate the various groups of stakeholders according to their prioritization attitude? As already mentioned, DA was first applied to the initial 4 inherent groups, but the model was not satisfactory since only the 58,8% of the cross validated groups of stakeholders were classified correctly. Hence we applied it to a new grouping with only two categories. Specifically, we congregated our stakeholders into 2 groups.

The first group contains the first 3 role categories (*Senior Executives*, *Executives* and *Software Developers*) while the *Users* consist the second group. This division was decided so as to have in the first group employees with high expertise and high access in internal company information in technical or managerial and administrative issues (Experts) and in the second group Regular Users. Of course, it can be argued that users are experts too, especially regarding specific operations within the organization. However, their perspective of the system as a whole is quite different since they do not actually have to take managerial decisions. The distribution of the stakeholders within the second grouping is shown in Table 7.

According to the stepwise DA results, the statistically significant requirements for the discrimination of the stakeholders groups are the following four: *Data Security (A4)*, *System Response Time (A3)*, *Configurability (B2)* and *Insource Development (C3)*. Moreover, according to the Wilk's Lambda criterion, the most important requirement for the discrimination of the groups is *Data Security (A4)*.

The classification accuracy is satisfactory since the overall cross validation accuracy rate is 84.3%. More specifically, DA classifies correctly 91.3% of the Experts, while the Regular Users are correctly classified at a 78.6% rate. The ROC curve of DA discriminant scores is presented in Fig. 8. The AUC statistic is 0.936 ( $p < 0.001$ ) showing that the classification of groups of stakeholders is achieved with high accuracy based on their prioritization of the four aforementioned requirements.

The coefficients of the discriminant function resulted from DA show that Experts prioritize higher than Regular Users the requirement for *Data Security*. On the other hand, Regular Users prioritize higher all the other three requirements, i.e. *System Response Time*, *Configurability* and *Insource Development*.

**Table 7.** Groups of Stakeholders

<b>Role</b>	<b>TOTAL</b>	<b>Percent (%)</b>
Experts	23	45,1%
Regular Users	28	54,9%
<b>TOTAL</b>	<b>51</b>	<b>100%</b>

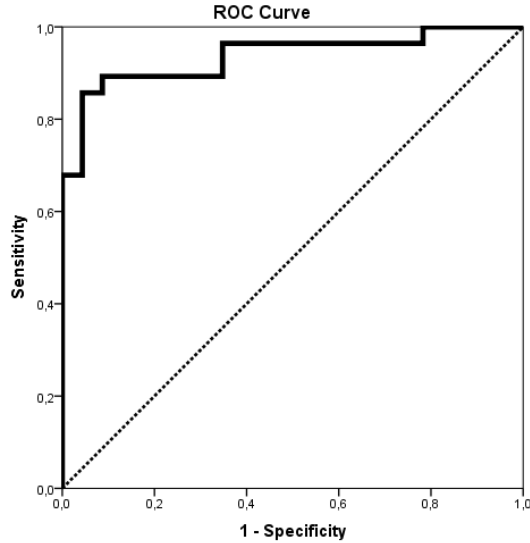


Fig. 8. ROC curve for the Discriminant Analysis for Experts and Regular Users prioritization

## 6 Discussion of Results and Threats to Validity

The statistical methods analyzed the different views, priorities and expectations for the new IT system. The results show that the four different groups of stakeholders and the merged two groups of users have only few differences in prioritization of requirements. Both analyses (ANOVA and DA) revealed the significant differences regarding the issues Data Security and Insource Development. Experts are very concerned for the security of data due to their sensitive position in the company. On the other hand, regular users pay more attention to the confidence they have for the company personnel developing the new system. In general, the results show that there are no strong disagreements that would threaten the success of the project.

Software development projects usually start with high expectations and promises to improve and support everyday business activities but sometimes end in failure (see for example [31]). Among the common factors of failure is the poor communication between the stakeholders [31], [32], [33]. In our case study the IT executives face the challenge of success in an urgent way, since the employees of the company already use the current system and therefore the comparison with the new one is inevitable. On the other hand, internal regular users have to use the system on an everyday basis since it is the only available for their work. Therefore user satisfaction is an important success indicator for the project. The results of the case study help the company's IT decision makers in consecutive phases. First, they can easily review the current situation and identify weaknesses and plans for improvements on the new system. Then, they need to develop an implementation plan considering the research results in order to deploy a new software that meets the users' expectations. Finally, during the testing period, they can choose beta testers from the interviewed employees and compare their experience obtained from the new system with their answers in the present study.

The fact that the differences found in prioritizations of stakeholders are only few and in general anticipated and interpretable is positive in the sense that they can be taken into account easily and implemented in the developed software. The lack of strong and conflicting disagreements is an indication that the transition to the new modernized Web technologies will be achieved smoothly and with limited criticism.

Regarding the validity threats, the study is clearly exploratory and by no means can the findings be generalized to other companies or situations. The stakeholders do not constitute a random sample; however they were approached for their experience and expertise so their responses are considered especially valid.

As a future work we plan to repeat the study with the same respondents after their interaction with the new system. This replication of the survey will concern issues of the new systems' functionality, corresponding to the requirements studied here. This will essentially provide feedback for quality evaluation of the new system.

**Acknowledgment.** We wish to thank all respondents that participated in the study. Also, we thank the reviewers for their constructive comments which led us to improve the paper.

## References

1. Mayle, D., Bettley, A., Tantoush, T. (eds.): *Operations management: A strategic approach*. Sage Publications Limited (2005)
2. Leung, H.K.: Quality metrics for intranet applications. *Information & Management* 38(3), 137–152 (2001)
3. Johansson, E., Wesslén, A., Bratthall, L., Host, M.: The importance of quality requirements in software platform development—a survey. In: *The 34th Annual Hawaii International Conference on System Science*. IEEE (2001)
4. Offutt, J.: Quality attributes of Web software applications. *IEEE Software* 19(2), 25–32 (2002)
5. Berntsson Svensson, R., Gorschek, T., Regnell, B., Torkar, R., Shahrokni, A., Feldt, R.: Quality Requirements in Industrial Practice—An Extended Interview Study at Eleven Companies. *IEEE Transactions on Software Engineering*, 38(4), 923–935 (2012)
6. Phillips, L.B., Aurum, A., Svensson, R.B.: Managing Software Quality Requirements. In: *The 38th EUROMICRO Conference in Software Engineering and Advanced Applications (SEAA)*, pp. 349–356. IEEE (2012)
7. Berander, P., Jönsson, P.: Hierarchical Cumulative Voting (HCV) -Prioritization of requirements in hierarchies. *International Journal of Software Engineering and Knowledge Engineering* 16(6), 819–849 (2006)
8. Berander, P., Jönsson, P.: A goal question metric based approach for efficient measurement framework definition. In: *The 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE 2006)*, pp. 316–325. ACM, New York (2006)
9. Leffingwell, D., Widrig, D.: *Managing software requirements: A Use Case Approach*, 2nd edn. Addison-Wesley, Boston (2003)
10. Regnell, B., Host, M., Natt och Dag, J., Beremark, P., Hjelm, T.: An industrial case study on distributed prioritisation in market-driven requirements engineering for packaged software. *Requirements Eng.* 6, 51–62 (2001)

11. Berander, P., Wohlin, C.: Difference in views between development roles in software process improvement – A quantitative comparison. In: *Empirical Assessment in Software Engineering, EASE* (2004)
12. Karlsson, L.: Requirements prioritisation and retrospective analysis for release planning process improvement. PhD Thesis, Department of Communication Systems Lund Institute of Technology (2006)
13. Berander, P.: Evolving prioritization for software product management. Phd Thesis Department of Systems and Software Engineering School of Engineering Blekinge Institute of Technology, Sweden (2007)
14. Chatzipetrou, P., Angelis, L., Barney, S., Wohlin, C.: Software product quality in global software development: Finding groups with aligned goals. In: *The 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2011)*, pp. 435–442. IEEE Press (2011)
15. Aitchison, J.: *The Statistical Analysis of Compositional Data*. The Blackburn Press, London (2003)
16. Chatzipetrou, P., Angelis, L., Rovegård, P., Wohlin, C.: Prioritization of Issues and Requirements by Cumulative Voting: A Compositional Data Analysis Framework. In: *The 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2010)*, pp. 361–370. IEEE press (2010)
17. Chatzipetrou, P., Papatheocharous, E., Angelis, L., Andreou, A.S.: An Investigation of Software Effort Phase Distribution Using Compositional Data Analysis. In: *The 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2012)*, pp. 367–375. IEEE press (2012)
18. Riņķevičs, K., Torkar, R.: Equality in cumulative voting: A systematic review with an improvement proposal. *Information and Software Technology* (2012), doi:10.1016/j.infsof.2012.08.004
19. Rovegard, P., Angelis, L., Wohlin, C.: An empirical study on views of importance of change impact analysis issues. *IEEE Transactions on Software Engineering* 34(4), 516–530 (2008)
20. Robson, C.: *Real World Research*. Blackwell (2002)
21. Freeman, R.E.: *Strategic Management: A stakeholder approach*. Pitman, Boston (1984)
22. Kotonya, G., Sommerville, I.: *Requirements Engineering: processes and techniques*. John Wiley (1998)
23. Nuseibeh, B., Easterbrook, S.: Requirements engineering: a roadmap. In: *Proceedings of the Conference on the Future of Software Engineering*. ACM (2000)
24. Aurum, A., Wohlin, C.: The fundamental nature of requirements engineering activities as a decision-making process. *Information and Software Technology* (2003)
25. Eason, K.: *Information Technology and Organisational Change*. Taylor and Francis (1987)
26. Sharp, H., Finkelstein, A., Galal, G.: Stakeholder identification in the requirements engineering process. In: *Proceedings of 10th International Workshop on Database and Expert Systems Applications*. IEEE (1999)
27. Martín-Fernández, J.A., Barceló-Vidal, C., Pawlowsky-Glahn, V.: Zero replacement in compositional data sets. In: Kiers, H., Rasson, J., Groenen, P., Shader, M. (eds.) *The 7th Conference of the International Federation of Classification Societies (IFCS 2000)*. *Studies in Classification, Data Analysis, and Knowledge Organization*, pp. 155–160. Springer, Berlin (2000)
28. Comas-Cufí, M., Thió-Henestrosa, S.: CoDaPack 2.0: a stand-alone, multi-platform compositional software. In: Egozcue, J.J., Tolosana-Delgado, R., Ortego, M.I. (eds.) *The 4th International Workshop on Compositional Data Analysis* (2011)

29. Field, A.: *Discovering statistics with SPSS*. Sage, London (2005)
30. Krzanowski, W.J., Hand, D.J.: *ROC curves for continuous data*. Chapman & Hall/CRC (2009)
31. Charette, R.N.: Why software fails. *IEEE Spectrum* 42(9) (2005)
32. Sumner, M.: Critical success factors in enterprise wide information management systems projects. In: *The 1999 ACM SIGCPR Conference on Computer Personnel Research*, pp. 297–303. ACM (1999)
33. Poon, P., Wagner, C.: Critical success factors revisited: success and failure cases of information systems for senior executives. *Decision Support Systems* 30(4), 393–418 (2001)

# Challenges and Solutions in Global Requirements Engineering – A Literature Survey

Klaus Schmid

Software Systems Engineering, Institute of Computer Science,  
University of Hildesheim, Hildesheim, Germany  
schmid@sse.uni-hildesheim.de

**Abstract.** Requirements engineering is a core part of the software engineering lifecycle with tremendous leverage on software development success. It becomes particularly difficult in the context of global software engineering due to the need to coordinate many different stakeholders in a distributed setting. In this paper we survey some main results in global requirements engineering. We will address both the development of systems with internationally distributed customers as well as the situation of globally distributed development.

**Keywords:** requirements engineering, global software engineering, distributed development, internationalization, communication, coordination.

## 1 Introduction

Today, software development happens increasingly in a globally distributed context. This is due to a number of factors. In particular, markets themselves are increasingly globalized, which means that sometimes even small companies build software for a customer on another continent. But also the competition for competent software engineers is strong in many parts of the world, leading to the creation of development centers around the world by many large corporations. Whatever the reason is, the consequence is that it is important to perform requirements engineering in a way that deals with the aspects of *distribution* and *internationalization*. In this study, we aim to summarize results from an analysis of results from global requirements engineering.

The recognition that global requirements engineering poses particular and challenging problems is not new. Cheng and Atlee described this as an important requirements engineering problem [1]. Herbsleb also described requirements engineering as an important problem in global software engineering [2].

When discussing the problem of global requirements engineering, we found it important to distinguish:

1. *Internationalization*: the development for a set of international (globally distributed) customers (perhaps with a single, localized development team) *and*
2. *Distribution*: the development in a globally distributed environment, where many developers are in a different location from the customer(s).

While both problems often co-occur, they are different and may require slightly different strategies for handling them. This is particularly true as key parameters like the possibility to influence the organizations differ. For example, a development organization may decide to change its processes for all development personnel, but it is not able to influence processes in the customer organization. However, also common strategies exist as some problems are crosscutting such as handling cultural differences.

The first of these problems also entails that systems must be customized to the demands of a particular customer, especially if the same or similar systems must be provided to multiple customers. The adaptation of software to an international customer base is often described as *Internationalization* [3][4][5]. Hence, we refer to this form of Requirements Engineering as *International Requirements Engineering (IRE)*. While internationalization addresses the customization to regional differences such as language, time zones, etc., often also functional customizations are required to adapt software to different customer needs [6]. This need for diversity also makes product line engineering relevant to this situation [7].

On the other hand requirements engineering may need to interact with a globally distributed development team. This might even mean that people within the company, who are involved in requirements engineering, are globally distributed. We will refer to this company-internal aspect as (Globally) *Distributed Requirements Engineering (DRE)*, as the main characteristic of this is the distribution of requirements engineering and the consequences this leads to, like company-internal culture clashes. We will use the term *Global Requirements Engineering (GRE)* to refer to the combination of both. In global software engineering the main focus is usually on those aspects that are related to distribution. It should be noted that our use of the term global is intentionally broader than this and also includes the identification of international differences in requirements and providing the basis for the corresponding adaptation, as this is part of the globalization problem as a whole.

While in practice DRE and IRE may occur together, they are conceptually different and each may appear alone as well. In particular, a situation where IRE will occur without (relevant) DRE is when the customer itself is distributed or there are a number of different international customers, but the development is collocated at a single site. On the other hand, the customer might be next door to the requirements engineer, while the development team is globally distributed.

In Section 2 we will survey results on IRE, while in Section 3 we will discuss DRE. Finally, in Section 4 we will conclude.

## 2 International Requirements Engineering

In this section, we focus on the issue of internationalization of software engineering and what this implies for requirements engineering. Thus, while we do not necessarily assume that software development itself is distributed (actually the whole software development might be collocated in a single office), we assume that at least some customers will be in a different country. This entails, in particular, that there is a



difference in location and often in culture and language between the customer and the development organization. Moreover, often an additional dimension of complexity is introduced, if multiple customers in different countries must be supported at the same time and their demands must be combined into a single system or product line. While this seems a rather common situation in software engineering, we could only identify few sources in the globalization and requirements engineering literature that addressed the requirements engineering problems in internationalization explicitly.

## 2.1 Research Questions

Internationalization and interfacing with an internationally diverse set of customers leads to a number of problems. The questions we tried to answer as part of our survey were mostly developed based on trying to identify those issues we expected that differences in internationalization would make the largest impact (i.e., context issues, language issues, interacting with a set of distributed customers). Our list of questions was also influenced by an initial analysis of the relevant literature on internationalization and global requirements engineering [7], thus, we do not list here research questions, for which we did not find a substantial amount of information. This led us to the following research questions:

1. Which models exist for organizing customer interaction?
2. How to address issues of cultural diversity? More precisely how to bridge a cultural gap between a client and the development organization?
3. How to address context issues (e.g., different legal, climate, educational environments)?
4. How to deal with requirements negotiation and integration?

We are fully aware that internationalization may impact many more aspects of requirements engineering. However, we restricted ourselves to this list to ensure that the problem would remain manageable. We will now discuss our core findings with respect to each of the questions given above in individual subsections.

## 2.2 Organizational and Communication Issues in Customer Interaction in Requirements Elicitation

Customer interaction is a core part of requirements engineering as it is central to requirements acquisition. If it is difficult to interact with all relevant stakeholders, often problems with inadequate requirements occur, which may significantly hinder development. This problem is particularly strong if due to distance the relevant stakeholders cannot be identified and included in the communication [10]. Prikładnicki et al. discuss a number of case studies from a range of different situations (internal and external development in offshore and near-shoring) and conclude that especially requirements elicitation is very problematic in offshoring models [11].

While we could not find a definite categorization of customer interaction models in literature, we believe that most situations can be categorized based on the following scheme. At least all cases we found in studies belong to the following categories:

1. *Own sales office*: there is an office of the development organization in the country.
2. *Partnering*: A partner relationship with a company from the target country exists. Thus, requirements exchange usually happens with this organization, as opposed to direct customer interaction.
3. *Repeated Meetings*: Repeated meetings with the customer organization are organized to ensure good communication. This might culminate in bringing developers to the customer organization.
4. *Co-located customer*: A member of the customer organization is embedded with the development team, e.g., as a product owner in an agile setting.
5. *Remote Interaction*: Use of techniques like video-teleconferences, email, etc.

Of course, the different approaches are not mutually exclusive. Rather they can be well combined to address the various problems of International Requirements Engineering. It is interesting to note that we did hardly find approach 2 discussed in scientific literature, while it seems that it is useful for small companies that produce standardized software for a broad market as it simplifies interaction and reduces costs. However, this implies a three-tiered organization: development organization, partner organization, and customer. As a downside this makes it hard to correctly identify the requirements of a customer as they may become distorted in the process. This is emphasized in [12], where it is mentioned that “It is typical that resellers distort the facts...”. It also leads to added complexity of the stakeholder network, but this problem is only partially related to internationalization, but rather to the very large number of entities with whom interaction is necessary. In this context also problems of dealing with prioritization and assessing the relative business value of requirements are amplified. They also report that the added link in the market communication introduces delays. These issues are mostly relevant to market-driven development, which often relies on strategies 1 or 2.

Some work also aims to provide techniques to analyze and improve distributed information flow in a general way (i.e., independent of a specific setup). This is typically related to communication and organizational issues. These approaches are not necessarily specific to requirements engineering [19], [20].

Most papers focus on cases with direct customer interaction. In this context it is still rather unclear how to organize it as evidence so far seems inconclusive. Hanisch and Corbitt [10] describe a case where several approaches are tried over the course of a project (users at development site, each at their site, developers (partially) at user site), but no matter which way the organization is done, still problems arise. They summarize their experience in the recommendation that it is important to put one person in charge of communication and also to use a video conferencing facility. But the question remains whether this should be regarded as a success case, as they state that the involved people would not like to repeat the experience.

While often the importance of personal interaction is emphasized (a viewpoint, which is also shared by the author), the study by Calefato et al. [13] shows that text-based elicitation can be as effective as face-to-face communication. However, they also note that participants prefer face-to-face communication. Thus, it might be more an issue of (social) preference than a fundamental quality of an approach.

As emphasized by Damian [8] a problem in global software engineering is the language. While the involved parties may share a language (e.g., English), they might only know it as a foreign language. In an activity that relies heavily on communication like requirements engineering, this weighs particularly heavy. However, this is only rarely explicitly discussed in most case studies. Rather, it seems that typically English is used as a common language and the assumption is that sufficient language skills exist on both sides to achieve a sufficiently fluent communication. An exception is presented in [14] who discuss a Chinese-German case study. English was used as a communication language, but they emphasize that problems arose, as it was a foreign language to both parties. Here, proxies who were explicitly tasked with keeping up communication helped to improve the situation.

If there is a huge time difference among the participating organizations this poses a big problem as Hanisch and Corbitt [10] discuss. This can be an additional driver to bring part of the development to the customer site. They also point out that it might lead to further effects like late working hours to create a shared time window.

### **2.3 Cultural Diversity**

An important problem in communication in Requirements Engineering is the issue of cultural diversity [23]. There might be fundamental differences between development organization and customer organization. Cultural differences are rather standard to requirements engineering on a mundane level like different company cultures or those that arise from different personal backgrounds (like engineering vs. marketing). However, this problem may be strongly compounded in the context of international development that stretches across different cultural zones (like Asian vs. European).

Brockmann and Thaumüller [14] discuss this issue in the context of a project with a German development organization and a Chinese customer organization. They focus in particular on the context of agile requirements engineering and point to several conflicts of agile approaches with both German and Chinese values. The problem is compounded by the fact that German and Chinese values are also directly in conflict. However, while they emphasize a number of problems, they only describe the approach of using proxies in order to address the problems. They do not discuss in detail how successful this approach is, actually it seems the evaluation is still ongoing.

It is interesting to note that Hanisch and Corbitt [10] discuss that even the cultural difference between New Zealand and the UK posed significant problems, despite the strong relations between the countries and their culture. They also mention that it is possible that the problems were compounded by the cultural differences between a small company and a large government organization.

An important question is to what degree does formal cultural knowledge help to bridge the gap. While the corresponding analysis by Gotel and Kulkarni [25] focuses on cultural differences within a development team, their results are rather negative: teams that received such training did not cooperate better than others, which did not get such training. It remains, however, unclear whether this result generalizes to other kinds of cultural training or is specific to their approach.

**Table 1.** Overview of Context Issues Particularly Relevant to Requirements Engineering

No.	Context Factor	Description
1	<i>Language</i>	customers may use a different language
2	<i>User Interface</i>	due to language issues and cultural issues it might be necessary to create different kinds of user interfaces
3	<i>Local Standards</i>	the customer might use different standards like calendars or measurement systems than the developer
4	<i>Laws and Regulations</i>	they might be fundamentally different between customer and developer, but taken as obvious, as the customer is never concerned with other rules, thus the presence of a difference might even go easily unnoticed
5	<i>Cultural Differences<sup>a</sup></i>	due to cultural differences the expectations regarding functionality, behavior, design, etc. of the final system may be profoundly different, again this usually forms tacit knowledge
6	<i>Regional Issues</i>	the situation for the customer might again be regionally sub-divided, so that not a single solution, but a product line or customizable system is required
7	<i>Educational and Work Context related Issues</i>	in different regions different levels of educational background might be expected, which may require very different user interfaces (e.g., when operating certain machinery [6])
8	<i>Environmental Conditions</i>	the products might need to work in environmental conditions, fundamentally different from those the developers assume, giving rise to corresponding requirements

a. This refers to cultural issues that have an impact on the final product as opposed to cultural issues that have an impact on the customer-developer-interaction.

So far cultural issues in global software engineering have been mainly discussed in the context of development team internal issues. Based on our overview, it is important to note that many of the typically recommended practices, such as cultural trainings, seem to provide less benefit than typically claimed.

## 2.4 Usage Context Issues

While the previous sections mainly focused on aspects that influence the requirements engineering method, context issues also influence the requirements themselves. Thus, it is important to IRE that these requirements are also identified as part of requirements engineering. The importance of context is also emphasized in [8].

We use the term (*usage*) *context issue* in a broad sense by referring with it to any issue that impacts the requirements of a system due to the place where the software is used. Sub-categories that we could identify from work on internationalization are given in TABLE 1. While this list is not necessarily complete, it is a good starting

point for addressing differences. The interesting point is, however, that we hardly found guidelines on how to deal with them in the literature that we analyzed from the requirements engineering and global software engineering backgrounds.

However, these issues are recognized in the area of software internationalization [3][6], although most work there typically focuses on languages and language-induced differences. The first three items in TABLE 1 are widely considered in internationalization [6][4]. For many of them (e.g., different languages for the user interface or different calendars or units of measurement) very good support exists also from a technical perspective. However, some impacts are much more subtle. For example, [6] describe a case study where differences in language (German vs. Chinese) led to differences in fonts and finally required a hardware redesign to ensure readability of the user interface. The authors also state that taking such requirements into account is of great importance to achieve product success (their first version, which did not take this into account was not successful). Another language issue is that some countries have more than one language and thus systems must be capable to switch among these languages at any time. An extreme case is India, where more than 20 languages and 11 scripts are in use [18].

Laws and regulations in different countries might introduce very subtle, but key distinctions, while having very profound effects on the product requirements. As customers cannot be expected to know them all, the development team faces the question of how to identify these requirements in a foreign state. A non-trivial challenge for which we could not find good advice in literature, especially as the developer team might not even be aware that relevant laws and regulations may exist. The standard requirements engineering answer for identifying such tacit knowledge is to use ethnographic methods like observation. However, these methods also require a significant amount of time, while being still prone to omissions of rare events [17].

Cultural differences may also significantly impact the details of the product requirements as they may form expectations. This might be in the form product requirements like the supported business process or it might impact the design of the user interface. Examples of the former are everyday processes like the usage of a gas pump, where substantial differences exist among countries with a corresponding impact on related products. This makes it very important to identify and understand cultural expectations early on, as otherwise the result may not be acceptable.

Also the educational background of people or environmental characteristics might significantly impact the requirements of the resulting system. For example, in some countries one needs to include particularly low-skilled workers. This might then have an impact on how they are expected to use machinery or what they are allowed to do. This might be further compounded if different groups are introduced with different skill levels [6]. It is very hard to identify such issues, without any ethnographic studies, as the customers will often not be able to voice these issues in the beginning (as they are obvious to them). Jantunen et al. [12] also emphasized the problems of identifying the needs (even in the presence of a proxy).

## 2.5 Requirements Negotiation and Prioritization

Requirements need not only be elicited, but negotiated and prioritized. It seems that these activities are significantly different in some respects from elicitation. For example, while Calefato et al. [13] found that for requirements elicitation text-based communication is roughly as efficient as face-to-face communication, their experiments show that for requirements negotiation face-to-face is more effective. A key problem of internationalization is identified in [12], where the authors conclude that it is very difficult to determine the value of a requirement in a distributed situation. However, their conclusion might be partially due to the fact that they look at a case where the company can only indirectly communicate with the customer. In their context (market-driven development) they additionally face the problem of integrating multiple requests in a balanced way. This problem is also addressed in the context of product line engineering as scoping from different perspectives [15], [16].

Bhat and Gupta address requirements engineering in a maintenance situation [22]. They emphasize that requirements must be clustered to identify what requirements belong together and with which role they are associated to prioritize them correctly. They also emphasize the need to communicate priority back to the stakeholders. Damian emphasizes the importance and difficulty of establishing a trust relationship in a distributed setting [8]. Crnkovic et al. also mention the importance of relationship building as a success factor in the context of the many student courses they conducted [24]. In their courses they support this by enforcing intensive communication.

## 2.6 Summary

In this section, we discussed the problem of identifying requirements with customers, who are in a different country as the development organization. We introduced the term International Requirements Engineering (IRE) to emphasize this situation. We identified a number of different issues in this context, which we discussed based on existing literature from requirements engineering and global software engineering. We also used additional literature, where appropriate (in particular from software internationalization).

Overall, there is a broad range of issues that come up in the context of IRE: they impact all phases of requirements engineering from requirements elicitation over requirements analysis to negotiation and prioritization. A fundamental question here is how to organize customer interaction. This strongly influences further decisions in IRE. However, we found little discussion of this as most cases reported on single system, contract-based development with direct customer-developer interaction. The introduction of a separate sales office leads to a significant difficulty in creating close cooperation between development and customer, but it also provides major benefits to small- and medium-size companies when they aim at an international market.

Organizational issues are a very important problem in IRE. The challenge in IRE is that the possibility to influence this is rather low as the development organization has typically no direct power to make changes to the customer organization. A fundamental decision is how to organize the customer interface. Using an intermediary organization

has fundamental advantages, but also disadvantages. Thus, we assume this is only useful if the development organization deals with a large range of customers like in the case studies by Jantunen et al. [12].

Cultural differences between developer and customer organization may negatively impact the requirements engineering process. This is not only true for regional differences, but also company cultures or the cultural difference between small and large company must be taken into account. Cultural differences may easily lead to interpretations of requirements that are inadequate and hence result in unacceptable products.

We also identified context issues that give rise to specific requirements that are often hard to identify. Typically ethnographic studies are proposed for this. This reinforces the need for people exchange between both organizations and it seems to promote that more benefits are gained if the development organization visits the customer organization than vice versa.

Finally, requirements negotiation and prioritization are impacted by international development. It seems to be very important to facilitate direct communication between development and customer and among different customer groups, preferably face-to-face. If this is not possible, at least transparency of process and status must be achieved for all involved stakeholders. In particular, the multiple, distributed stakeholder needs must be clustered to adequately represent the corresponding roles.

### **3 Distributed Requirements Engineering**

In this section we focus on Distributed Requirements Engineering (DRE) as defined in Section 1. Thus, the main focus is on how to ensure that everyone in development has the same understanding of the system that must be developed and that the understanding corresponds to the customer intentions.

#### **3.1 Research Questions**

In order to identify guidelines for DRE, we identified three research questions based on the following rationale: If development is distributed, then it might become problematic to identify what information must be elicited. Also people on the development side who need to participate in the elicitation may be distributed leading to problems integrating them in the elicitation. Then, once elicited, the information must be communicated across the organization and a common understanding must be achieved. Finally, people need to act in a coordinated way based on the gathered information. This is summarized by the following questions, which provide the basis of our analysis:

1. How to organize elicitation in the context of requirements engineering?
2. How to ensure communication of requirements and a coherent vision of the system across the organization?
3. How to ensure good cooperation across organizational borders?

Many of the issues (and approaches) we found are not specific to requirements engineering. Actually, many are of general importance to distributed software development. However, we restricted our survey to issues that are explicitly raised in the context of RE.

### 3.2 Organization of Elicitation

The organization of elicitation has already been partially addressed in Section 2.2. However, our focus here is on the elicitation problems that may occur due to the distributed nature of the development and not because of distributed customer sites. This situation is characterized by the fact that the development organization is distributed and that different parts of the development organization may have different information needs and may even try to communicate with the customer simultaneously, perhaps in an uncoordinated fashion. Unfortunately, we could not find very much information on this, as most work that addresses global requirements engineering addresses the basic situation that one customer interacts with one (part of) a development organization. Probably the closest to this is the situation described by Jantunen et al. [12]. In their case study the core development organization communicates through a number of sales offices and partner companies with the customer. In this case the core elicitation function is actually handled not by the development company, but is part of the sales offices, thus, they are responsible for part of requirements engineering. The developers emphasize that through this indirection step time is lost and the integration of requirements that arrive through different channels becomes hard. Moreover, it is extremely difficult in such a situation to assess the (relative) business value of the various requirements.

Bhat et al. also provide an interesting case study [23]. They discuss the situation that both the business managers on the client side as well as the development team are distributed. This arrangement became particularly problematic as access to the business managers had to be routed through the IT-organization of the customer. While this simplified communication, as only one point of contact was needed, it was much harder to communicate with the user. In this situation also multiple parts of the developer organization communicated with different parts of the client organization. This led to disagreements among the different development teams on issues like the requirements approach or the current status of the project. The fact that multiple teams worked in parallel also led to further misunderstandings and conflicts.

In these examples, we see the issues that may occur highlighted. In both cases the authors of the studies also made some recommendations on how to deal with this situation. Jantunen et al. [12] provide some examples of how the organization aimed to address the problem by establishing a central product management and try to create direct links from there to the customers or to introduce rigorous prioritization techniques to ensure that tradeoffs among the requirements relevant to the different markets can be made. Bhat et al. [23] also discuss some proposals for improving the situation. However, while they give several recommendations, they do not clearly relate them to the situation described above. Some of the probably relevant practices are: create a shared process, this can include methods like distributed quality function deployment for requirements prioritization. Another one that is relevant to these



problems is to encourage shared responsibility through measures like increasing visibility through frequent deliverables. Finally they mention the need to create trust, e.g., through scheduling ongoing informal meetings. However, in this second case the guidelines are not as precise and it is not clear what evidence exists for them.

### 3.3 Communication of the Requirements

Communicating requirements in a distributed software development organization is important and difficult. Damian [8] discusses this intensively and identifies the following approaches to address the problems:

- Define a clear organizational structure with communication responsibilities for the distributed project.
- Establish peer-to-peer links at all management, project, and team levels across distributed sites.
- Partially synchronize inter-organizational processes and perform frequent iterations and deliveries.
- Establish cultural liaisons.
- Maintain “open communication lines” between well-defined stakeholder roles.
- Frequently inform and monitor progress on commonly defined artifacts.

We do not want to discuss these guidelines in detail. It is important to note that they all focus on enforcing communication and improving cultural understanding. Damian also mentions that modern requirements engineering tools might provide a good basis for improving this situation [8]. However, they mainly contribute towards a commonly visible status. In general, it is hard to determine the validity of these guidelines, as they rely on a meta-study and no analysis of the validity is given.

Crnkovic et al. also share the view that communication should be enforced based on their experience of several years of global software engineering lab courses [24]. However, this seems to be partially in contrast with results by Gotel and Kulkarni [25] who report that enforcing cultural understanding is not necessarily helpful. It should be added that both experiences stem from student experiments and are not necessarily representative of requirements engineering practice. This limits their external validity.

Gorschek et al. [26] propose a similar view when they contend that tools, communication and mutual understanding are an important issue. Based on their experiences in distributed projects, Bhat et al. [23] mention agreement on processes, shared culture and goals, and appropriate tools as major success factors. Laurent et al. also mention a common requirements database or a Wiki as an important factor to achieve transparency of the current status [19]. Hagge and Lappe also mention standardized documentation as an important factor in [27]. This was identified based on an effort to gather requirements process patterns. Similar advice is given by Berenbach and Wolf, who propose an approach to requirements documentation, which emphasizes the use of traceability among different requirements (sub-)models such as feature lists or use cases and a common requirements database [21]. This approach was successfully applied in several case studies at Siemens.

Nicholson and Sahay point out that cultural issues can also strongly influence requirements communication [9]. In a case study of shared development between England and India they identified cultural differences as major obstacles in the area of requirements engineering. For example, they mention that Indians tended to take all requirements without questioning them, while the site in England expected that requirements would be questioned. This led to significant misunderstandings wrt. to the requirements.

### **3.4 Ensure Cooperation across Organizational Borders**

Requirements engineering requires intensive cooperation and communication. On the one hand this needs to happen in relation to the customer, but on the other hand also among requirements engineers and with other functions in the software development organization like architects or testers.

In distributed organizations, there are significant inhibitors to this: this is on the one hand due to typical problems of regional diversity, but in addition, as Gumm finds [28], organizational distribution can be an even larger challenge than physical distribution. In requirements engineering we find that engineers can often be assigned to different countries, customer groups, key accounts, etc.; moreover, in some organizational setups architects and/or testers might belong to different organizational groups, again. Thus, we recognize that these organizational borders might pose significant problems in their own right with respect to the dissemination of requirements knowledge and a coherent product vision throughout an organization.

So, how can we ensure that despite such problems a fluent cooperation is ensured? Crnkovic et al. emphasize the importance of creating intensive communication links early on in a project [24]. Damian mentions the need to take different languages into account [8]. In the internal communication sometimes distributed organizations attempt to address this by introducing a reference language, often English. However, it needs to be taken into account that using a non-native language will usually introduce significant overhead for people. Further, while Damian focuses here on languages like English vs. Chinese, different interpretations may also exist within the same country, in different organizations (e.g., requirements engineers vs. architects), or even among different requirements engineers (especially after a company merger), all this may negatively impact cooperation and communication.

Damian also mentions the need to recognize and understand differences among the cooperating groups [8], a viewpoint that is supported by Crnkovic et al. [24]. In order to achieve this, they propose to create cultural liaisons, i.e., people from one background switch to a different organization to help bridge the gap. Another strategy is to establish what Damian calls peer-to-peer links at all levels across distributed sites. In this case, on both sides a person gets the dedicated role to establish relationships with the corresponding role at the other site.

### 3.5 Summary

DRE requires adapting the requirements engineering process and activities, the organizational structure and most importantly the mindset of people. It is critical that the corresponding problems are explicitly recognized and addressed in order to achieve a successful requirements process. Surprisingly, especially the organization of elicitation has been addressed rather little in the scientific literature (e.g., the role of sales organizations in the overall requirements engineering structure). More research would certainly be beneficial here.

Important guidelines are to explicitly recognize and address distribution difficulties and challenges, especially language issues, cultural issues, and geographic distribution. Major approaches to achieve this are proxies (embedded personal from the other organization), intensive communication (e.g., visits, scheduled interactions, etc.), and most importantly direct communication. Also, already some models emerge that aim to give prescriptive guidance on how to set up an organization in a way that supports good global communication like the global teaming model [29].

A somewhat unclear point is, however, the importance of addressing cultural distance as different authors give this different importance. We think, based on the existing literature that it is important to address cultural issues, but this can in no way replace other measures and the technical content must remain in the foreground. We were also surprised that the use of tools (e.g., internal requirements databases where everyone has access) and similar measures that aim at supporting continuous communication did not come up more often.

## 4 Conclusion

Global requirements engineering is a challenge and will remain a challenge for the future. It is inherently more complex and difficult than requirements engineering in closely collocated settings. However, given today's situation of globalized development, practitioners must cope with these problems. Here, we tried to condense some current knowledge on requirements engineering in global software engineering.

We decided to separate our analysis into two main parts: International Requirements Engineering and Distributed Requirements Engineering. While the first addresses customer-oriented interaction, the second focuses on development-internal interaction. While the corresponding issues overlap, also rather different issues came up. We could identify a number of specific solutions that have been identified in literature, however, we also needed several times to concede that evidence is so far inconclusive. Often, also the specific details of the support of a proposal are not given precisely enough. Most studies cover only a single case study or a small number of case studies. Often, it is also not clear how reliable the case study information is.

Further, in some cases, we have only conflicting information leading to different recommendations. It is at this point unclear how to resolve these conflicts. We believe this can only be achieved by further detailed studies which also explicitly take into account the variations in contexts for global requirements engineering (for example an approach that is very adequate in one of the contexts identified in Section 2.2 is not

necessarily appropriate in another one). Unfortunately, existing studies often do not provide this information. Thus, it is not possible to derive the additional discriminatory factors. Thus, while the information provided in this report already gives a good overview both to researchers and practitioners, we conclude that more studies must be performed, also on a more detailed level to provide a basis for furthering in a reliable way the existing state of the art in global requirements engineering.

**Acknowledgment.** The work on this publication was partially supported by the Software Engineering Center (SEC), NIPA, South Korea. The author was also partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq).

## References

- [1] Cheng, B., Atlee, J.: Research directions in requirements engineering. In: Future of Software Engineering (FOSE 2007) Part of International Conference on Software Engineering, pp. 285–303 (2007)
- [2] Herbsleb, J.: Global software engineering: The future of socio-technical coordination. In: Future of Software Engineering (FOSE 2007), Part of International Conference on Software Engineering, pp. 188–198 (2007)
- [3] Esselink, B.: A Practical Guide to Localization. John Benjamins (2000)
- [4] Phillips, A.: Internationalization: An introduction part II: Enabling. In: Tutorial at the Internationalization and Unicode Conference (2009)
- [5] Watson, G.L.: A Quick Guide to Software Internationalization Issues. Kindle E-Book (2010)
- [6] VDMA. Software-Internationalisierung. VDMA (2009) (in German)
- [7] Schmid, K.: Requirements Engineering for Globalization (RE4G): Understanding the Issues, Report for the Software Engineering Center (SEC), NIPA, South Korea (2012)
- [8] Damian, D.: Stakeholders in global requirements engineering: Lessons learned from practice. *IEEE Software* 24(2), 21–27 (2007)
- [9] Nicholson, B., Sahay, S.: Embedded Knowledge and Offshore Software Development. Information and Organization, pp. 329–365. Elsevier (2004)
- [10] Hanisch, J., Corbitt, B.: Requirements engineering during global software development: Some impediments to the requirements engineering process — a case study. In: 12th European Conference on Information Systems (ECIS), pp. 628–640 (2004)
- [11] Prikladnicki, R., Audy, J., Damian, D., de Oliveira, T.: Distributed Software Development: Practices and challenges in different business strategies of offshoring and onshoring. In: 2nd International Conference on Global Software Engineering, pp. 262–274 (2007)
- [12] Jantunen, S., Smolander, K., Gause, D.: How internationalization of a product changes requirements engineering activities: An exploratory study. In: 15th International Requirements Engineering Conference, pp. 163–172 (2007)
- [13] Calefato, F., Damian, D., Lanubile, F.: An empirical investigation on text-based communication in distributed requirements workshops. In: 2nd International Conference on Global Software Engineering, pp. 3–11 (2007)

- [14] Brockmann, P., Thaumüller, T.: Cultural aspects of global requirements engineering: An empirical chinese-german case study. In: 4th International Conference on Global Software Engineering, pp. 353–357 (2009)
- [15] John, I., Eisenbarth, M.: A decade of scoping — a survey. In: 13th International Conference on Software Product Lines, pp. 31–40 (2009)
- [16] Helferich, A., Schmid, K., Herzwurm, G.: Reconciling marketed and engineered software product lines. In: 10th International Software Product Line Conference, pp. 23–27 (2006)
- [17] Maiden, N., Rugg, G.: ACRE: Selecting Methods for Requirements Acquisition. *Software Engineering Journal* 11(3), 183–192 (1996)
- [18] Bhatia, M., Vasal, A.: Localisation and Requirement Engineering in Context to Indian Scenario. In: 15th IEEE International Requirements Engineering Conference, pp. 393–394 (2007)
- [19] Laurent, P., Mäder, P., Cleland-Huang, J., Steele, A.: A Taxonomy and Visual Notation for Modeling Globally Distributed Requirements Engineering Projects. In: International Conference on Global Software Engineering, pp. 35–44 (2010)
- [20] Stapel, K., Knauss, E., Schneider, K., Zazwoprka, N.: FLOW Mapping: Planning and Managing Communication in Distributed Teams. In: 6th International Conference on Global Software Engineering, pp. 190–199 (2011)
- [21] Berenbach, B., Wolf, T.: A unified requirements model; integrating features, use cases, requirements, requirements analysis and hazard analysis. In: 2nd International Conference on Global Software Engineering, pp. 197–203 (2007)
- [22] Bhat, J., Gupta, M.: Enhancing Requirement Stakeholder Satisfaction during Far-shore Maintenance of Custom Developed Software using Shift-Pattern Model. In: 2nd International Conference on Requirements Engineering, pp. 322–327 (2007)
- [23] Bhat, J., Gupta, M., Murthy, S.: Overcoming requirements engineering challenges: Lessons from offshore outsourcing. *IEEE Software* 23(5), 38–44 (2006)
- [24] Crnkovic, I., Bosnic, I., Žagar, M.: Ten tips to succeed in global software engineering education. In: International Conference on Software Engineering, Software Engineering Education, pp. 101–105 (2012)
- [25] Gotel, O., Kulkarni, V., Say, M., Scharff, C., Sunetnanta, T.: Quality indicators on global software development projects: Does “getting to know you” really matter? In: 4th IEEE International Conference on Global Software Engineering, pp. 3–7 (2009)
- [26] Gorschek, T., Fricker, S., Felt, R., Torkar, R., Wohlin, C., Mattsson, M.: 1st international global requirements engineering workshop (GREW 2007). *ACM SIGSOFT Software Engineering Notes* 33(2), 29–32 (2008)
- [27] Hage, L., Lappe, K.: Sharing requirements engineering experience using patterns. *IEEE Software* 22(1), 24–31 (2005)
- [28] Gumm, D.: Distribution dimensions in software development projects: A taxonomy. *IEEE Software* 23(5), 45–51 (2006)
- [29] Beecham, S., Noll, J., Richardson, I., Dhungana, D.: A Decision Support System for Global Software Development. In: 6th International Conference on Global Software Engineering, Workshops, pp. 48–53 (2011)

# Automated Feature Identification in Web Applications

Sarunas Marciuska, Cigdem Gencel, and Pekka Abrahamsson

Free University of Bolzano-Bozen

Marciuska@inf.unibz.it, {Cigdem.Gencel,Pekka.Abrahamsson}@unibz.it

**Abstract.** Market-driven software intensive product development companies have been more and more experiencing the problem of feature expansion over time. Product managers face the challenge of identifying and locating the high value features in an application and weeding out the ones of low value from the next releases. Currently, there are few methods and tools that deal with feature identification and they address the problem only partially. Therefore, there is an urgent need of methods and tools that would enable systematic feature reduction to resolve issues resulting from feature creep. This paper presents an approach and an associated tool to automate feature identification for web applications. For empirical validation, a multiple case study was conducted using three well known web applications: Youtube, Google and BBC. The results indicate that there is a good potential for automating feature identification in web applications.

**Keywords:** feature creep, feature expansion, feature identification, feature reduction, feature location, feature monitoring, software bloat.

## 1 Introduction

Feature creep [1,2] (i.e. addition or expansion of features over time) has become a significant challenge for market-driven software intensive product development. Today's software intensive products are overloaded with features, which have led to an uncontrollable growth of size and complexity. A recent study [3] revealed that most of the software products contain from 30 to 50 percent of features that have no or marginal value.

One of the major consequences of feature creep is feature fatigue [4], when a product becomes too complex and has too many low value features. Users then usually switch to other, simpler products. Moreover, feature creep can also result in software bloat [5] that makes a computer application slower, which requires higher hardware capacities, and increases the cost of maintenance. One of the most recent example of software bloat is Nokia Symbian 60 smartphone platform [6]. The system grew so much that it was too expensive to maintain it, and therefore it was abandoned.

Currently, lean start-up [7] software business development methodology tackles the feature creep problem by finding a minimum viable product that contains

only essential and the most valuable features. However, not all lean-start up companies start development from scratch and can easily determine the minimum viable product as they already have complex systems. For example, by understanding how users are using the features, a company might discover that a set of features maintained by the company are actually not too much valuable for their customers. Thus, decision makers could analyse if removal of such features would bring any long term benefits for the company as there would be less features to maintain. Therefore, there is a need to monitor and identify the features that are not too valuable in order to systematically remove them from the product [6].

To start with the feature reduction process it is crucial to identify the complete set of features. Features should be identified automatically in order to reduce feature reduction process cost. After identifying the features, they can be monitored by the company to detect how their values change in time. For example, feature usage monitoring could indicate that the usage of some features is decreasing, and thus such features might become candidates for feature reduction. In our previous work, we showed that low feature usage is a good indicator for the features that are potentially losing customer value [8].

There exists some approaches that tackle with feature monitoring and identification problem (see Section 2). For example, a number of methods aim to locate features from the source code [9], but they still lack precision. Others aim to monitor system changes by observing user activity [10,11]. However, such approaches collect too much irrelevant information (i.e. random mouse clicks, mouse scrolling, all key strokes) and fail to monitor the system on a feature level. Another set of approaches [12,13] monitor system usage on too high level of abstraction (i.e. page usage, but not a feature level usage).

The focus of this paper is to address the problem of automated feature identification for web applications for feature reduction purposes. We set our research questions as follows:

- RQ1 : What constitutes *a feature* in web applications for the purposes of feature reduction?
- RQ2 : How well features could be identified in an automated manner in the context of web applications?

Here, we present our approach and an associated tool that identifies elements of a web application (based on HTML5 technologies) that correspond to features. To this end, first, we investigated definitions of *a feature* by making a literature survey. Then, we defined formally what constitutes a feature in web applications. Finally, we developed a tool, which implements the rules for automatically identifying features in web applications.

To evaluate the performance of our tool, we conducted a multiple case study. We selected three well known web sites as the cases: Google, BBC, and Youtube. The features for these web applications were first identified manually by the participants of the case study and then in an automated way using our tool. At the end, we compared the results using two operational measures: precision and recall [14].

The rest of the paper is structured as follows: Section 2 presents the related work. In Section 3, we elaborate on definitions of feature in the literature. Then, by formalising the definition of a feature for web applications, we describe our approach for feature identification. Section 4 presents the case study and the results. In Section 5, we discuss threats to validity for this study. Finally, Section 6 concludes the work and presents future directions.

## 2 Related Work

The feature location field aims to locate features and their dependencies in the code of a software system. A recent systematic literature review on feature location [9] categorizes the existing techniques in four groups: static, dynamic, textual, and historical.

Static feature location techniques [15,16,17] use static analysis to locate features and their dependencies from the source code. The results present detailed information such as variable, class, method names and relations between them. The main advantage of these approaches is that they do not require executing the system in order to collect the information. However, they require to have an access to the source code. Moreover, static analysis generate a set of features dependent on the source code, so they involve a lot of noise (i.e. variable names that do not represent features).

Dynamic feature location techniques [18,19,20] use dynamic analysis to locate the features during runtime. As an input this technique requires a set of features, which has to be mapped to source code elements of the system (i.e. variables, methods, classes). As a result, a dependency graph among given features is generated. The main advantage of these techniques is that it shows the parts of the code called during the execution time. However, dynamic feature location techniques rely on the user predefined initial feature set, so they cannot generate a complete features set beforehand.

Textual feature location techniques [21,22,23,24,25] examine the textual parts of the code to locate features. As an input this technique requires to define a query with feature descriptions. Later, the method uses information retrieval and language processing techniques to check the variables, classes, method names, and comments to locate them. The main advantage of these techniques is that they map features to code. However, like dynamic feature location technique, it requires a predefined feature set with their descriptions.

Historical feature location techniques [26,27] use information from software repositories to locate features. The idea is to query features from comments, and then associate them to the lines that were changed in respective code commit. The main advantage of these techniques is that they can map features to a very low granularity of the source code, that is to exact lines. However, as in dynamic and textual approaches, this technique cannot determine a complete features set in an automated manner. In the next section, we present our approach to address this issue.



### 3 A Method for Automated Feature Identification in Web Applications

There are a number of definitions in literature for what constitutes a feature. Below, we elaborate on some of these definitions and then present our formal definition for a *feature* for web applications.

#### 3.1 Feature Definitions

The definition of ‘a feature’ varies widely depending on the area and purpose of the study. Classen et al. [28] made a detailed analysis on different definitions of a feature in the contexts of requirements engineering, software product lines and feature oriented software development. Below there are some of the definitions from this study:

- “A feature represents an aspect valuable to the customer” [29].
- “A feature is a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems” [30].
- “Features are distinctively identifiable functional abstractions that must be implemented, tested, delivered, and maintained” [31].
- “A logical unit of behaviour specified by a set of functional and non-functional requirements” [32].
- “A product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements” [33].
- “An elaboration or augmentation of an entity(s) that introduces a new service, capability or relationship” [34].
- “An increment in product functionality” [35].
- “A structure that extends and modifies the structure of a given program in order to satisfy a stakeholder’s requirement, to implement and encapsulate a design decision, and to offer a configuration option” [36].

Classen et al. [28] claimed that there is a need to have a definition, which covers all kinds of requirements, domain properties and specifications. Thus, they provided their own definition as: “A feature is a triplet,  $f = (R, W, S)$ , where R represents the requirements the feature satisfies, W the assumptions the feature takes about its environment and S its specification” [28].

However, most of the aforementioned definitions are either very generic or vague in order to be used for automatic feature identification, because they rely on subjective human evaluation (features may be interpreted differently by different people). For example, the aforementioned feature definition by Classen et al. [28] can identify different features for the same system depending on how the requirements specification document is written.

To the best of our knowledge, only one definition provided by Eisenbarth et al. [37] removes the subjective human element. It has been cited by most of the work done in feature location area. The authors define a feature as “a realized functional requirement of a system (i.e. is an observable unit of behaviour of a system triggered by the user)”.

This definition makes it clear that 1) features are identified based on events triggered by users, and 2) they realise functional requirements. For example, a case where a user has to enter his email, password and press the login button in order to login is different from the case where system remembers his credentials and he just has to press the login button in order to login. In the first scenario, three features are identified, while in the second, only one even though the final state was the same (that is, the user logged into the system).

These two scenarios might be interpreted differently by different people depending on the abstraction level how they perceive what a feature is. In this study, we focus on the lowest granularity level features in order to be able to identify them automatically. Then, our approach allows decision makers to group similar features to represent them at higher granularity levels (see [38] for details).

Moreover, according to the definition non-functional requirements (such as performance requirements) are not features, but they might affect how features are implemented. In addition, some of them might evolve into functional requirements (such as usability requirements) during implementation, but then these would be identified based on the events triggered by users.

In this study, we extended the definition to include the features that are triggered not only by users, but by other systems as well (i.e. web services), since in some cases they can also be considered as users. We used this version of the definition when conducting our case study to evaluate the performance of the tool in automatically identifying features against manual identification.

### 3.2 A Formal Definition for a Feature in Web Applications

In the context of web applications, features that relate to functional requirements, are visible for system users through a web browser. This means that to identify features, it is not necessary to analyse a complex server side implementation of a system, which can be developed using different programming languages (i.e. Java, PHP, C). Therefore, in this study we focus only on the client side analysis. Currently, vast majority of the client side web based applications are designed using HTML5, JavaScript, and CSS technologies.

According to our feature definition, there are limited possibilities how a feature could be implemented using the aforementioned technologies. Therefore, a feature in a web application is: 1) a specific HTML5 element, which changes system's state that can be observed when event is triggered; 2) any HTML5 element, which has attached event that changes system's state in observable way.

In HTML5 there are only three elements that can be considered features without having any event attached to them [39]. The elements are called anchor, input, and textarea. In addition, input element should not be of a type "hidden", because in such a way it used as a variable and is not a feature. The remaining elements become features if one of the four event types is attached to them: a mouse event (see Table 1), a keyboard event (see Table 2), a frame and object event (see Table 3), or a form event (see Table 4).

**Table 1.** Mouse Events

<b>Event</b>	<b>Description</b>
onclick	Event occurs when the user clicks on an element
ondblclick	Event occurs when the user double-clicks on an element
onmousedown	Event occurs when a user presses a mouse button over an element
onmousemove	Event occurs when the pointer is moving while it is over an element
onmouseover	Event occurs when the pointer is moved onto an element
onmouseout	Event occurs when a user moves the mouse pointer out of an element
onmouseup	Event occurs when a user releases a mouse button over an element

**Table 2.** Keyboard Events

<b>Event</b>	<b>Description</b>
onkeydown	Event occurs when the user is pressing a key
onkeypress	Event occurs when the user presses a key
onkeyup	Event occurs when the user releases a key

**Table 3.** Frame and Object Events

<b>Event</b>	<b>Description</b>
onabort	Event occurs when an image is stopped from loading before completely loaded (for object)
onerror	Event occurs when an image does not load properly
onload	Event occurs when a document, frameset, or object has been loaded
onresize	Event occurs when a document view is resized
onscroll	Event occurs when a document view is scrolled
onunload	Event occurs once a page has unloaded (for body and frameset)

**Table 4.** Form Events

<b>Event</b>	<b>Description</b>
onblur	Event occurs when a form element loses focus
onchange	Event occurs when the content of a form element, the selection, or the checked state have changed (for input, select, and textarea)
onfocus	Event occurs when an element gets focus (for label, input, select, textarea, and button)
onreset	Event occurs when a form is reset
onselect	Event occurs when a user selects some text (for input and textarea)
onsubmit	Event occurs when a form is submitted

### 3.3 An Algorithm for Automated Feature Identification

We designed an algorithm to identify features in web applications. The main idea of the algorithm is to parse the Document Object Model (DOM) and to select all HTML5 elements that correspond to features, or all HTML5 elements that have one of the four aforementioned events assigned to them. The pseudo code of the algorithm is given below (see Algorithm 1). The algorithm identifies features on a low level of abstraction, which later can be manually increased by grouping features using Feature Usage Diagram [38].

---

#### Algorithm 1. A Complete Feature Set Identification

---

```

1: result_set  $\leftarrow \emptyset$ 
2: event_set  $\leftarrow \{\text{all HTML5 events}\}$ 
3: feature_set  $\leftarrow \{\text{all HTML5 features}\}$ 
4: DOM  $\leftarrow \{\text{DOM of interest}\}$ 
5: element  $\leftarrow \text{DOM.first}$ 
6: while element  $\neq \emptyset$  do
7:   if element IN feature_set then
8:     result_set  $\leftarrow \text{element}$ 
9:   else if element.events IN event_set then
10:    result_set  $\leftarrow \text{element}$ 
11:   end if
12:   element  $\leftarrow \text{DOM.next}$ 
13: end while
14: return result_set

```

---

Algorithm 1 requires three input parameters: HTML5 event set of interest (*event\_set*), HTML5 element set that represent features (*feature\_set*), and the DOM of a website of interest (*DOM*).

The algorithm iterates through all elements of DOM and checks whether a selected element (*element*) is one of the elements from *feature\_set*, or it has one of the events from the *event\_set* attached. If it is the case the *element* is added to a result set (*result\_set*), which is returned after loop iterations are finished.

The runtime complexity of the algorithm is  $O(n(1 + m)) = O(n^2)$ . It takes  $O(n)$  times to iterate over  $n$  elements in DOM. Assuming that *feature\_set* and *event\_set* are implemented using hash table, then to check if *feature\_set* contains *element* takes  $O(1)$  time. Therefore, to iterate over all *element.events* and to check if they are in *event\_set* it takes  $O(m)$  time.

We implemented this algorithm using the JavaScript programming language. The code is available as the external JavaScript library on the following website: <http://www.featurereduction.org>. Initially, it has a *feature\_set* predefined, which contains anchor, input, and textarea elements. Since the vast majority of the websites are built using just few events (i.e. onclick, onkeystroke), the tool allows user to select custom *event\_set* from a complete list of HTML5 events: a mouse event (see Table 1), a keyboard event (see Table 2), a frame and object

event (see Table 3), or a form event (see Table 4). Then, a custom JavaScript library is generated, which contains the selected event set.

There are two ways to use this tool when identifying the set of features for a web application of interest: 1) put the aforementioned library directly on a website where it is hosted; 2) download the website and include the library in the downloaded version of it. Obviously, the latter approach is able to identify features only in the downloaded pages. It would not work on the pages that are dynamically generated or cannot be downloaded. Nevertheless, in a normal use case scenario, companies have a full access to their websites and can apply the first method.

Finally, the following information is presented as the result:

1. the full path to an element,
2. the title attribute of the element,
3. the name attribute of the element,
4. the id attribute of the element,
5. the value attribute of the element,
6. the text field of the element.

The full path to the element helps to find it in the DOM. We assumed that the title, name, id, value attributes and text field, if present, usually contain human understandable information, which contains a description of the feature. We test this assumption in our case study, which we present in the next section.

## 4 Case Study

We conducted a multiple-case study according to [40] to evaluate whether our tool performs well in identifying the features in web applications with respect to manual identification.

We selected three web applications for the case study: Google, BBC, and Youtube. The selected websites cover a wide range of daily used applications having search, news and video streaming features. One major reason why we chose these websites was that they are used in different contexts, and thereby, we could test our tool to identify variety of features developed for different contexts. Another reason was that most readers would be familiar with these applications as they are widely used (according to the Alexa traffic rating [41] they are among top 10 most popular websites in Great Britain). Finally, as these websites are not customised based on user demographics, the set of features would be the same for all participants of the case study and hence the results could be comparable.

In addition to the first author of this paper, 9 subjects, who are frequent users of the case applications, participated in this study. This is a convenience sample, where the participants have varying backgrounds (i.e. medicine, design, computer science) and sufficient knowledge and experience in using web applications.

## 4.1 Case Study Conduct

Before the case study, we downloaded the main web pages of the case websites to make sure that all participants have the same version of the website. After that, we distributed the downloaded pages as executable systems to the participants, introduced the participants our formal feature definition and then asked them to manually identify the features. The participants were given as much time as they need to complete the task.

As our purpose in this case study was to identify manually all the features of the selected applications in order to compare them to the results of our automated tool, we needed to have a correct and complete set of features that were identified manually. Therefore, after the set of features identified by the participants converged to a one final set, we considered this as the complete set and we stopped asking more people to participate in the case study. In parallel, the first author of this paper also identified the set of features in the same way as the participants to cross-check the features identified by the participants.

Later, we asked the participants to write down the information about features in a text file. Finally, we used our tool to identify the features from the downloaded web pages. The results from the tool were printed out to the console of a web browser.

To evaluate the performance of our tool we used operational measures: *precision* and *recall*. *Precision* indicates whether the tool collects elements that are not features and *recall* indicates whether the tool determines the complete features set.

As a pre-analysis, we verified with all the participants each feature, which they identified and recorded. We excluded the data provided by two of the participants as they provided too generalized outputs (i.e. “menu widget features”). They mentioned that it was too much time consuming task. Therefore, they could not provide a complete feature set.

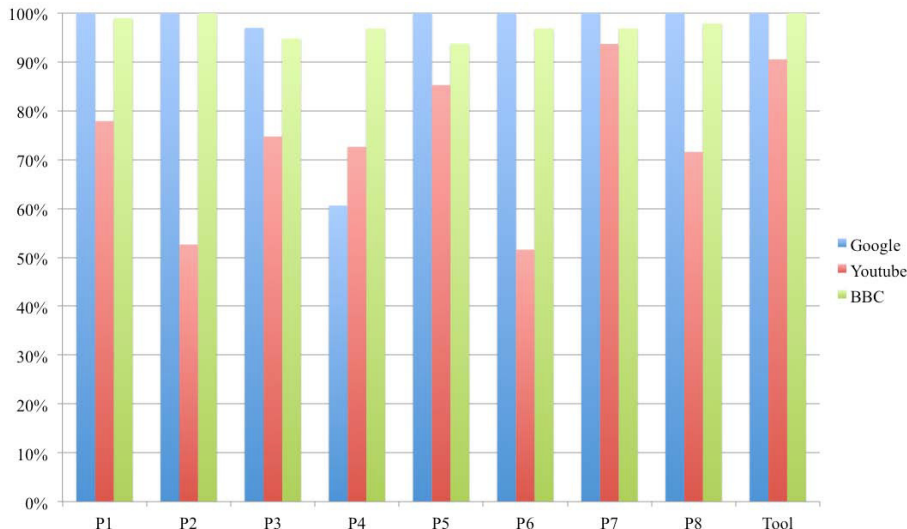
To compute the precision and recall, we compared features identified by the participants and the tool to the complete feature set. In addition, we compared the data collected by the tool with the feature description provided by the participants.

Then, we made informal interviews with the participants to receive feedback on the features, which were identified by them, but not identified by the tool, and vice versa.

## 4.2 Results and Analysis

The results showed that there were both *visible features* and *hidden features* in the applications of the case study. Visible features are the ones that participants were expected to detect manually through a web browser without using other tools or looking at the code of a given website. Google had 33, Youtube had 80 and BBC had 96 visible features in total.

The analysis of the results showed that the precision was 100% in both manual and automatic identification of visible features. This means that both the tool



**Fig. 1.** Case Study Results – Recall

and the participants could identify the elements that correspond to features. Figure 1 presents the results of the recall measure.

The recall measure to automatically identify visible features by the tool was 100% for the Google website, 91% for the Youtube website, and 100% for the BBC website. When we investigated why our tool could not identify all the features in the Youtube website, we saw that this is due to Flash technologies used in this website. This showed that our tool has some technological limitations when identifying features.

When we compared the performance of our tool to manual identification of the features, we saw that our tool overperforms the participants in most of the cases. We checked with the subjects why they failed to identify all the visible features. Here are the reasons we found after our interviews:

- *Missed.* Some of the features were simply missed by the participants due to carelessness in manual identification. It was understood that the great majority of such features were not visible instantly in the main page as they were placed in drop down, or pull down lists.
- *Redundant.* The participants reported that there were some features that had the same functionality as the ones, which they had already identified. Therefore, they didn't add these features in the list. For example, the link on the icon of the commentator and the link on the name of the commentator in Youtube leads to the same page. Therefore, some participants identified only one feature instead of two in this case.

Finally, we compared the representation of the results provided by our tool and the results provided by the participants. We noticed that most of the participants

used the name of a feature from a website to describe its functionality. After analyzing the tool results, we found out that this information was stored in text, value, or title attributes. The rest of the attributes (id attribute and name attribute) were not useful for this task. To describe the location of a feature participants indicated the main container where a selected feature belongs to. On the other hand, the full path of the feature provides even more detailed information as the one provided by the participants.

One major result of the case study was that our tool could identify a high number of *hidden features*, which could not be detected by the participants manually. In the Google website, our tool identified 15, in the Youtube website 8, and in the BBC website 126 hidden features.

After analyzing the source code of such features, we found out that most of these were related to personal user preferences, or the device that is used to open the website. For example, if a user opens a website using a mobile phone or tablet, then the feature set is adapted accordingly. In addition, there were a few features, which did not seem to add any value (such as 3 hidden textarea elements in the main google website that are not necessary). Those features might be important for the developers of the website for some reason, or might be just obsolete features, which hang in the code without any specific purpose. This shows that our tool has the potential to indicate different kind of features, which could be removed after developers assess them.

## 5 Threats to Validity

We discuss the validity threats of this study according to the categorization suggested by Runeson and Host for case studies [42]: 1) Construct validity, 2) Internal validity, 3) External validity, and 4) Reliability.

*Construct Validity.* Construct validity refers to what extent the operational measures represent what is investigated according to the research questions. One validity threat could have been related to our definition for feature for web applications as well as how it is interpreted. Therefore, we first built our definition upon one of the widely cited definition in the feature location research area [37]. We also kept the granularity level of a feature at the lowest level so that it would be possible to group related features if one needs to represent them at higher levels.

Furthermore, we formalised our definition in order to be able to implement an algorithm to automate feature identification. This in turn also helped in avoiding subjective misinterpretation of the definition by the participants when they were identifying the features manually. One of the operational measures used in this study was precision. As the precision values for both manual and automatic identification of visible features were 100%, we also conclude that all elements identified as features with the tool comply with our definition.

Another possible validity threat could have been due to making a mistake in identifying a complete feature set as the second operational measure; recall, was



calculated based on this figure. To mitigate this threat, first, one of the authors of this study manually identified the features. Then, we observed whether the manually identified features converge to the one final set as we receive input from more participants. If participants identified new features, which were not present in the initial complete feature set, then we extended it with those features. We stopped involving more subjects in a manual feature identification when we saw that adding more participants was not adding any new information. Therefore, we believe that this set should reflect the complete set.

*Internal Validity.* Internal validity concerns the causality relation between the treatment and the outcome, and whether the results do follow from the data. In this study, we evaluated the performance of our method and tool in comparison to manual identification of features. One validity threat could have been if the participants did not have enough knowledge about selected websites to perform the task well. To mitigate this task we chose the participants, who were existing users of the selected websites. Furthermore, we purposefully chose the case applications that are frequently used. In this way, we believe that the subjects had sufficient knowledge about the features of the websites. In fact, all participants showed similar performance for all three cases.

*External Validity.* External validity refers to what extent it is possible to generalize the findings to different or similar contexts. We designed the case study as a multiple-case study where we used three different web applications. Therefore, we could evaluate the method and the tool developed in this study for a variety of features coming from different web applications. However, one threat, which we could not totally avoid, might have occurred due the technology used when developing the case applications. We observed that our tool has some technological limitations in identifying the features. There is a possibility that the results could have been affected if other case applications, which use very different technologies, had been selected. However, we believe that our results can be generalizable to some extent, as we replicated the study using three different web applications developed for different contexts using different technologies. Still, there is a need to test the method and the tool for very different web applications. Furthermore, we could not evaluate our approach for the cases where decision makers prefer to group features and represent them at higher granularity levels. Our results are also not generalizable for other systems, which are not web applications (i.e. desktop applications). The goal and the scope of this study was to design an approach to automatically identify features in web applications. Therefore, there is a need to define what a feature constitutes for other application types. Then, algorithms and/or rules could be created to identify a complete feature set.

*Reliability.* Reliability reflects to what extent the data and the analysis depend on the specific researchers. We used two objective operational measures in this study: precision and recall. Therefore, we do not see any validity threat in interpreting them. However, one validity threat could have been the interpretation of

the feature set descriptions provided by the participants. To mitigate this treat, we verified with each participant what we understood from their inputs.

## 6 Conclusion and Future Work

In this paper, we presented an approach and an associated tool for automating features identification in web applications. The results of the case study showed that our approach has a significant potential to identify features in web applications.

Furthermore, we found out that our tool is also able to identify hidden features in addition to visible ones, which could not be identified manually by users. Some of these features appeared to be with no significant value, that is they might be candidates to be removed from the system. Therefore, we see this method as a first step towards automatic feature reduction process.

Moreover, the results of the case study showed that manual feature identification is prone to human mistakes even for websites with a relatively small number of features. Therefore, we conclude that development of such a tool provides more benefits when used to identify features in big systems with complex structures and high number of features.

As a future work, we plan to investigate ways to capture features implemented by other technologies such as Flash. We will analyze if similar approach can be applied in desktop applications. Furthermore, we will explore the ways how a complete feature set can be visualized. We plan to investigate how the relations between features can be detected automatically.

The automatic feature identification is the first step towards automatic feature usage monitoring. We plan to explore how such information can be used to improve product. For example, it would be interesting to understand if relocation of features can increase the usage and bring more value for the company.

**Acknowledgment.** The authors would like to thank the participants of the case study who provided their precious time and effort.

## References

1. Elliott, B.: Anything is possible: Managing feature creep in an innovation rich environment. In: Engineering Management Conference, pp. 304–307. IEEE (2007)
2. Davis, F.D., Venkatesh, V.: Toward preprototype user acceptance testing of new information systems: implications for software project management. *IEEE Transactions on Engineering Management* 51(1) (2004)
3. Ebert, C., Dumke, R.: *Software Measurement*. Springer (2007)
4. Rust, R.T., Thompson, D.V., Hamilton, R.W.: Defeating feature fatigue. *Harvard Business Review* 84(2), 98–107 (2006)
5. Xu, G., Mitchell, N., Arnold, M., Rountev, A., Sevitsky, G.: Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (2010)

6. Ebert, C., Abrahamsson, P., Oza, N.: Lean Software Development. *IEEE Software*, 22–25 (October 2012)
7. Ries, E.: The Lean Startup: How Today’s Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses. *Journal of Product Innovation Management* (2011)
8. Marcuska, S., Gencel, C., Abrahamsson, P.: Exploring How Feature Usage Relates to Customer Perceived Value: A Case Study in a Startup Company. In: Herzwurm, G., Margaria, T. (eds.) *ICSOB 2013. LNBIP*, vol. 150, pp. 166–177. Springer, Heidelberg (2013)
9. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software Maintenance and Evolution: Research and Practice* (2011)
10. Atterer, R., Wnuk, M., Schmidt, A.: Knowing the user’s every move: user activity tracking for website usability evaluation and implicit interaction. In: *Proceedings of the International Conference on World Wide Web*, p. 203 (2006)
11. Microsoft Spy++, [http://msdn.microsoft.com/en-us/library/aa264396\(v=vs.60\).aspx](http://msdn.microsoft.com/en-us/library/aa264396(v=vs.60).aspx) (last visited on March 31, 2013)
12. OpenSpan Desktop Analytics, [http://www.openspan.com/products/desktop\\_analytics](http://www.openspan.com/products/desktop_analytics) (last visited on March 31, 2013)
13. Google Analytics, <http://www.google.com/analytics> (last visited on March 31, 2013)
14. Manning, C.D., Raghavan, P., Schütze, H.: *Introduction to information retrieval*. Cambridge University Press, Cambridge (2008)
15. Chen, K., Rajlich, V.: Case Study of Feature Location Using Dependence Graph. In: *Proceedings of 8th IEEE International Workshop on Program Comprehension*, pp. 241–249 (2000)
16. Robillard, M.P., Murphy, G.C.: Concern Graphs: Finding and describing concerns using structural program dependencies. In: *Proceedings of International Conference on Software Engineering*, pp. 406–416 (2002)
17. Trifu, M.: Using Dataflow Information for Concern Identification in Object-Oriented Software Systems. In: *Proceedings of European Conference on Software Maintenance and Reengineering*, pp. 193–202 (2008)
18. Eisenberg, A.D., De Volder, K.: Dynamic Feature Traces: Finding Features in Unfamiliar Code. In: *Proceedings of 21st IEEE International Conference on Software Maintenance*, Budapest, Hungary, pp. 337–346 (2005)
19. Bohnet, J., Voigt, S., Dollner, J.: Locating and Understanding Features of Complex Software Systems by Synchronizing Time, Collaboration and Code-Focused Views on Execution Traces. In: *Proceedings of 16th IEEE International Conference on Program Comprehension*, pp. 268–271 (2008)
20. Edwards, D., Wilde, N., Simmons, S., Golden, E.: Instrumenting Time-Sensitive Software for Feature Location. In: *Proceedings of International Conference on Program Comprehension*, pp. 130–137 (2009)
21. Petrenko, M., Rajlich, V., Vanciu, R.: Partial Domain Comprehension in Software Evolution and Maintenance. In: *International Conference on Program Comprehension* (2008)
22. Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.: An Information Retrieval Approach to Concept Location in Source Code. In: *Proceedings of 11th IEEE Working Conference on Reverse Engineering*, pp. 214–223 (2004)
23. Grant, S., Cordy, J.R., Skillicorn, D.B.: Automated Concept Location Using Independent Component Analysis. In: *Proceedings of 15th Working Conference on Reverse Engineering*, pp. 138–142 (2008)

24. Hill, E., Pollock, L., Shanker, K.V.: Automatically Capturing Source Code Context of NL-Queries for Software Maintenance and Reuse. In: Proceedings of 31st IEEE/ACM International Conference on Software Engineering (2009)
25. Poshyvanyk, D., Marcus, A.: Combining formal concept analysis with information retrieval for concept location in source code. In: Program Comprehension, pp. 37–48 (2007)
26. Chen, A., Chou, E., Wong, J., Yao, A.Y., Zhang, Q., Zhang, S., Michail, A.: CVSSearch: searching through source code using CVS comments. In: Proceedings of IEEE International Conference on Software Maintenance, pp. 364–373 (2001)
27. Ratanotayanon, S., Choi, H.J., Sim, S.E.: Using Transitive changesets to Support Feature Location. In: Proceedings of 25th IEEE/ACM International Conference on Automated Software Engineering, pp. 341–344 (2010)
28. Classen, A., Heymans, P., Schobbens, P.-Y.: What’s in a feature: A requirements engineering perspective. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 16–30. Springer, Heidelberg (2008)
29. Riebisch, M.: Towards a more precise definition of feature models. In: Modelling Variability for Object-Oriented Product Lines, pp. 64–76 (2003)
30. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study, Carnegie-Mellon University, Pittsburgh, Software Engineering Institute (1990)
31. Kang, K.C.: Feature-oriented development of applications for a domain. In: Proceedings of Software Reuse, pp. 354–355. IEEE (1998)
32. Bosch, J.: Design and use of software architectures: adopting and evolving a product-line approach. Addison-Wesley Professional (2000)
33. Chen, K., Zhang, W., Zhao, H., Mei, H.: An approach to constructing feature models based on requirements clustering. In: Proceedings of Requirements Engineering, pp. 31–40. IEEE (2005)
34. Batory, D.: Feature modularity for product-lines. Tutorial at: OOPSLA, 6 (2006)
35. Batory, D., Benavides, D., Ruiz-Cortes, A.: Automated analysis of feature models: challenges ahead. Communications of the ACM 49(12), 45–47 (2006)
36. Apel, S., Lengauer, C., Batory, D., Moller, B., Kastner, C.: An algebra for feature-oriented software development, Number MIP-0706. University of Passau (2007)
37. Eisenbarth, T., Koschke, R., Simon, D.: Locating Features in Source Code. IEEE Computer 29(3), 210 (2003)
38. Marciuska, S., Gencel, C., Wang, X., Abrahamsson, P.: Feature usage diagram for feature reduction. In: Baumeister, H., Weber, B. (eds.) XP 2013. LNBIP, vol. 149, pp. 223–237. Springer, Heidelberg (2013)
39. HTML Reference, <http://www.w3schools.com/tags/default.asp> (last visited on March 31, 2013)
40. Yin, R.K.: Case study research: Design and methods. SAGE Publications, Incorporated (2002)
41. Alexa traffic rating, <http://www.alexa.com/topsites/countries/GB> (last visited on March 31, 2013)
42. Runeson, P., Host, M.: Guidelines for conducting and reporting case study research in software engineering. In: Empirical Software Engineering, pp. 131–164 (2009)

# Value-Based Migration of Legacy Data Structures

Matthias Book, Simon Grapenthin, and Volker Gruhn

paluno – The Ruhr Institute for Software Technology  
University of Duisburg-Essen, Gerlingstr. 16, 45127 Essen, Germany  
{matthias.book, simon.grapenthin,  
volker.gruhn}@paluno.uni-due.de

**Abstract.** The maintenance and evolution of legacy applications and legacy data structures poses a significant challenge for many organizations that rely on large-scale information systems, e.g. in the financial services domain: Not only is the budget for modernizations that add more technical stability and flexibility than business functionality often slim, but it is also difficult to understand and design the best migration strategies for a large and organically grown system landscape. We report on the experiences from a migration project at a large bank that pursued a value-based approach, in which migration efforts were first focused on a small set of business processes that were identified as most crucial for the enterprise. The migration strategies developed and validated in this pilot phase could later be successfully applied to the larger system landscape.

**Keywords:** Value orientation, legacy systems, migration.

## 1 Introduction

One of the largest IT challenges that many organizations – especially large enterprises in data-intensive domains such as the finance, insurance and healthcare sectors – are currently facing is not the design and implementation of new systems, but the continuous adaption and migration of their legacy application and data landscape to more modern and flexible platforms and systems. Obviously, changing parts of a complex integrated application landscape that supports essential business processes and holds large amounts of customer data is a delicate endeavor in any domain. It is exacerbated by the fact that such landscapes typically evolve more organically than structurally over time, often incorporating technical solutions for business requirements that work very efficiently in their specific context, but are tedious to maintain and hard to generalize or transfer to other contexts. Also, systems tend to stay with a company much longer than the engineers who built them, so the knowledge about system components and data structures, their relationships and their design assumptions, may erode over time if not carefully maintained.

The consideration of these factors obviously makes system migration projects more complex than first meets the eye – however, austerity policies may limit the budget that is available for migration projects in many organizations, especially since such projects usually seem to add little business value (in terms of new products, new

customers etc.), but just provide an incremental efficiency improvement that might not even be measurable in the short run.

In such a situation, it is important to focus the project effort on the critical aspects by understanding which legacy components and data structures are supporting the core business processes, which are of peripheral relevance, and which ones can even safely be ignored because they are no longer required in the new system landscape. Given this understanding, the team's resources can be employed more effectively, and the migration thus carried out more diligently, than if the interdependencies and priorities of the various legacy structures remained unclear.

In this paper, we present our experiences from an ongoing migration project at a large bank, in which a value-based approach was followed to obtain an overview of the dependencies between business processes, system components and data structures, to identify patterns in their dependencies, and to define migration strategies tailored to those patterns. After an overview of the initial situation that led to the migration project (Sect. 2), we present the steps taken in this migration process in Sect. 3, and discuss lessons learned from the project in Sect. 4. We close with an overview of related approaches and concluding observations in Sect. 5 and 6.

## 2 Project Background

In the past, standard IT solutions were not available in the banking domain, so the system landscape of our project partner comprised a large set of heterogeneous self-developed applications for every department, and even for single products. Initially, the IT landscape had been completely based on an IBM mainframe COBOL technology stack using sequential and indexed files, as well as hierarchical and relational databases. Application components usually communicated through complex sequential files. To identify and relate records of all kinds in these data structures, a large set of so-called **keys** was defined and used bank-wide by nearly all applications. A key can be a simple attribute value identifying a data entity (e.g. an account number identifying an account) – we call this a **simple key**. However, a key can also be derived from a combination of values or value ranges of several attributes (e.g. a combination of department ID, service group ID and account number range that identify a certain class of customers) – we call these **complex keys**.

Despite continuous evolution (e.g. the introduction of a client-server environment with message protocols and web services for some applications), the bank never changed their overall technology landscape. One reason is that especially in the last ten years, mostly mandatory or regulatory requirements had to be implemented (e.g. the Euro introduction, Y2K, new tax regulations, etc.), so there was no opportunity for a comprehensive change of the landscape, which therefore remained quite heterogeneous in terms of software technology, database techniques and integration patterns.

Recently, the bank decided to introduce a new back-end system that is based on a standard banking platform by a third-party supplier. From the beginning of the project, it was known that this product would not be able to cover the full spectrum of

the bank's existing services, so those applications whose functionality was not covered by the new back-end would have to remain active in the new environment. The Investments department was one of the areas where the new banking software could not replace existing applications, so interfaces between them and the new system had to be developed.

One of the biggest challenges in the introduction of the new banking system was the deprecation of many legacy keys used by applications throughout the bank to identify all kinds of data records. Their replacement with a much leaner and cleaner set of keys was necessary because over the course of several decades of evolution, most of the key fields originally introduced with the old COBOL components had become semantically overloaded by employing them for different purposes in other applications as well. In other cases, the lack of suitable dedicated keys had led to the use of various content fields as de-facto key fields (for example, some account number ranges were reserved for specific types of customers, because an explicit key field to designate the customer type was not available). This way, many data fields and thus the corresponding keys had become overloaded with semantics over the course of the application landscape's evolution, and the overview of which data fields and keys were used for which purpose had become increasingly hard to maintain – a deterioration as forecast by Lehman already in 1980 [1].

With the introduction of the new banking system, the bank therefore aimed to replace most of the overloaded legacy keys with a more stringent set of just a few clearly defined keys. This made it necessary to analyze the impact of the planned key transformation on the Investments department's landscape of about 50 applications, and to develop suitable migration strategies for those applications and the keys they depended on. Since these changes would affect production applications, some of which were older than 30 years, the guiding principles of the migration project were:

- Keeping the impact on the legacy applications as small as possible, up to the point of designating “untouchable” applications that were not to be changed at all, so as not to introduce new bugs.
- Minimizing the number of legacy keys that had to be included in the new banking system's set of keys, so as not to “taint” it with legacy data structures, and to avoid long-term maintenance responsibilities for those keys in the new system.
- Finding migration strategies that were internal to the Investments department, wherever possible; and in particular, not adding any Investments-specific keys to the globally used banking system.

Since the size of the legacy application landscape made it infeasible to analyze all applications in a straightforward way, it was decided to follow the concepts of value-based software engineering [2] and the Pareto principle, i.e. to develop solutions for a small set of most critical applications first – thereby focusing resources on where they were most urgently needed, and designing solutions around those aspects that allowed the least compromises. It was hoped for (and is actually being confirmed now in the ongoing migration project) that the solutions found in this pilot phase would later also be applicable to the rest of the application landscape, which is less time-critical and more open to adaption.

### 3 Development of Adaption Strategies

To identify the most critical applications, to analyze the impact of the key transformations on them, and to conceive suitable adaption strategies, we undertook the following steps:

1. **Analysis of process value:** Since the criticality of applications depends to a large degree on the importance of the business processes they support, our analysis began with an assessment of the value drivers of the Investments department's 86 business processes (as described in Sect. 3.1).
2. **Analysis of key exposure:** The second factor in determining applications' migration criticality is their exposure to the legacy keys, which was analyzed in the second step (Sect. 3.2).
3. **Identification of most crucial processes:** Based on the processes' value and key exposure, we narrowed the process landscape down to a small set of the most relevant processes, which were to be tackled in the pilot phase of the migration project (Sect. 3.3).
4. **Detailed analysis of key usage:** For all the relevant business processes identified in the previous step, we analyzed the software applications and components that implemented them, and examined on the source code level how they depended on the legacy keys (Sect. 3.4).
5. **Analysis of application components' necessity:** The deprecation of the legacy keys begged the question whether the application functions that they had supported would still be needed in the future, or if they could also be replaced by workarounds, in order to simplify the migration (Sect. 3.5).
6. **Derivation of adaption strategies:** Looking at all the individual instances of key usage we had charted in the previous two steps, we identified a set of recurring key usage patterns, and derived strategies for the migration of the legacy keys to the new key set (Sect. 3.6).
7. **Adaption of application components:** For adapting the application components to the new key structures, architectural solutions had to be found that had as little impact as possible on the existing logic (Sect. 3.7).

Note that the examples presented to illustrate these steps in the following sections have been sanitized to protect the bank's IT landscape and business processes. As such, they represent only a small fraction of the surveyed applications' and data structures' actual complexity.

#### 3.1 Analysis of Process Value

As described in Sect. 2, it would have been neither cost-effective nor manageable to migrate the whole application landscape at once. Instead, it was decided to concentrate on the applications that implement the most important business processes first. However, identifying the most important business processes is not as trivial as it sounds, especially in a landscape of 86 processes for the Investments department alone. This is particularly true given that the degree of "importance" depends on a multitude



of (sometimes competing) criteria. We therefore used our technique of value and effort annotations [3] that was specifically developed to foster joint understanding of critical aspects of a business domain by technical and domain experts.

Following the concept of value-based software engineering [2], we are striving to make the *value* that is inherent in an organization's business processes – and thus the value the information system is expected to provide/support – explicit in the models of the system used in the engineering process. At the same time, we aim to highlight model elements in whose implementation particular *effort* must be invested to address issues, requirements or complexity that would otherwise not immediately meet the eye.

Rather than conceiving new types of models to express these value and effort drivers that cannot be effectively expressed in traditional models, we make them explicit through graphical annotations of existing models (independently of their notation). This has the advantage that stakeholders can use the same set of value and effort annotations in a wide variety of model types (e.g. business process models, data models, architecture models, etc.) to express issues that are often of an overarching nature.

**Value annotations** are directly related to the enterprise's value creation; for example, trading with large amounts of money or calculating financial metrics in reports that are essential for corporate management. **Effort annotations** are often non-functional requirements – e.g. laws and policies set forth by authorities and regulatory bodies, requirements regarding execution time, or secure data storage. When implementing a system, these annotations indicate the need for particular technical effort to be invested in the non-functional requirements (hence the name), while in this migration project, they indicated that the respective processes were likely less amenable to technical compromises.





In the following subsections, we first introduce value and effort annotations covering a broad spectrum of potential issues that otherwise easily remain hidden or implicit in system models. Each annotation is described with its graphical symbol, its usage areas and implications. Then we proceed to report how the annotations were used in our project.

### 3.1.1 Value Annotations

Value annotations denote particularly important system aspects whose support must be ensured so that the benefit expected from the system's features can actually manifest itself. Ignorance of value drivers means that a system will not be useful because its features cannot be used effectively in practice. Since system models typically focus on the features, but not the quality attributes expected of them, we let stakeholders denote these with the following value annotations:


- **Financial responsibility:** Even if they are quite detailed, models of a business process or software component may not convey the amounts of money or other valuable assets that are processed by them. Some components may perform calculations (e.g. interest adjustments) of such volume that any errors – especially when accumulated over many transactions – can lead to significant financial risks for the organization and its clients.







- **Number of users:** Aside from performance considerations, a high number of users imposes particularly high standards onto the quality of a component's user interface: A dialog that only a few people work with (e.g. the Controlling department) does not need to have similarly fine-tuned usability, and must not be suitable for such a diverse audience, as e.g. the point-of-sale interface used by hundreds of cashiers. This annotation indicates that even small improvements made in such a component can yield a significant efficiency gain in a lot of users' work. 
- **Image sensitivity:** Any interaction of an organization with outside parties – customers, suppliers, media etc. – affects the organization's image, i.e. its public perception. Even for interactions that are not deemed crucial in terms of the other value annotations, quick response times and professional presentation (e.g. of the online customer support interface or the monthly statement layout) can affect the company image positively. This annotation therefore indicates features that might seem to be candidates for low prioritization, but whose quality can have a significant “soft” impact on a company's success. 
- **Frequent execution:** Similarly to a high number of users, awareness of a component's high usage frequency (be it by human users or automated services) is important in order to design it for high performance, as well as optimize it for the most frequent usage scenarios in order to increase the efficiency of the dependent business processes. As most value and effort drivers, this annotation may be applied not just to process models, but also to data models, where it highlights that a particular data structure should be optimized for efficient access. This annotation may therefore impact developers' choice of implementation technique in significant ways, based on information that they could not have gained from a data model without the additional usage information. 
- **Company policies:** Many aspects of a system may need to comply with policies that are not immediately visible from class or process diagrams, such as archival policies, best practices, decision-making guidelines etc. Banks, for example, have complex sets of rules governing investment decisions that differ considerably between products. This annotation indicates that developers should not just rely on business requirements and technical context when implementing a process, but be aware that there may be precise guidelines prescribing how the implementation should behave. 

### 3.1.2 Effort Annotations

Effort annotations serve as warning signs for system components that should be expected to require more work than meets the eye, such as performance or security requirements, the need for compliance with legal regulations, etc.:


- **High reliability:** Reliability covers aspects such as the continuous availability of components or data, the interruption-free processing of tasks, etc. It is especially crucial in distributed and mobile 


systems, where mobile workers may depend on certain functionality or data being available even if their network connection to the server breaks. This annotation therefore indicates that developers need to take extra precautions to ensure the availability of the annotated resources (application logic or data structures) under all circumstances.

- **Time constraints:** Time constraints in information systems typically refer to prescribed processing or response times (e.g. the maximum time it should take to make a decision on accepting or declining clients because of their risk profile), the availability of current data records (e.g. stock quotes), or the time of execution of certain operations (e.g. financial transactions). Thus, this annotation indicates that developers need to take real-time aspects much more strictly into account than they usually would in business information systems. 
- **Resource demands:** Beside components that obviously require significant computing resources (i.e., processing power and storage space), such as a bank's customer database, information systems may be exposed to resource demands that are harder to identify, especially when that demand is not constantly present, but occurs only in more or less predictable peaks – example scenarios might include annual mailings of interest earnings, but also unexpected peaks in share subscriptions or sales. In these cases, processes or components that may look peripheral or trivial at first glance may actually require sophisticated scaling techniques. This annotation therefore not just indicates which components require large processing resources, but especially where precautions for scaling – potentially at short notice – must be taken, e.g. through the use of cloud computing infrastructures. 
- **Security standards:** The need for digital signage of certain records to ensure non-repudiation, the need for anonymization of certain records before processing, or the enforcement of confidentiality of certain parts of records (in a bank, e.g., not all staff members that are allowed to access a client's checking account may also be allowed to see that client's custody) are special security requirements that the implementation of business process and data structures must respect. This annotation therefore highlights processes and data structures for which particular security mechanisms need to be implemented that are not part of the organization's overall security standards, and not obvious from the model itself. 
- **Legal regulations:** For some business processes, there may be legal prescriptions that govern how the process must be executed. As opposed to company policies that usually describe preferred best practices, legal policies often do not provide an immediate value for the organization, but are motivated by external interests such as market or consumer protection. This annotation therefore highlights processes or data structures that need to be implemented in particular, non-obvious ways in order to satisfy applicable legislation. It can also indicate areas that are known to become regulated in the future, even if the law's details are still under 

discussion, and thus serve as a warning that a certain degree of flexibility should be built into that system component.

- Precision:** Some business processes are subject to particular requirements regarding fault intolerance. Even small deviations in algorithms, for example rounding errors of interest rates, can cause great damage, because many executions could be affected. Therefore this annotation indicates that extreme care must be taken in ensuring the component's correct implementation and testing in close cooperation with business experts.



- Sensitive data:** Data privacy is an important cross-cutting aspect in most business information systems. Some components, however, may process or hold particularly sensitive data, such as customer data or custody accounts. This annotation indicates that the implementing components have special requirements for data security and a poor implementation could lead to significant damage for the organization.



### 3.1.3 Wildcard Annotation

In addition to the above annotations that highlight particular value or complexity inherent in designated processes, data structures or components, we use one further annotation to point out complexity of a more general nature:

- Implementation complexity:** To highlight highly domain- or technology-specific value or effort drivers that do not fall into one of the above-mentioned categories, this annotation can be attached to any model element, where it serves as a warning that implementation of this process, component or data structure will require more effort or needs to satisfy higher quality standards than immediately meets the eye. Examples might be complex integration with third-party services, historically evolved conventions for the proper formatting of legacy data structures, additional effort for the adaption of COBOL components, etc.



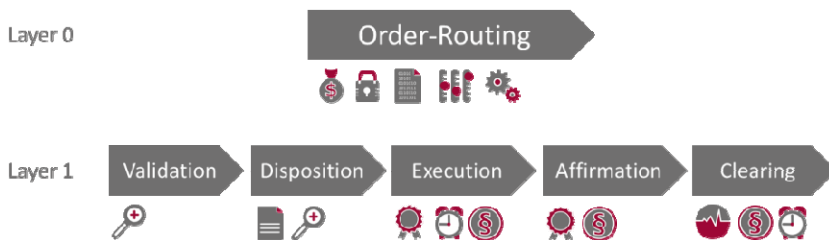


Fig. 1. Annotated *Order Routing* business process














### 3.1.4 Assessing the Investments Department's Business Processes

Using these annotations, all factors that could make a business process or activity particularly critical were highlighted in the process models and discussed by business

and technical stakeholders of the bank. For this purpose, the high-level “layer 0” processes were broken down into more fine-grained “layer 1” sub-processes, as shown in Fig. 1 using the example of the *Order Routing* process. Annotations could be placed on either layer – with an annotation on layer 0 meaning that the respective value or effort driver applied to the complete process, while an annotation on layer 1 indicated a value or effort driver to have specific impact on a particular sub-process.

The results of the process annotation were documented in a process value table (excerpted in Table 1), which ranks the layer 0 processes by the number of different types of value and effort annotations (#A) associated with them and their sub-processes. This approach of just counting the annotation types may seem quite simplistic at first sight, as one might consider calculating a more precise ranking by assigning different weights to the annotations. However, we found in our discussions that the annotations would have to be weighed differently depending on the individual process in question. Instead, we found counting the annotations to be a pragmatic heuristic for judging the processes’ relative importance, which would still be refined in the following steps. Also, while not reflected in the raw numbers in the table, the discussions sparked between stakeholders in the course of the annotation process provided valuable insights into technical or business particularities of the individual processes that would have to be considered during their adaption.

**Table 1.** Excerpt of process value table

Process	#A													
Settlement	12	x	x	x	x	x	x	x	x	x		x	x	x
Order Routing	10	x	x		x		x	x	x	x		x	x	x
Securities Account Pricing	10	x	x	x		x		x		x	x	x	x	x
Transaction Pricing	10	x	x	x	x	x		x	x	x		x		x
Asset Services	9	x	x	x		x		x		x		x	x	x
Investments Proposal	8	x		x			x	x			x	x	x	x
Order Execution	8	x	x		x			x	x	x		x	x	
Securities Account Statement	5			x			x				x	x	x	
...	...													

### 3.2 Analysis of Key Exposure

In addition to the relative importance of the processes assessed in the first step, another significant criterion for their criticality was their exposure to the legacy keys

that were about to be phased out. We therefore also analyzed all of the business processes with regard to what data they processed, and which key fields they depended on for that purpose. The results were documented in a key exposure table (excerpted in Table 2), which ranks all processes by the number of keys (#K) they depend on. While we did not yet look in detail into the applications implementing the processes in this step, it seemed natural to assume (and was later confirmed) that the complexity of the applications would correlate with the processes’ key exposure – the more keys a process depends on, the more difficult it would be to adapt the respective application, and thus the more important to find solutions for it in the pilot phase already.

One might wonder why we did not examine the applications directly, but instead took the “detour” via the business processes to assess value and key exposure. The reason is that examining the implementation of 50 applications (some of them several decades old) in detail would require prohibitive effort, while examining the general business processes, whose requirements and relationships the team members were familiar with, was a significant but still feasible undertaking. Of course, we eventually did look at application implementations (see Sect. 3.4), but only after we identified the most crucial ones that merited closer scrutiny in the pilot phase.

**Table 2.** Excerpt of key exposure table

Process	#K	$K_1$	$K_2$	$K_3$	$K_4$	$K_5$	$K_6$	$K_7$	$K_8$	$K_9$	$K_{10}$	$K_{11}$	$K_{12}$	$K_{13}$	$K_{14}$
Order Routing	9			x	x	x	x	x		x	x	x		x	
Transaction Pricing	9			x	x	x	x	x		x	x	x		x	
Securities Account Pricing	8			x		x	x	x		x	x	x		x	
Securities Account Statement	8			x	x		x	x		x	x	x		x	
Settlement	7			x	x	x	x			x		x		x	
Asset Services	6			x		x	x			x		x		x	
Investments Proposal	6			x	x		x				x	x		x	
Order Execution	2										x	x			
...	...														

### 3.3 Identification of Most Crucial Processes

Based on the ranking of business processes by value and effort annotations and key exposure, we could now identify those most crucial processes which should be addressed in the migration project’s pilot phase. For this purpose, we plotted the business processes in a coordinate system of number of annotations and key exposure, to obtain the business process criticality matrix (Fig. 2). This matrix enabled us to focus our efforts on those processes that were deemed to be most important for the business, and at the same time pose the most complex technical challenges, i.e. those processes that can be found in the top right area of the matrix.

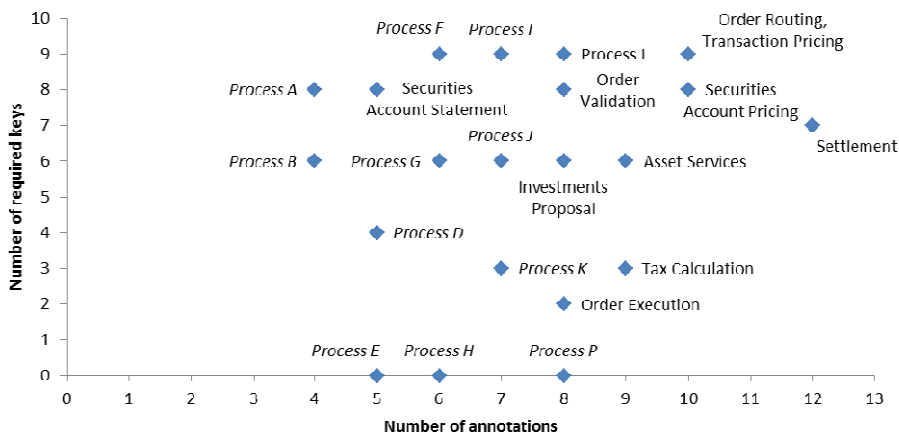


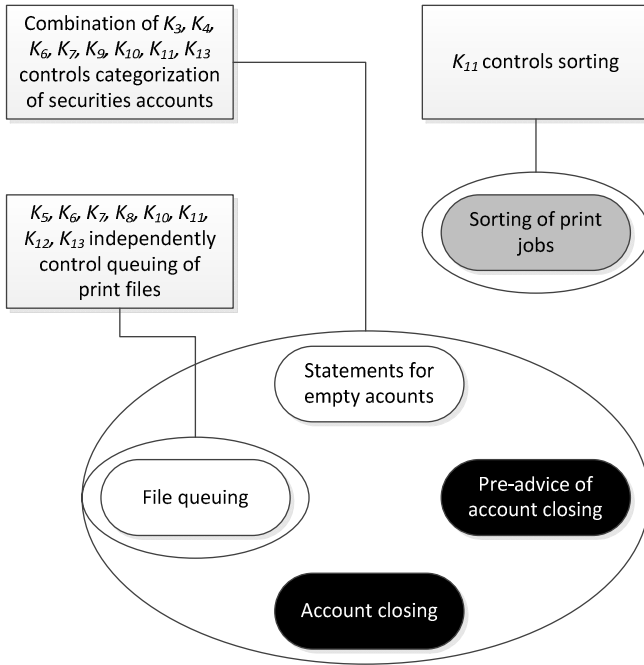
Fig. 2. Excerpt of process criticality matrix

However, instead of mechanically focusing on e.g. the five top-right-most processes, we looked at all highly annotated and highly exposed processes, in order to carefully pick those that we expected to be the most critical and urgent, based on our refined understanding of their dependencies and implications gained in the preceding discussions. We ultimately selected five processes for detailed analysis and adaption – among them e.g. the *Settlement* and *Order Routing* processes that were highly ranked in both dimensions, but also the *Securities Account Statement* process due to its very high key exposure despite fewer annotations.

### 3.4 Detailed Analysis of Key Usage

Having selected the most critical processes, we let five dedicated teams of business analysts and engineers investigate the corresponding software components in detail. The five processes were analyzed end-to-end – from the processes' beginning to their conclusion; from their front-end to their back-end components – by decomposing them into individual activities, mapping them to their implementing application components, and examining those on the source code level. After the preceding business-focused analysis, this technical step was necessary in order to examine how exactly the legacy keys were processed in the implementation. In the course of this analysis, the teams documented their findings in key usage diagrams that illustrate which keys an application component depends on, and how these keys are constructed or related.

As an example, Fig. 3 shows a key usage diagram for some of the application components of the *Securities Account Statement* process: Application components such as *File queuing* or *Account closing* are displayed as boxes with rounded corners (the different shadings will be explained in Sect. 3.5). Their relationship to particular



**Fig. 3.** Example key usage diagram

keys (displayed in boxes with rectangular corners) is visualized by encircling them and connecting them to the respective key box, which may denote a single key or a group of keys. This way, we could illustrate relationships such as whether a certain application component depends on a single key or a set of multiple keys; for application components depending on a set of keys, whether those keys are evaluated in combination, or pertain to independent aspects of the component; and whether a key (or key set) affects only one application component, or a whole set of them. For example, in Fig. 3, a combination of eight keys was required by virtually all components dealing with security accounts statements. In addition, eight other keys independently controlled various aspects of queuing the print files. Sorting the printouts just depended on one single key.

With detailed diagrams like this for all application components of the most crucial processes, the business analysts and engineers were able to discuss relationships between keys and applications in detail, to understand and communicate consequences if particular legacy keys would be phased out in the future, and thus to justify budget requests for adapting individual applications.

### 3.5 Analysis of Application Components' Necessity

While analyzing application components' dependence on the various keys, we also analyzed the consequences that phasing out a deprecated key would have for the



application function, and how the loss of functionality could be compensated for: Often, an application function would be essential, so the information carried by the legacy key also had to be included in the new key set in some way, and the application component had to be adapted to use the new key. In some cases, however, it also seemed possible to construct workarounds for the functionality, which made the continued dependence on the key obsolete, or even retire the functionality in question altogether. In the key usage diagrams, we illustrated these impacts by shading the application component boxes as specified in Table 3.

**Table 3.** Application component necessity shading

Shading	Application component necessity
White	Key-dependent functionality is required as-is
Gray	Key information and/or functionality can be replaced by workaround
Black	Functionality not needed anymore

In Fig. 3, for example, we see that the *Statements for empty accounts* – a business-wise relatively simple function – depends on a combination of eight keys, so its adaption would be quite complex. However, since this functionality is required by national legislation, the component is indispensable, as the white shading indicates. Adding this information to the key usage diagram provides important background information for the choice of adaption strategy, as the following steps show.

### 3.6 Derivation of Adaption Strategies

To recap, our selection of just five business processes in the pilot phase of the key migration project had a twofold purpose: Firstly, to ensure that the initially available time and budget would be invested in the most critical processes, and secondly, to develop solution strategies that could be applied to all other processes after the pilot phase as well.

Based on the overall picture we gained from the analysis of the processes' business value and key exposure, and from the application components' key usage and necessity, we therefore derived a number of typical key usage patterns, for which we developed general adaption strategies (Table 4).

The first pattern applies to all components shaded black in the key usage diagrams: If it turns out that the functionality enabled by the legacy key is actually not needed anymore, no effort has to be invested into migrating the key to the new system (unless it is still independently needed by another component), and into adapting application components to support it further. This was the option chosen e.g. for the *Pre-advice before account closing* functionality. (If such a component was still relevant for other departments outside Investments, maintaining it would become their responsibility.)

**Table 4.** Key usage patterns and software adaption strategies

Legacy key usage pattern	Adaption strategy
1. Key and component functionality dispensable	Retire component
2. Key dependence resolvable by workaround	Adapt process or component for workaround
3. Functionality depends on single key value	Transfer key into new key set
4. Functionality depends on single characteristic derived from multiple keys	Create single key for combined characteristic in new key set
5. Functionality depends on particular segment or value range of a single key	Create dedicated key for segment or value range in new key set

The second pattern applies to all components shaded gray in the key usage diagrams, i.e. those where a technical or operational workaround can be found to eliminate the need for the key. In this case, the component needs to be adapted to include the workaround – e.g. deriving the needed criterion from other context information, or restructuring the process so the key is not needed anymore. This was e.g. the strategy chosen for the *Sorting of print jobs* component, where a simple change in the manual process achieved the same effect as the automated solution, without having to keep the legacy key.

Patterns 3 to 5 apply to all components shaded white in the key usage diagrams, i.e. indispensable application functions that remain dependent on the information in the legacy keys. To maintain their functionality, the legacy key must be migrated to the new system's key set, and the component must be adapted to access it there. Depending on the structure of the key, we need to distinguish different patterns though:

Pattern 3 applies to application components that require a simple, atomic key, such as an account number or employee ID. These kinds of keys can be easily transferred to the new banking system's set of keys, and require only minimal changes to the component's implementation, since the key information is just retrieved from a different system, but processed in the same way.

Pattern 4 in contrast applies to components that derive certain information from a combination of various keys. For example, in the generation of securities account statements, a specific rule must be applied to a certain group of customers. This group of customers was identified by a certain range of account numbers in combination with the department ID and the service group ID associated with these customers. The combination of all these digits enabled a simple binary decision on whether this customer was part of the particular group or not. In the course of the migration, our strategy was to compute the derived customer group characteristic beforehand, and store this binary value as a simple key in the new system.

For many application components, a key change like this also promises a simplification of the historically evolved control logic that could be replaced with

much more compact logic relying just on the one new key. In case of the *Securities Account Statement* component, however, we were dealing with an “untouchable”, which required special precautions, as described in Sect. 3.7.

Pattern 5 finally applies to semantically “overloaded” key fields that may serve multiple purposes in different contexts – for example, an account number that has the overlaid semantic of customer type (represented by certain account number intervals), or a string identifier in which e.g. characters 1-5 and 6-10 denote independent characteristics of a particular record. In these cases, we strive to decouple the independent aspects that were conjoined in the legacy key, and express them through distinct keys in the new system. Again, this obviously also requires significant changes in the application components working with these keys, as discussed in Sect. 3.7.

The adaption strategies that we postulated for each of the key usage patterns were validated by applying them to the application components implementing the top processes we had selected for the pilot phase, where they were iteratively improved in detail and generally found to match well. While we could not expect that these solutions would be immediately transferrable to the remaining components, we now had a reduced solution space at our disposal that might only require minor tweaking to be applicable to the remaining large set of processes.

### 3.7 Adaption of Application Components

Most of the key transformations that were recommended by the above adaption strategies also had to be accommodated in the dependent application components’ implementation – in the case of simple keys (pattern 3), this involves just a minimal change regarding where to find the key in the new system, but in the case of the more complex keys (patterns 4 and 5), the adaption would involve replacing significant parts of the components’ control logic that originally merged or dissected the keys. Such significant changes in applications’ implementations were deemed undesirable in two situations:

- Firstly, if the application had been designated as a so-called “untouchable”, i.e. a possibly decades-old application whose outdated technology made it uneconomical to invest a lot of effort into its modernization (for one thing, because it will likely be replaced a few years into the future anyway, and for another, because changing a COBOL implementation is much more expensive and error-prone than adapting a modern object-oriented component).
- Secondly, if the legacy key in question was used by so many different applications that adapting them all would all incur prohibitively high effort; so they were effectively considered “untouchable” as well.

However, even in these situations, where the application components should remain dependent on the legacy key structures, we wanted to avoid cluttering the new banking system with the old keys.

The solution to this apparent dilemma was the introduction of a so-called **transformation layer** that decoupled the “untouchable” applications and their legacy-key-expecting interfaces from the new system with its more streamlined key

structures. As shown in Fig. 4, this transformation layer consists of individual adaptors for each “untouchable” application component: For new keys that were merged from a number of legacy keys (such as  $K'_4$ , which was merged from  $K_2$  and  $K_3$ ), the transformation layer dynamically rebuilds the set of legacy keys ( $K_2$  and  $K_3$ ) that the “untouchable” application would interpret just like the merged key. On the other hand, for new keys such as  $K'_{10}$  that were derived from a segment of a legacy key such as  $K_9$ , the transformation layer will recreate a key in the legacy format, which is however only populated with data in the segment that the legacy applications expect.

While this approach requires some effort for formulating appropriate key reconstruction algorithms in the transformation layer, and incurs a slight overhead in execution time, it enables much larger savings since it eliminates the need to change the key control logic in a large number of applications, or to ensure the correctness of changes in “ancient” components.

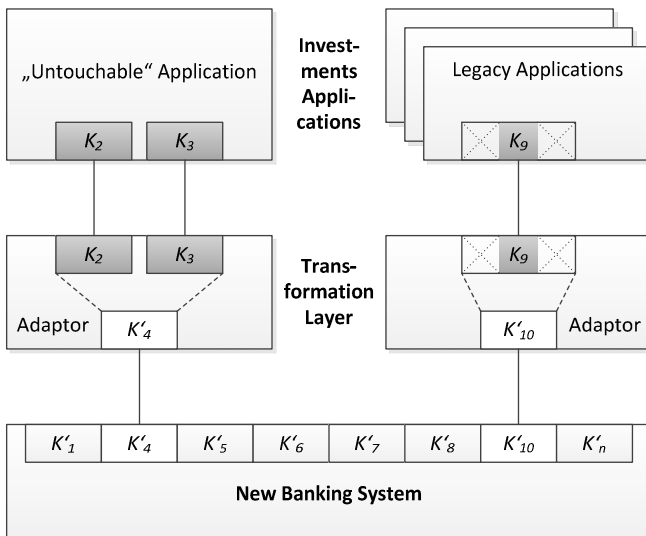


Fig. 4. Applications with transformation layer

## 4 Discussion

Beginning the migration project with the pilot phase described here enabled us to develop solutions for a small set of processes that delivered the most business value and addressed the most critical technical challenges first. The approach not only ensured that the Investments department’s resources and attention were focused on the most crucial processes, instead of being scattered over the whole process landscape, but also enabled them to develop and validate solution strategies that could then be applied to the rest of the process landscape (comprising about 80 further processes) with less effort than if they had to be developed from scratch for each process without prioritization.

## 4.1 Resulting Migration Process

The solution strategies developed in the course of the pilot phase were documented in the form of a migration process (Fig. 5) that is now being applied to every application component in the Investments department's system landscape. According to this decision process, business analysts and engineers first examine whether the component can be retired as well when the keys it depends on are phased out; if an operational workaround for the legacy keys can be found; or if the application is indispensable and thus must be adapted.

In the latter case, we distinguish between simple, atomic keys (which can simply be added to the new system's key set), and complex keys that are derived from multiple keys or extracted from a particular segment or value range of a key. For these, we construct according atomic keys in the keyset (the process model in Fig. 5 places these steps in concurrent branches to indicate that particularly complex legacy key structures might require both processing steps).

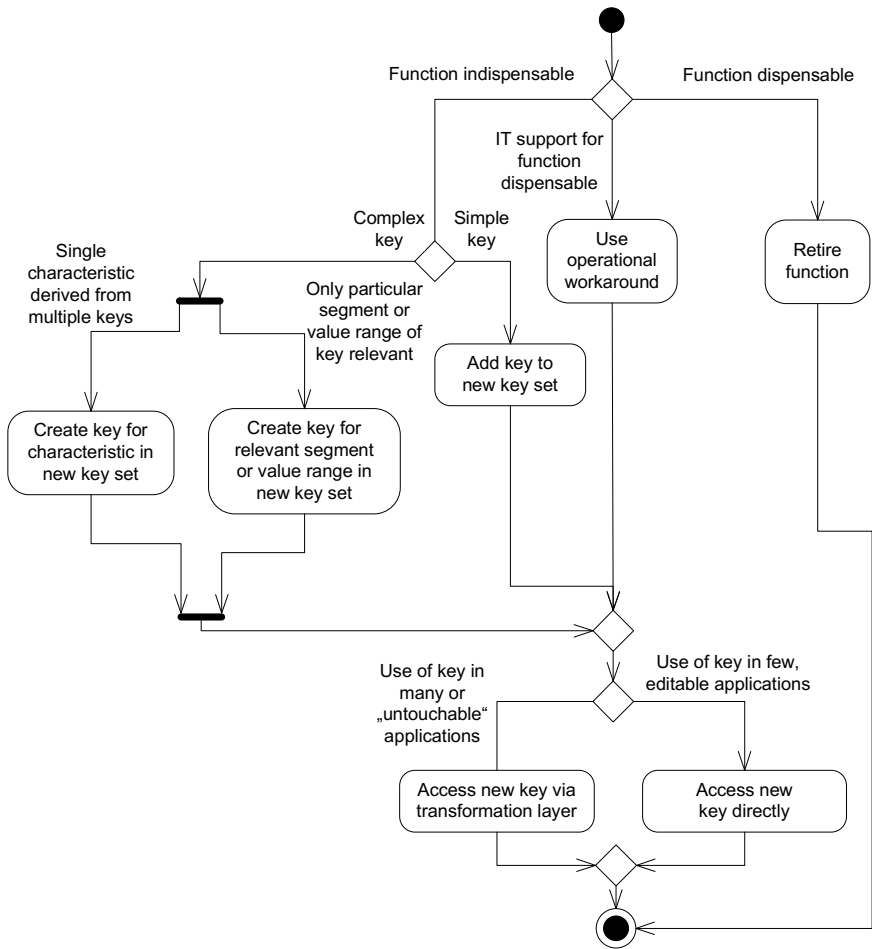
After adding the new keys to the new banking system, we can adapt the application component directly to employ the new keys – or, if the application is “untouchable” or the keys have a high reach, i.e. are used by a large number of applications, we implement an adaptor in the transformation layer to perform the translation between the new keys and the application's legacy interface.

During the ongoing migration of application components, the bank found the set of solution patterns identified in the pilot phase to be comprehensive, since no other paths than those described in the new migration process had to be pursued for any component until now.

## 4.2 Threats to Validity

Our approach for value-based piloting of a large legacy system's migration was applied here in the course of an actual industry project that examined production-critical processes and systems, which obviously did not give us much latitude in terms of controlled experiment design. Specifically, we had no control over the staffing of the project and thus the background and experience of the team members. Instead, we carried out action research, i.e. we served as advisors to the bank's business and IT analysts, rather than purely observing their execution of the method.

While our involvement (in terms of methodical guidance) can likely not be dismissed as one contributing factor to the project's success, the key success factor was certainly the team members' understanding, adoption and integration of the techniques with their analytical experience and domain knowledge. In particular, the continuing migration of the department's further processes according to the established strategies, and the willingness to adopt this method for future large-scale projects as well (without our ongoing or future participation) underscores the applicability and benefit of the approach in practice. Still, given the nature of the study, our findings should be treated as anecdotal rather than generalizable evidence for the effectiveness of the method.



**Fig. 5.** The bank’s newly established migration process for future key and component adaptations, derived from our experiences in the pilot phase

## 5 Related Work

There is a large body of literature on the topic of data and system migration, or software maintenance and evolution in general – Brodie and Stonebraker [4], Bennett [5] and Bisbal et al. [6], among others, described the field early already. The project we presented here is another confirmation of Brodie and Stonebraker’s statement that “the fundamental value of a legacy information system is buried in its data, not in its functions or code” [4]. Many discussions of system and data migration tend to be highly technically focused, reflecting the challenges of a branch of software engineering where the devil is usually in the details. For example, Ceccato et al. [13]

report on experiences from the migration of a large banking system to Java – while they were dealing mostly with issues on the code level, our work focuses on the migration of the actual business data and its interlinking references.

The economic aspects of system migration were addressed early on by Sneed [7] with a planning process that involved justifying the need for migration to sponsors, prioritizing applications, estimating costs and juxtaposing them with benefits, before actually contracting the migration work. Many of these aspects are reflected in our approach – for example, the dependencies illustrated in the key usage diagrams were used to support budget requests, and the component necessity analysis ensured that migration efforts were only expended on components where the same benefit could not be achieved through workarounds.

The use of a value-based approach in a migration project was described by Visaggio [8], who extended Sneed's approach by providing guidelines for scoring the economic value and the technical quality of legacy components. Instead of his rather detailed quantitative approach (as exemplified e.g. in the case study by Tilus et al. [12]), we found it more effective to follow a pragmatic, qualitative approach, in which the stakeholders used value and effort annotations to highlight not only how critical they believe processes and applications to be, but also of what nature the critical aspects are – such as supporting large numbers of transactions, affecting the company image, being exposed to particular legal regulations, etc. Given that our focus in this project was less on justifying the benefits of the migration, rather than prioritizing the migration steps and raising awareness of the risks and complexity involved in them, we felt this heuristic approach to be more appropriate than more quantitative approaches.

Involving the stakeholders in this analysis, and actually starting on the business process level rather than the application level, is an approach that worked well also in other migration projects, such as the (also data-intensive) migration of a legacy academic information system that Liem et al. [9] report on.

Bocciarelli and D'Ambrogio also employed the concept of annotations to extend the Business Process Model and Notation (BPMN) with information on non-functional requirements [10]. While some of our value and effort annotations (such as reliability and security) are related to non-functional requirements, we do not need the precise quantification that their approach provides, but rather a more diverse spectrum of value and effort drivers beyond performance and reliability.

The technical solutions we employed – mapping keys, and using adaptors in the transformation layer – are patterns whose application in migration projects was already described by e.g. Thiran and Hainaut [11]. Our focus here was however less on the technical solutions than on the process of breaking down the overall migration challenge into problem classes (the key usage patterns) and devising solution strategies for the most critical ones of them, that could then also be applied to all the others.

## 6 Conclusion

In this paper, we reported how a large bank dealt with a complex migration challenge under time and budget limitations by using a value-based approach to identify the legacy system components and data structures that drove the most critical business

processes, identifying data usage patterns in the existing components, and devising strategies to migrate them.

This approach had two benefits: Firstly, it ensured that the team developed solutions for the most pressing challenges first. Secondly, after implementing these solutions, the team now has a proven set of strategies and a formal migration process at hand that it can apply to the ongoing migration of the secondary systems that were initially disregarded in the pilot phase. Had the team tried to work out solutions for all systems at once, there would have been a significant risk that the team could have lost its focus on the most critical challenges amid all the peripheral aspects, and required more time to come up with less straightforward solutions.

This risk has been successfully averted by using the approach of fostering value-driven understanding and prioritization of tasks – an approach that has met widespread interest in other departments of the bank as well since the success of this project.

## References

1. Lehman, M.M.: Programs, life-cycles, and the laws of program evolution. *Proc. IEEE* 68(9), 1060–1076 (1980)
2. Boehm, B., Huang, L.: Value-based software engineering. *ACM Software Engineering Notes* 28, 4–10 (2003)
3. Book, M., Grapenthin, S., Gruhn, V.: Seeing the forest and the trees: Focusing team interaction on value and effort drivers. In: *Proc. 20th Intl. Symp. Foundations of Software Engineering (FSE-20) New Ideas Track*, art. no. 30. ACM (2012)
4. Brodie, M., Stonebraker, M.: *Migrating legacy systems*. Morgan Kaufmann (1995)
5. Bennett, K.: Legacy systems: Coping with success. *IEEE Software* 12(1), 19–23 (1995)
6. Bisbal, J., Lawless, D., Wu, B., Grimson, J.: Legacy information systems: Issues and directions. *IEEE Software* 6(5), 103–111 (1999)
7. Sneed, H.M.: Planning the reengineering of legacy systems. *IEEE Software* 12(1), 24–34 (1995)
8. Visaggio, G.: Value-based decision model for renewal processes in software maintenance. *Annals of Software Engineering* 9(1-2), 215–233 (2000)
9. Liem, I., Schatten, A., Wahyudin, D.: Data integration: An experience of information system migration. In: *Proceedings of the 8th Intl. Conf. Information Integration, Web Applications and Services (IIWAS 2006)*, Österreichische Computer Gesellschaft, vol. 214 (2006)
10. Bocciarelli, P., D’Ambrogio, A.: A BPMN extension for modeling non-functional properties of business processes. In: *Proc. 2011 Symposium on the Theory of Modeling & Simulation: DEVS Integrative M&S Symposium (TMS-DEVS 2011)*, pp. 160–168. Society for Computer Simulation Intl. (2011)
11. Thiran, P., Hainaut, J.L.: Wrapper development for legacy data reuse. In: *Proc. 8th Working Conference on Reverse Engineering (WCRE 2001)*, pp. 198–207. IEEE Computer Society (2001)
12. Tilus, T., Koskinen, J., Ahonen, J.J., Lintinen, H., Sivula, H., Kankaanpää, I.: Software evolution strategy evaluation: Industrial case study applying value-based decision model. In: *Proc. 9th Intl. Conf. Business Information Systems (BIS 2006)*. LNI, vol. P-85, pp. 543–557. GI e.V (2006)
13. Ceccato, M., Dean, T.R., Tonella, P., Marchignoli, D.: Migrating legacy data structures based on variable overlay to Java. *Journal of Software Maintenance and Evolution: Research and Practice* 22(3), 211–237 (2010)



# An Integrated Analysis and Testing Methodology to Support Model-Based Quality Assurance

Frank Elberzhager, Alla Rosbach, and Thomas Bauer

Fraunhofer IESE,  
67663 Kaiserslautern, Germany

{frank.elberzhager, alla.rosbach, thomas.bauer}@fraunhofer.iese.de

**Abstract.** Model-based development of software is an increasing trend. As quality assurance is one major activity during software development, we aim at improving this task by combining analysis and testing more strongly during model-based development. We first give an overview of the state of the art of combined quality assurance and optimization of quality assurance based on the use of metrics, which reveals a lack of knowledge and methodology regarding this process. We then introduce our initial concepts of a combined quality assurance methodology, and present a proof of concept via an example that applies the approach. The approach is widely applicable and presents a basis for gathering knowledge between different static and dynamic quality assurance techniques in order to improve quality assurance. However, identifying concrete knowledge will remain a major challenge in the future. Finally, we present further lessons learned and give an outlook on future work.

**Keywords:** analysis, testing, combination, integration, defect types, Matlab Simulink.

## 1 Introduction

Defects are a disturbing, but inevitable fact of today's software. Especially defects that are not found before software is delivered can result in serious consequences such as high monetary loss or even risk for human life. This is especially true in the embedded domain. Two very prominent examples are the software defect of the Therac-25 radiation therapy machine which resulted in deaths and serious injuries by overdoses between 1985 and 1987 [64] and the software bug of the Ariane 5 rocket in 1996 which led to the explosion of the rocket and a loss of about 500 million \$ [63]. A lot of different strategies and solutions have emerged in recent decades to improve software before delivery. One such solution is to conduct analytic quality assurance, for example by applying different analysis or testing techniques. A tremendous number of approaches have been published in the last decades [53, 54]. However, certain problems remain, such as increased effort for performing testing, which sometimes consumes more than 50 % of the overall development effort, or unreliable software products with a large number of defects.

One solution is to apply different kinds of quality assurance techniques to find more defects, and to find them cheaper. For example, if a defect is found during a static model analysis instead of during later testing, the issue can be corrected earlier and effort might be saved. However, different quality assurance techniques are usually not applied in a way that they influence each other, but are rather applied in isolation without further synergy effects being exploited. Consequently, we propose to further improve analytical quality assurance by combining different quality assurance techniques more thoroughly, i.e., by integrating different types of static and dynamic quality assurance to improve overall quality assurance. This is not a complete new trend, as, for example, research in combining concrete and symbolic testing [45] or combining inspections and testing [46] has been published before. However, a major challenge is to identify relationships between different static and dynamic quality assurance techniques in order to be able to apply them in an efficient manner. For example, if we know that a Pareto distribution of defects is valid in our context, then those parts of a model in which a static analysis finds most of the problems can be tested further.

Our general goal is to provide an approach that is able to integrate different static and dynamic quality assurance techniques on different development levels which exploit knowledge about the relationships between these techniques in order to address various quality goals. Tools used in these quality assurance activities should be considered. Our focus currently is on the embedded domain due to the partners that provide us with data. Thus, the approach may be most applicable in the embedded domain first, due to the knowledge gathered; however, most of the general concepts are also valid in other domains. Fig. 1 shows our overall research goal including examples for each aspect. In this paper, we present the basic methodology and give some proof of concept, i.e., we provide a generic framework to be able to gather knowledge about integrated quality assurance in the future. In the remainder of this paper, we use analysis synonymously to static quality assurance, and testing synonymously to dynamic quality assurance.

RESEARCH GOAL	EXAMPLES
In order to be able to identify relationships between different quality assurance techniques, and to optimize quality assurance, develop an approach that..	
integrates static quality assurance techniques (analysis)	Stateflow model analysis, requirements review
and dynamic quality assurance techniques (testing)	Unit testing, integration testing
on different levels	Design, code, integration, system
considering different objectives and	Reducing #defects, improving coverage
considering typical tools (respectively models from the tools).	Matlab Simulink, Uppaal

**Fig. 1.** Research goal for an integrated approach in the embedded domain

The remainder of this article is structured as follows: Section 2 gives an overview of related work regarding combined quality assurance approaches, quality assurance in the embedded domain, and how to optimize quality assurance using different metrics. Section 3 presents our integrated quality assurance methodology, and Section 4 shows a concrete instantiation together with initial experiences. Finally, Section 5 closes with a summary and an outlook on future work.

## 2 Related Work

### 2.1 Combined Approaches

A combination of different quality assurance techniques is no new idea, but a lot of different approaches have been pursued in the past. Even early development models, such as the classic V-model, provided motivation to apply different quality assurance techniques during the development process. However, strong integration of quality assurance techniques has only been observed in recent years.

Elberzhager et al. [47] performed a systematic literature review aimed at identifying approaches that combine different static and dynamic quality assurance techniques. They found that the improvement of quality assurance processes through a combination of static and dynamic quality assurance techniques received increased interest during the last decade. Goals such as improved effectiveness or efficiency, or increased coverage of the system under test should be achieved through synergy effects resulting from the systematic combination of different static and dynamic quality assurance techniques. A variety of different techniques have been combined, often at the code level. A recent systematic mapping study by Elberzhager et al. [47] classified such combinations into either compilation or integration approaches.

Compilation means that a static and a dynamic quality assurance technique are combined, but they do not use each other's results, i.e., they are applied in isolation. For example, Zimmerman and Kiniry propose a combination of a static checker, a runtime assertion checker, and a unit-test generator [48]; and Chen et al. describe an approach that checks synchronizations and accesses to shared variables via static analysis and performs dynamic analysis of run-time values [49]. Such approaches are typically supported by tools. Furthermore, a large number of publications describe the combined application of different inspection and testing techniques in isolation (e.g., [50, 51]).

Integration approaches comprise approaches where one technique uses outputs from the other technique to overcome disadvantages from using them in isolation (i.e., in a compiled manner). For instance, Chen et al. integrate model checking and model-based testing [52]. Furthermore, Elberzhager et al. propose the In<sup>2</sup>Test approach, which explicitly integrates inspection and testing techniques [46].

Though some of the approaches have been evaluated, little experience with such combined approaches is available. Knowledge about relationships between different static and dynamic techniques that could be used for combinations is either non-existent or only scratches the surface. Furthermore, no general approach for combining static and dynamic quality assurance techniques exists; rather, there are only a lot of very specific solutions that cannot be applied in various environments.

## 2.2 Quality Assurance in the Embedded Domain

Quality assurance of embedded software-intensive systems is performed on different levels of abstraction regarding the software, hardware, and integrated system aspects. The maturity of the quality assurance process highly depends on the criticality of the devices and systems being developed. International standards for the development of software-intensive safety-relevant systems, such as IEC 61508 [59] and ISO 26262 [60], call for systematic and complete quality assurance of construction artifacts on different abstraction levels using various several means depending on the criticality of functions and components. Quality assurance of non-safety-related embedded systems is often less strict and less mature.

In the last decade, standardized system and software architectures like AUTOSAR [61] have been defined to facilitate manufacturer-supplier relations and increase flexibility when third-party components are used. The introduction of model-based development has led to additional quality assurance activities on the design level before the actual system is implemented and assembled. Depending on the level of abstraction, different kinds of structural, behavioral, or data models become available for early quality assurance. Examples include Simulink / Stateflow and ASCET block diagrams and data flow models as well as UML state machine diagrams, state charts, and activity diagrams.

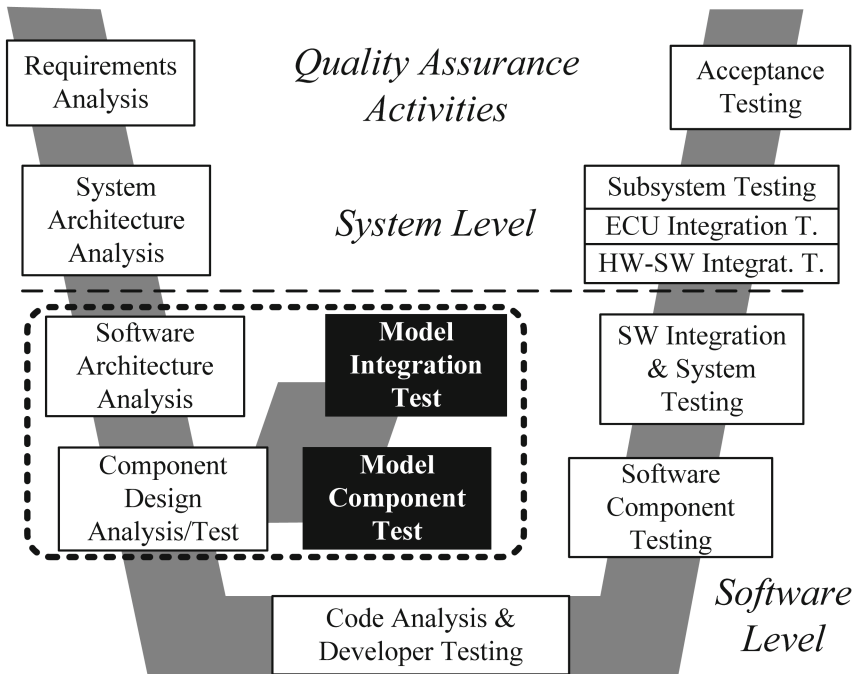


Fig. 2. Sample quality assurance process

The figure shows a simplified quality assurance process for embedded systems that runs in parallel to the development and construction process. Construction is characterized by the systematic refinement of higher-level artifacts. The outcomes of each stage of the development process are usually quality assured using appropriate means. Informal artifacts like textual requirements and high-level architectural descriptions are analyzed manually, for example by means of systematic inspection. Formal artifacts such as concrete design and program code are formally verified in order to check critical properties like deadlocks, out-of-bound signal values, uninitialized variables, unreachable elements, and worst-case execution times.

Executable artifacts like program code and simulatable design models are additionally tested to check their compliance with corresponding specifications. Testing is characterized by the stimulation of the test object with input data and the evaluation of the system response. In practice, testing is performed manually and highly depends on the experience of the test engineers and their system knowledge. In industry, traditional function-oriented testing techniques are used, such as requirements-based testing, equivalence class partitioning, and boundary value testing. Such approaches lead to inefficient quality assurance in terms of effort, test coverage, and product quality.

Model-based test approaches have been developed in research to enable the automated generation or selection of test cases from models. They improve the efficiency and the degree of coverage of requirements, design artifacts, and code. Testing on the software design level opens up a new branch in quality assurance with model component testing and subsequent model integration testing. Available test tools usually provide the automated execution of manually pre-defined test cases on the target platform. A widely accepted test tool for the design level is TPT (time partition testing) which supports several modeling environments for test execution, such as Simulink and ASCET. The tool CTE XL is another industrially relevant functional testing tool for the graphical modeling of test cases with equivalence class partitioning, the so-called classification tree method. It also supports combinatorial generation of test cases and test execution on different platforms.

### **2.3 Optimizing Quality Assurance Based on Metrics**

Different approaches exist for optimizing quality assurance activities, such as automation or test case reduction approaches. Another major approach consists of focusing quality assurance based upon metrics that consider data, expert knowledge, or a hybrid form of these two. If the focus is on data, consideration may be given to process, product, or historical data, and different metrics can be applied on these data. For example, typical product metrics are “lines of code”, “number of methods”, or “complexity”. The general idea is to exploit knowledge about the correlations between such product metrics and defect-prone parts in the software. A variety of metrics has been developed in the past, and many evaluations have been performed to identify and exploit such relationships (e.g., [55], [62]). However, there exists no single metric that fits all contexts best for such prioritizations; rather, it depends on the context which metric is suited best.

### 2.3.1 Defect Types

In addition to the aforementioned metrics, we are particularly interested in defect types that different quality assurance techniques are able to find. If such knowledge is available, quality assurance techniques can be applied in a much more focused way, which will help to improve quality assurance in general.

A lot of different defect classifications exist, such as the ODC by IBM [56]. As our current research context is the embedded domain and Matlab Simulink is one of several widely used tools in different embedded domains, we performed a literature survey on quality assurance techniques applied to Matlab Simulink models [57]. Our initial analysis did not focus on a defect type analysis of the quality assurance techniques in the identified papers. Thus, we will present the results of this analysis next together with a short summary of how we identified the papers (for more details, we refer to [57]).

The main goal of our systematic literature review was to provide an overview of existing quality assurance techniques applied to Matlab models, including approaches that already combine static and dynamic quality assurance. In addition, we were interested in the evaluation level and the tool support. After defining the search term, we searched three common databases (IEEE, ACM, and Scopus), excluding and including papers based on titles, abstracts, and full papers, and ended up with a final set of 44 papers. For this article, we analyzed these 44 papers again in order to check whether the quality assurance approaches address particular defect types. Table 1 provides the overview of this analysis. The quality assurance (Q.A.) approach column describes if the paper describes only testing, only analysis, or a combination of analysis and testing, and the focus column describes the defect type aspects on which the papers focus. Of course, some papers only provide general information, or even do not focus on any specific defect type aspect at all.

First of all, about 40% of the papers do not provide any information on how to focus their quality assurance approaches to dedicated defect types. Furthermore, if a paper focuses on defect types, the results are very heterogeneous. For example, [1] talks about reliability and [11] mentions functional properties, which is very general. Other papers mention very concrete defect classes, for instance, [8] mention dead code and the violation of code guidelines. Most of the papers, however, just give examples of defect types and do not provide a complete list of the defect types the quality assurance approaches address. Another reason for the great heterogeneity is that different levels are addressed, such as the relationships between requirements and the Matlab models, among the Matlab models themselves, and with the code generated from the Matlab models. Consequently, no single, unique defect classification exists, nor does it make sense. Instead, knowledge about the variety of different defect types can support quality assurance engineers in improving their requirements, models, and code. For practitioners, existing tools provide a good basis for addressing several defects. However, additional defect classes addressed in research papers can further improve the defect detection ability.

**Table 1.** Overview of papers from the literature review [57] regarding their defect type focus

<i>Ref.</i>	<i>Q.A. approach</i>	<i>Focus (defect types, quality properties)</i>
[1]	A&T	Reliability
[2]	Test	Structural issues (e.g., alterations to operators, constant values or variables), initialization faults, assignment faults, condition check faults and even function/subsystem faults
[3]	Analysis	Numerical errors such as truncation errors and round-off errors as well as sensor errors like quantization and sampling, floating-point numbers with errors, rounding errors, computer arithmetic and sensor errors, approximation errors
[4]	Test	Contradictions, coverage errors, and computational errors
[5]	Test	No focus
[6]	A&T	No focus
[7]	Test	Faulty transitions, design and implementation faults
[8]	Analysis	Dead code, redundant update, modeling guidelines in a model; missing cases in both programming and modeling
[9]	A&T	No focus
[10]	Analysis	Clone detection
[11]	Analysis	Functional properties
[12]	Test	No focus
[13]	Test	No focus
[14]	A&T	No focus
[15]	Test	No focus
[16]	Analysis	Dependencies, functionality, consistency
[17]	Analysis	Improper definition of data types, untranslated model parts, code inefficiencies, faulty translation of arithmetics within Stateflow portions, optimization errors, inappropriate handling of timing
[18]	Analysis	Input and output issues, “islands”, bad pass connections, high block count, high levels, missing connections, bad parameters, mixing of types, non-discrete block usage
[19]	Analysis	Deadlock-freeness and linear temporal logic properties
[20]	Analysis	Properties of a model and run-time errors (e.g., overflow, out of bounds array access, illegal pointer access, division by 0, uncaught exceptions, and other run-time errors)
[21]	Analysis	No focus
[22]	Analysis	Deadlocks in cache coherence protocols and misuse of API rules by third-party device driver code
[23]	Analysis	Consistency
[24]	Analysis	No focus
[25]	Analysis	Illegal arithmetic operations, incomplete or inaccessible model parts, improper fixed-point data type scaling
[26]	Test	No focus

**Table 1.** (Continued.)

<i>Ref.</i>	<i>Q.A. approach</i>	<i>Focus (defect types, quality properties)</i>
[27]	Test	No focus
[28]	Test	Connections between model elements
[29]	Analysis	No focus
[30]	Test	Structural issues
[31]	Test	No focus
[32]	A&T	No focus
[33]	Analysis	Standard compliance violations
[34]	Test	Safety problems
[35]	Test	Structural issues, initialization faults, assignment faults, condition check faults and even function/subsystem faults
[36]	Test	No focus
[37]	Test	No focus
[38]	Analysis	No focus
[39]	A&T	No focus
[40]	Test	Divide by zero and array index out of bounds errors in code, corresponding modeling errors in the MATLAB models
[41]	A&T	Inappropriate arithmetic operators
[42]	Test	No focus
[43]	Test	Numerical overflows
[44]	Analysis	Inconsistencies, implementation errors

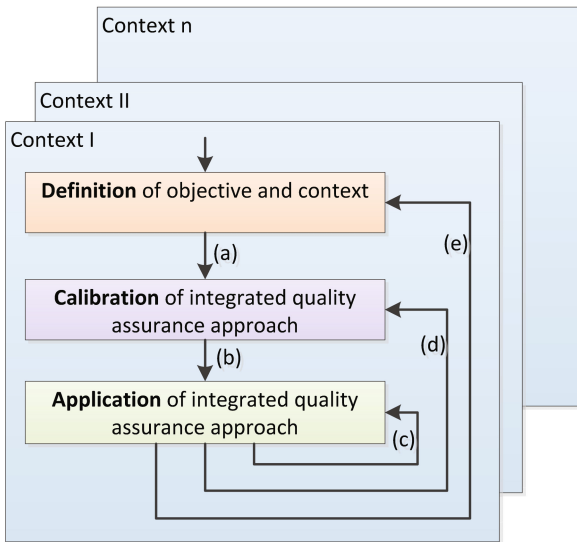
### 3 An Integrated Quality Assurance Methodology

The general idea of an integrated quality assurance methodology is to integrate different analysis and testing techniques to optimize the overall quality of the developed software. Currently, we are developing our methodology with respect to different embedded domains, e.g., automotive, railway, or avionics. This means in particular that during quality assurance, we consider especially Matlab Simulink models and the generated program code of such models. The main goals of such a methodology are (a) to provide a basis for integrating a variety of different static and dynamic quality assurance, and (b) to generate knowledge between the integrated quality assurance techniques for further improving the quality assurance. Such improvements can be, for instance, increasing effectiveness (i.e., find more defects), efficiency (i.e., be faster), or coverage of the system under quality assurance (i.e., assure more parts in more detail).

Fig. 3 shows an overview of the general methodology. Basically, three steps can be distinguished. First of all, the objective and the context have to be clarified. This determines the general setting. Second, the approach has to be calibrated in order to be able to exploit relationships between the different quality assurance techniques. This step is usually performed upon existing historical data, especially based on

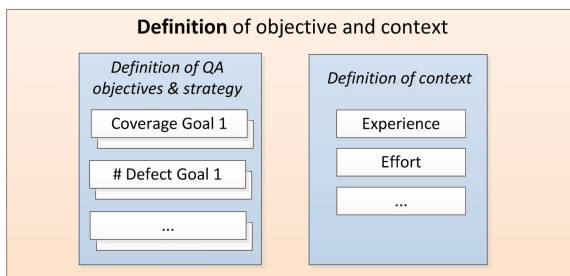


defect data, and based upon defined assumptions. Third, if knowledge about the relationships is gathered and evaluated in the specific context, the integrated quality assurance techniques can be used during development time to perform (and optimize) quality assurance. After one iteration (a-b), more applications can be performed (step c), the calibration can be refined (step d), or the objective may be adapted (step e). Furthermore, the approach can be applied in different contexts; however, for each new context, the knowledge about the relationships between the quality assurance techniques has to be re-evaluated.



**Fig. 3.** Basic steps of an integrated quality assurance methodology

Below, we will describe each of the three main activities in more detail. Fig. 4 shows the **definition**.



**Fig. 4.** The definition activity of the integrated QA methodology

It starts with a definition of the quality assurance (QA) objective and the strategy. The strategy might be to save effort. Concrete objectives may be, for example, to find more defects in less time, to find a certain number of defects, to find certain kinds of

defects, or to achieve a specific coverage of the system under quality assurance. Finally, the context describes the factors that may have an influence on quality assurance, and which later on will determine the environment in which the exploited relationships between different quality assurance techniques used in the integrated approach are valid. Checklists, as provided by Peterson and Wohlin [65], for example, can partially support context definition.

Fig. 5 presents the **calibration**. In this activity, the main goal is to identify relationships between integrated quality assurance techniques.

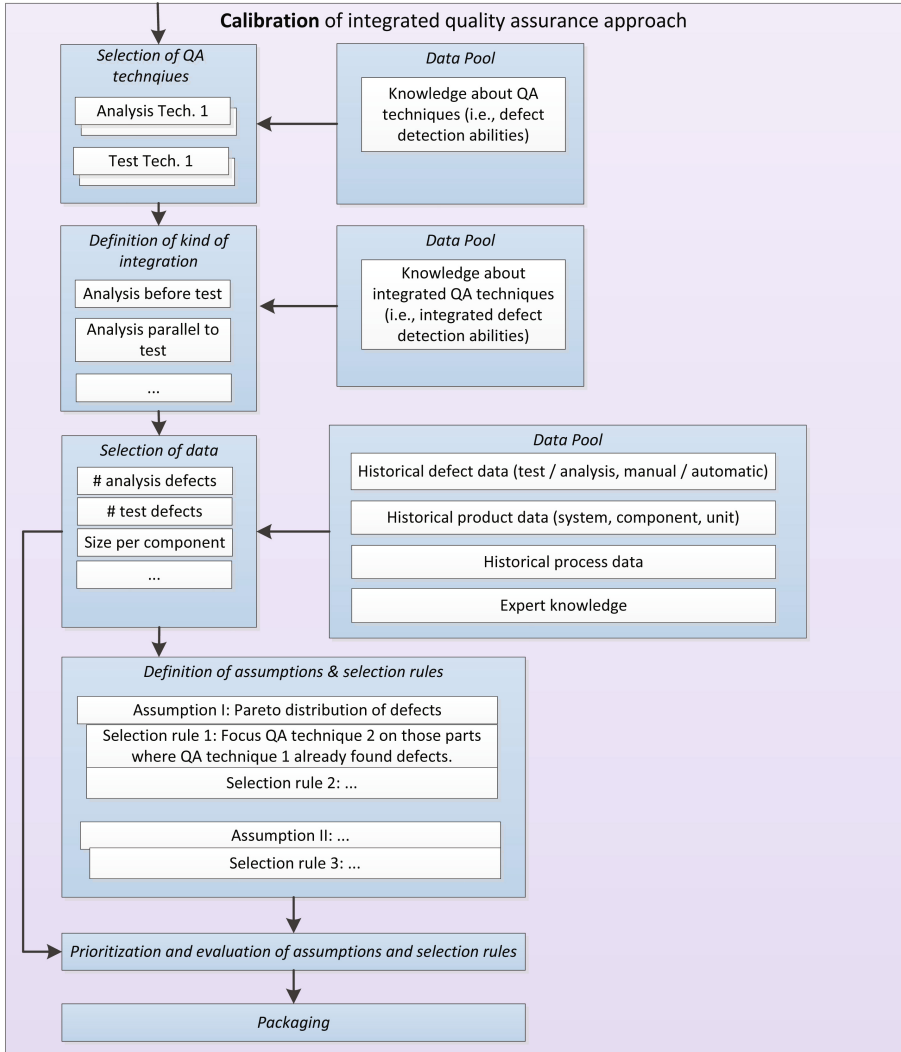


Fig. 5. The calibration activity of the integrated QA methodology

First of all, the quality assurance techniques have to be selected that should be integrated, and the kind of combination has to be determined (i.e., the order). If knowledge about the defect detection abilities is available, this can support the selection (e.g., which quality assurance technique can find which kind of defects, which quality assurance technique is most reasonable in a certain step, etc.). For finding out relationships, historical data is required. If such data is not available, it has to be gathered first by applying the quality assurance techniques in isolation. Assuming that we have a context specific data pool that stores, for instance:

- defect data from applied analysis and testing techniques,
- product data such as the size or complexity of the corresponding software and the models,
- or process data such as the number of check-ins or the number of developers per model,

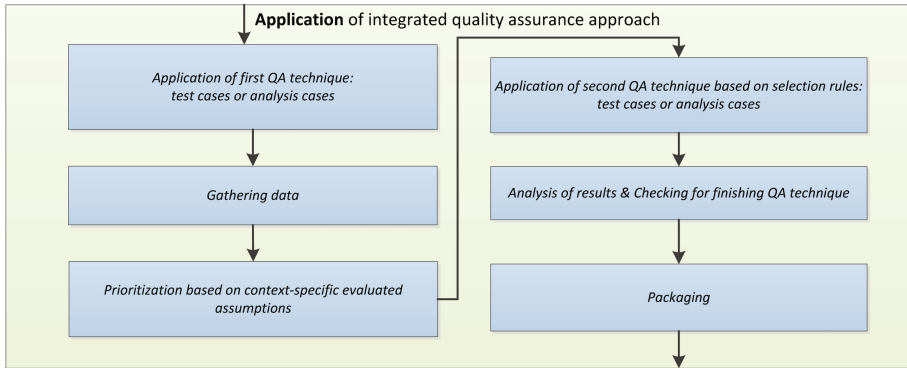
we can extract and select the relevant data for the calibration.

Usually, it is unclear how different analysis and testing techniques may influence each other. Therefore, the next step is to define assumptions and refined selection rules to find out typical relationships that can be exploited. Such assumptions can be derived in different ways, e.g., based upon expert knowledge, or based upon empirical knowledge. One frequently observed defect distribution is the so-called Pareto distribution (i.e. an accumulation of defects in certain parts, such as code classes or model blocks). Following this empirically validated assumption in our context, we can refine this into a concrete selection rule, which says, for example, to focus one quality assurance technique on those parts in which the other quality assurance technique has already found defects. Indeed, such a selection rule can be further refined to make it more applicable, e.g., by defining thresholds for the defect count.

Many other assumptions make sense, too, e.g. regarding historical defect numbers, size or complexity metrics, or additional metrics. After assumptions and refined selection rules have been defined, prioritization, respectively focusing takes place, and the assumptions are evaluated based upon the historical data. Different possibilities exist for judging the significance, starting with a simple significance level where each positive evaluation of a selection rule increases the number by one, to sophisticated calculations of precision/recall and F-measures for each rule [58]. The results are packaged afterwards.

Once the relationships between the selected quality assurance techniques have been identified, they can be exploited in subsequent **applications** in order to optimize the quality assurance. Fig. 6 shows these steps. For the sake of simplicity, we assume here that one quality assurance techniques is applied after the other (and omit parallel execution or more than two quality assurance techniques being applied). The first quality assurance technique is applied as defined and observed in the calibration activity (see Fig. 5). During its application, defect data and, optionally, additional product and process data are gathered, depending on what is needed for evaluating the corresponding assumptions. After the necessary data is available, the prioritization for the second quality assurance activity is done based upon the context-specific

evaluated assumptions. The second quality assurance technique is then applied based upon the refined selection rules until a stopping criterion is achieved (e.g., a certain number of defects are found). The results are analyzed and it is checked again whether the assumption worked in this quality assurance run. Finally, all data is packaged for further analysis and for subsequent applications.



**Fig. 6.** The application activity of the integrated QA methodology

As mentioned above, a decision has to be made about future iterations, i.e., an answer has to be found as to whether

- adaptations of the goals, the context, or the approach in this context should be performed,
- new relationships should be identified,
- the existing knowledge about relationships should be exploited without modifications in subsequent quality assurance runs.

## 4 Proof of Concept

### 4.1 Example from the Embedded Domain

Next, we will provide a more detailed example to show what a concrete instantiation of the integrated quality assurance methodology could look like, focusing on the first two activities (i.e., definition and calibration), and will sketch some additional ideas for extensions.

Fig. 7 shows the example. It starts with the definition activity. The objective is to find at least the same number of defects within less time (and without additional manpower), i.e., quality assurance should be conducted more efficient by a focused procedure. Some of the context factors include the fact that inspectors and testers are well experienced and that we are in the embedded domain. Additional factors are likely to exist.

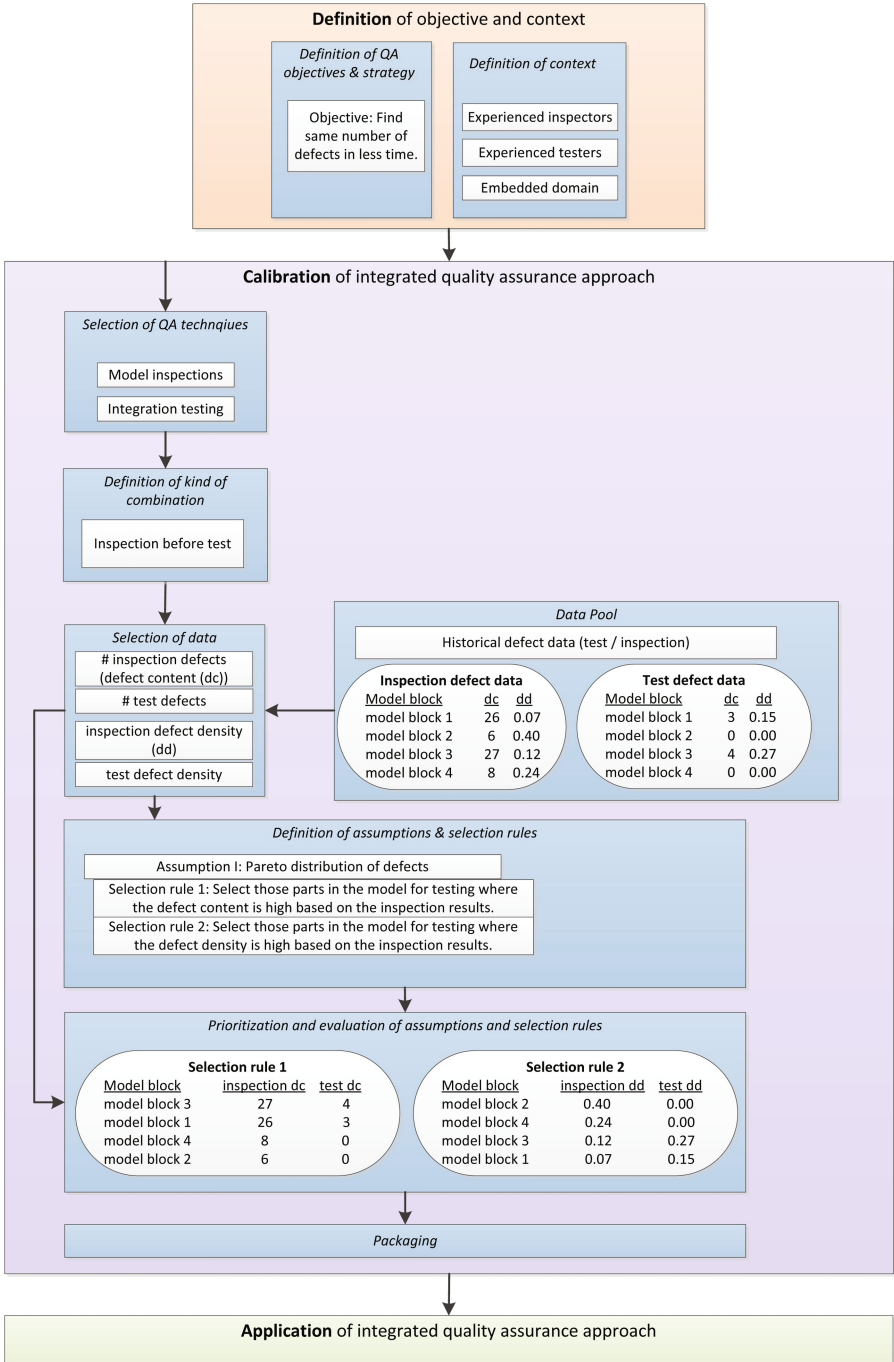


Fig. 7. Example of how to apply the integrated QA methodology

Following the definition of the general setting, calibration takes place. The selected quality assurance techniques are model inspections of Matlab Simulink models, and integration testing; here, inspections should be performed before the test activity. For simplicity, we do not expect specific knowledge about the techniques stored in a database though experts would likely have knowledge about the effectiveness of the selected quality assurance techniques.

We assume that we have data from one already performed quality assurance run (more data would be better in terms of validity). Four model blocks were inspected and tested, and defect content (dc, the number of defects) and defect density (dd, number of defects per size (e.g., block elements, block inheritance depth)) were gathered. We again assume a Pareto distribution of defects and derive two selection rules: one that uses defect content and one that uses defect density. Of course, concrete thresholds might be defined, but as we have no experience, we will omit this initially. Afterwards, we can prioritize the model blocks and evaluate which selection rules are the best ones. In this example, selection rule 1 ranks model blocks 3 and 1 the highest; these also had the largest number of defects during testing. This selection rule fits well as those model parts are prioritized that contains more defects, and its significance can be increased. Selection rule 2, on the other hand, prioritizes model blocks that do not contain more defects during testing, i.e., it is a bad selection rule in our context and thus its significance is very low. These results can be packaged to store the results.

The application then is straightforward: The evaluated assumption, respectively selection rule 1 should be used in the subsequent integration of inspections and testing of models.

Of course, the example is kept quite simple for demonstration purposes. For example, a lot more metrics could be considered, a lot more assumptions and selection rules would make sense, and a much more detailed analysis would increase the knowledge about the relationships between quality assurance activities in the given context (here, we only considered the “top-2” prioritized model blocks). However, the basic concepts remain the same.

In addition, many other instantiations are possible. For example, one could focus more on certain defect types. Different quality assurance techniques can complement each other in finding those defect types at whose detection they are best. Analysis techniques and testing techniques might be applied in reverse order, as the concepts of the integrated quality assurance methodology do not determine the order of the quality assurance techniques, i.e., testing can be done first and the resulting testing outcomes could help to focus inspections or other kinds of analyses on certain parts. Moreover, more than one quality assurance activity can influence a later activity; for instance, results from inspections or static analysis, or other product data may control testing activities.

## 4.2 Lessons Learned

We learned several lessons during the definition of the methodology and in the context of an initial tool landscape we recently set up to apply the integrated approach. The main experiences are listed in the following:

- The general methodology is very generic, which allows instantiating it in many different contexts. In the MBAT project, we are currently conducting 20 case studies with several industrial partners, most of whom follow a combined

quality assurance approach; consequently we will get much feedback from partners in the embedded domain to further improve the approach.

- It is challenging to find suitable relationships between static and dynamic quality assurance approaches. One reason for this is that there is often no explicit database that describes the defect detection abilities of certain quality assurance techniques for selecting the ideal integrated approach, and no or little data is available. When initially starting with the integrated approach, it is a creative step to define assumptions. Existing knowledge from the literature can be a starting point, but such assumptions need to be validated in each new context. Knowledge from experts strongly supports this step.
- Finding reasonable data for the calibration phase is difficult. Historical data is often of minor quality or difficult to extract, and generating new data is a time-consuming endeavor. However, the combined approach can only exploit its full potential if data exist that can be used in the assumptions.
- When setting up a tool landscape aimed at combining different analysis and testing tools, technical problems might occur that will require additional time to fix.
- We started with a similar approach on the code level before, which was easier since it was applied at the same level (i.e., analysis and testing on code). It seems to be more difficult to establish relationships between analysis and testing on the model level, and it becomes even more complicated when additional phases (such as requirements and code) are taken into account.
- Some defect types are known to be found on models, but a general classification of model defect types does not exist, and it is unknown to a certain extent which quality assurance techniques are the best at finding specific kinds of defects. Though we have some basic knowledge, much more research and experience is necessary to optimally focus quality assurance on specific defect types.
- When following such an integrated quality assurance approach, it is still an open question how to balance the pure application and the calibration of the approach. Context factors may change over time, and assumptions that were once valid may become invalid in the future. As our current focus is mainly on the calibration phase, more experience is needed to answer this question.

### 4.3 Implications

The integration of static and dynamic quality assurance techniques provides high benefits during software development. Reduced costs, higher coverage, or improved efficiency can be achieved. This approach can help practitioners in identifying improvement potential for their quality assurance, as analysis and testing techniques are usually performed in isolation. The assumptions mentioned in this paper can be a starting point for focusing quality assurance. Furthermore, general concepts that should be considered are given by the generic integrated methodology.

For researchers, many open questions remain, however. For instance: What kinds of combinations are worthwhile, which relationships exist between the different

quality assurance techniques, how to adapt the approach in a concrete environment, or how to optimally exploit the gathered knowledge regarding the relationships between quality assurance techniques.

## 5 Summary and Outlook

In this paper, we presented a methodology that is able to integrate different static and dynamic quality assurance techniques in order to further improve quality assurance activities. Currently, we are focusing on model-based approaches in the embedded domain, especially on supporting Matlab Simulink models and the corresponding quality assurance. We made initial experiences with the methodology that were positive, but also faced several challenges. Besides technical problems while setting up a tool landscape, one of the most challenging aspects is to identify context-specific assumptions that make it possible to focus and optimize quality assurance activities, i.e., to identify valid relationships between different static and dynamic quality assurance activities to be exploited. With our methodology, we created the basis for future analysis, and provide a process for identifying such relationships. The methodology is kept generic and could be applied in other domains.

Our next major step will be to instantiate and apply the methodology in concrete contexts, as already mentioned above. On the one hand, we expect to get feedback on how to improve the approach. On the other hand, we are very interested in finding out how the different quality assurance techniques influence each other, e.g., which metrics provide valuable results, and which assumptions and selection rules provide the highest benefit. Currently, we set up a tool landscape that integrates a research prototype for generating test models, the analysis tool Design Verifier from the Matlab toolbox, and the test tool TPT. In addition, tool prototypes that analyze defect data and present different representations of the data are integrated. We aim at obtaining defect and product data when analyzing different models, and expect to identify which analysis and testing techniques find which kinds of defects, and where the defects are located. Such knowledge may be one more step towards combining quality assurance techniques in a reasonable way, and may be a starting point for combined quality assurance in other embedded environments.

**Acknowledgments.** Parts of this work have been funded by the ARTEMIS project “MBAT” (grant: 269335). We would also like to thank Sonnhild Namingha for proofreading.

## References

1. Garro, A., Tundis, A.: A model-based method for system reliability analysis. In: 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium, pp. 1–8 (2012)
2. Zhan, Y., Clark, J.A.: A search-based framework for automatic testing of MATLAB/Simulink models. *Journal of Systems and Software* 81(2), 262–285 (2008)
3. Chapoutot, A., Martel, M.: Abstract Simulation: A Static Analysis of Simulink Models. In: *International Conference on Embedded Software and Systems*, pp. 83–92 (2009)



4. Boden, L.M., Busser, R.D.: Adding natural relationships to Simulink models to improve automated model-based testing. In: Digital Avionics Systems Conference, vol. 2, pp. 1–9.
5. Cleaveland, W.R., Smolka, S.A., Sims, S.T.: An instrumentation-based approach to controller model validation. In: Broy, M., Krüger, I.H., Meisinger, M. (eds.) ASWSD 2006. LNCS, vol. 4922, pp. 84–97. Springer, Heidelberg (2008)
6. Peranandam, P., Raviram, S., Satpathy, M., Yeolekar, A., Gadkari, A., Ramesh, S.: An integrated test generation tool for enhanced coverage of Simulink/Stateflow models. In: Design, Automation Test in Europe Conference Exhibition, pp. 308–311 (2012)
7. Siegl, S., Hielscher, K.-S., German, R., Berger, C.: Automated testing of embedded automotive systems from requirement specification models. In: 12th IEEE Latin-American Test Workshop (2011)
8. Sims, S., Cleaveland, R., Butts, K., Ranville, S.: Automated Validation of Software Models. In: 16th IEEE International Conference on Automated Software Engineering, p. 91 (2001)
9. Mohalik, S., Gadkari, A.A., Yeolekar, A., Shashidhar, K.C., Ramesh, S.: Automatic test case generation from Simulink / Stateflow models using model checking (2013)
10. Deissenboeck, F., Hummel, B., Jürgens, E., Schätz, B., Wagner, S., Girard, J.-F., Teuchert, S.: Clone detection in automotive model-based development. In: 30th International Conference on Software Engineering, pp. 603–612 (2008)
11. Boström, P.: Contract-based verification of simulink models. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 291–306. Springer, Heidelberg (2011)
12. Broy, M., Chakraborty, S., Goswami, D., Ramesh, S., Satpathy, M., Resmerita, S., Pree, W.: Cross-layer analysis, testing and verification of automotive control software. In: 9th ACM International Conference on Embedded Software, pp. 263–272 (2011)
13. Satpathy, M., Yeolekar, A., Peranandam, P., Ramesh, S.: Efficient coverage of parallel and hierarchical stateflow models for test case generation. *Software Testing Verification and Reliability* 22(7), 457–479 (2012)
14. Barnat, J., Brim, L., Beran, J., Kratochvíla, T., Oliveira, Í.R.: Executing model checking counterexamples in Simulink. In: 6th International Symposium on Theoretical Aspects of Software Engineering, pp. 245–248 (2012)
15. Sims, S., DuVarney, D.C.: Experience report: the reactis validation tool. *SIGPLAN* 42(9), 137–140 (2007)
16. Merschen, D., Polzer, A., Botterweck, G., Kowalewski, S.: Experiences of applying model-based analysis to support the development of automotive software product lines. *ACM Int. Conference Proceeding Series*, pp. 141–150 (2011)
17. Stürmer, I., Conrad, M., Fey, I., Dörr, H.: Experiences with model and autocode reviews in model-based software development. In: *Int. Conf. on Software Eng.*, pp. 45–51 (2006)
18. Kemmann, S., Kuhn, T., Trapp, M.: Extensible and automated model-evaluations with iNProVE. In: Kraemer, F.A., Herrmann, P. (eds.) SAM 2010. LNCS, vol. 6598, pp. 193–208. Springer, Heidelberg (2011)
19. Chen, C.: Formal analysis for stateflow diagrams. In: 4th IEEE International Conference on Secure Software Integration and Reliability Improvement Companion, pp. 102–109 (2010)
20. Popovici, K., Lalo, M.: Formal model and code verification in Model-Based Design. In: *Joint IEEE North-East Workshop on Circuits and Systems and TAISA Conference* (2009)
21. Toyn, I., Galloway, A.: Formal validation of hierarchical state machines against expectations. In: *Australian Software Engineering Conference*, pp. 181–190 (2007)
22. Alur, R.: Formal verification of hybrid systems. In: 9th ACM International Conference on Embedded software, pp. 273–278 (2011)

23. Kanade, A., Alur, R., Ivančić, F., Ramesh, S., Sankaranarayanan, S., Shashidhar, K.C.: Generating and analyzing symbolic traces of simulink/Stateflow models. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 430–445. Springer, Heidelberg (2009)
24. Whalen, M., Cofer, D., Miller, S., Krogh, B.H., Storm, W.: Integration of formal analysis into a model-based software development process. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 68–84. Springer, Heidelberg (2008)
25. Hu, W., Wegener, J., Stürmer, I., Reicherdt, R., Salecker, E., Glesner, S.: MeMo - methods of model quality. In: Workshop Modellbasierte Entwicklung eingebetteter Systeme, pp. 127–132 (2011)
26. Böhr, F.: Model Based Statistical Testing of Embedded Systems. In: 4th International Conference on Software Testing, Verification and Validation Workshops, pp. 18–25 (2011)
27. Bringmann, E., Krämer, A.: Model-based testing of automotive systems. In: 1st International Conference on Software Testing, Verification and Validation, pp. 485–493 (2008)
28. Morschhauser, I., Lindvall, M.: “Model-Based Validation Verification Integrated with SW Architecture Analysis: A Feasibility Study. In: IEEE Aerospace Conference, pp. 1–18 (2007)
29. Mazzolini, M., Brusaferrri, A., Carpanzano, E.: Model-checking based verification approach for advanced industrial automation solutions. In: 15th IEEE International Conference on Emerging Technologies and Factory Automation (2010)
30. Brillout, A., He, N., Mazzucchi, M., Kroening, D., Purandare, M., Rümmer, P., Weissenbacher, G.: Mutation-based test case generation for simulink models. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 208–227. Springer, Heidelberg (2010)
31. Satpathy, M., Yeolekar, A., Ramesh, S.: Randomized directed testing (REDIRECT) for Simulink/Stateflow models. In: 8th ACM International Conference on Embedded Software, pp. 217–226 (2008)
32. Schwarz, M.H., Sheng, H., Batchuluun, B., Sheleh, A., Chaaban, W., Borcsok, J.: Reliable software development methodology for safety related applications: From simulation to reliable source code. In: XXII International Symposium on Information, Communication and Automation Technologies, pp. 1–7 (2009)
33. Farkas, T., Grund, D.: Rule Checking within the Model-Based Development of Safety-Critical Systems and Embedded Automotive Software. In: 8th International Symposium on Autonomous Decentralized Systems, pp. 287–294 (2007)
34. Zhan, Y., Clark, J.: Search based automatic test-data generation at an architectural level. In: Deb, K., Tari, Z. (eds.) GECCO 2004. LNCS, vol. 3103, pp. 1413–1424. Springer, Heidelberg (2004)
35. Zhan, Y., Clark, J.A.: Search-based mutation testing for Simulink models. In: Conference on Genetic and Evolutionary Computation, pp. 1061–1068 (2005)
36. Windisch, A.: Search-based test data generation from stateflowstatecharts. In: 12th Annual Conference on Genetic and Evolutionary Computation, pp. 1349–1356 (2010)
37. Böhr, F., Eschbach, R.: SIMOTEST: A tool for automated testing of hybrid real-time Simulink models. In: Symposium on Emerging Technologies and Factory Automation (2011)
38. Reicherdt, R., Glesner, S.: Slicing MATLAB simulink models. In: Int. Conference on Software Engineering, pp. 551–561 (2012)

39. Alur, R., Kanade, A., Ramesh, S., Shashidhar, K.C.: Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In: 8th ACM Int. Conf. on Embedded Softw., pp. 89–98 (2008)
40. Venkatesh, R., Shrotri, U., Darke, P., Bokil, P.: Test generation for large automotive models. In: IEEE International Conference on Industrial Technology, pp. 662–667 (2012)
41. He, N., Rümmer, P., Kroening, D.: Test-case generation for embedded simulink via formal concept analysis. In: 48th Design Automation Conference, pp. 224–229 (2011)
42. Zhan, Y., Clark, J.A.: The state problem for test generation in Simulink. In: 8th Annual Conference on Genetic and Evolutionary Computation, pp. 1941–1948 (2006)
43. Oh, J., Harman, M., Yoo, S.: Transition coverage testing for simulink/stateflow models using messy genetic algorithms. In: 13th Annual Conference on Genetic and Evolutionary Computation, pp. 1851–1858 (2011)
44. Ray, A., Morschhaeuser, I., Ackermann, C., Cleaveland, R., Shelton, C., Martin, C.: Validating Automotive Control Software Using Instrumentation-Based Verification. In: IEEE/ACM International Conference on Automated Software (2009)
45. Kim, M., Kim, Y., Jang, Y.: Industrial Application of Concolic Testing on Embedded Software: Case Studies. In: 5th International Conference on Software Testing, Verification and Validation, pp. 390–399 (2012)
46. Elberzhager, F., Münch, J., Rombach, D., Freimut, B.: Integrating Inspection and Test Processes based on Context-specific Assumptions. *Journal of Software: Evolution and Processes* (2012)
47. Elberzhager, F., Muench, J., Tran, V.: A Systematic Mapping Study on the Combination of Static and Dynamic Quality Assurance Techniques. *Information and Software Technology* 54(1), 1–15 (2012)
48. Zimmerman, D.M., Kiniry, J.R.: A Verification-centric Software Development Process for Java. In: 9th International Conference on Quality Software, pp. 76–85 (2009)
49. Chen, Q., Wang, L., Yang, Z., Stoller, S.D.: HAVE: Detecting Atomicity Violations via Integrated Dynamic and Static Analysis. In: 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, pp. 425–439 (2009)
50. Runeson, P., Andrews, A.: Detection or Isolation of Defects? An Experimental Comparison of Unit Testing and Code Inspection. In: 14th International Symposium on Software Reliability Engineering, pp. 3–13 (2003)
51. Juristo, N., Vegas, S.: Functional Testing, Structural Testing, and Code Reading: What Fault Type do they each Detect? In: Empirical Methods and Studies in Software Engineering, pp. 208–232 (2003)
52. Chen, J., Zhou, H., Bruda, S.D.: Combining Model Checking and Testing for Software Analysis. In: International Conference on Computer Science and Software Engineering, pp. 206–209 (2008)
53. Juristo, N., Moreno, A.M., Vegas, S.: Reviewing 25 Years of Testing Technique Experiments. *Empirical Software Engineering* 9(1-2), 7–44 (2004)
54. Aurum, A., Petersson, H., Wohlin, C.: State-of-the-Art: Software Inspections after 25 Years. *Software Testing, Verification and Reliability* 12(3), 133–154 (2002)
55. Genero, M., Piattini, M., Calero, C.: A Survey of Metrics for UML Class Diagrams. *Journal of Object Technology* 4(9) (2005)
56. ODC. Orthogonal Defect Classification v5.11, IBM (2002), <http://www.research.ibm.com/softeng/ODC/ODC.HTM>
57. Elberzhager, F., Rosbach, A., Bauer, T.: Analysis and Testing of Matlab Simulink Models: A Systematic Mapping Study. In: JAMAICA Workshop (ISSTA) (accepted, 2013)

58. Arisholm, E., Briand, L.C., Johannesson, E.B.: A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software* 83(1), 2–17 (2009)
59. Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, IEC 61508 Standard
60. Road vehicles – Functional safety, ISO 26262 Standard
61. AUTomotive Open System Architecture (AUTOSAR), <http://www.autosar.org/>
62. Radjenovic, D., Hericko, M., Torkar, R., Zivkovic, A.: Software fault prediction metrics: A systematic literature review. *Information and Software Technology Journal* (accepted, 2013)
63. Jezequel, J.-M., Meyer, B.: Put it in the contract: The lessons of Ariane, <http://www.irisa.fr/pampa/EPEE/Ariane5.html> (accessed on June 17, 2013)
64. Leveson, N., Turner, C.S.: An Investigation of the Therac-25 Accidents, [http://courses.cs.vt.edu/~cs3604/lib/Therac\\_25/Therac\\_1.html](http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html) (accessed on June 17, 2013)
65. Petersen, K., Wohlin, C.: Context in industrial software engineering research. In: 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 401–404 (2009)

# Effects of Test-Driven Development: A Comparative Analysis of Empirical Studies

Simo Mäkinen and Jürgen Münch

University of Helsinki,  
Department of Computer Science,  
P.O. Box 68 (Gustaf Hällströmin katu 2b),  
FI-00014 University of Helsinki, Finland  
{simo.makinen,juergen.muench}@cs.helsinki.fi

**Abstract.** Test-driven development is a software development practice where small sections of test code are used to direct the development of program units. Writing test code prior to the production code promises several positive effects on the development process itself and on associated products and processes as well. However, there are few comparative studies on the effects of test-driven development. Thus, it is difficult to assess the potential process and product effects when applying test-driven development. In order to get an overview of the observed effects of test-driven development, an in-depth review of existing empirical studies was carried out. The results for ten different internal and external quality attributes indicate that test-driven development can reduce the amount of introduced defects and lead to more maintainable code. Parts of the implemented code may also be somewhat smaller in size and complexity. While maintenance of test-driven code can take less time, initial development may last longer. Besides the comparative analysis, this article sketches related work and gives an outlook on future research.

**Keywords:** test-driven development, test-first programming, software testing, software verification, software engineering, empirical study.

## 1 Introduction

Red. Green. Refactor. The mantra of test-driven development [1] is contained in these words: *red* refers to the fact that first and foremost implementation of any feature should start with a failing test, *green* signifies the need to make that test pass as fast as possible and *refactor* is the keyword to symbolize that the code should be cleaned up and perfected to keep the internal structure of the code intact. But the question is, what lies behind these three words and what do we know about the effects of following such guidelines? Test-driven development reshapes the design and implementation of software [1] but does the change propagate to the associated software products and in which way are the processes altered with the introduction of this alternative way of development? The objective here was to explore these questions and to get an overview of the observed effects of test-driven development.

To seek out answers to these questions, we performed an integrative literature review and analyzed the experiences from existing empirical studies from the industry and academia that reported on the effects of test-driven development. A quality map containing ten quality attributes was formed using the data from the studies and the effects were studied in greater detail. The results of the comparative analysis suggest that test-driven development can affect quality attributes. Positive effects were reported in particular for defect densities and maintainability in industrial environments. Some studies, however, reported an increased development effort at the same time. Many of the other observed effects are neutral or inconclusive.

This paper has been divided into five sections. Related work is presented in Section 2 and the research method for this review is described in Section 3. The effects derived from the reviewed primary studies are detailed in Section 4 and finally Section 5 concludes the paper.

## 2 Related Work

Test-driven development has been subject of reviews before. For instance, Turhan et al. [2] performed a systematic literature review on test-driven development that highlighted internal and external quality aspects. The review discovered that during the last decade, there have been hundreds of publications that mention test-driven development but few report empirically viable results.

Based on the reports, Turhan et al. were able to draw a picture of the overall effect test-driven development might have. They categorized their findings as internal and external quality, test quality and productivity. Individual metrics were assigned to these categories which were labeled to have *better*, *worse*, *mixed* or *inconclusive* effects. The rigor of each study was assessed by looking at the experimental setup and studies were further categorized into four distinct rigor levels.

Results from the review of Turhan et al. for each of the categories indicate that the effects vary. Internal quality—which consisted of size, complexity, cohesion and other product metrics—was reported to increase in several cases but more studies were found where there either was no difference or the results were mixed or worse. External quality, however, was seen to be somewhat higher as the majority of reviewed studies showed that the amount of defects dropped; relatively few studies showed a decrease in external quality or were inconclusive. The effect of test-driven development on productivity wasn't clear: most industrial studies reported lowered productivity figures while other experiments had just the opposite results or were inconclusive. Surprisingly, Turhan et al. conclude that test quality, which means such attributes as code coverage and testing effort, was not superior in all cases when test-driven development was used. Test quality was considered better in certain studies but some reported inconclusive or even worse results.

A few years earlier, Jeffries and Melnik wrote down a short summary of existing studies about test-driven development [3]. The article covered around twenty

studies both from the industry and academia, describing various context factors such as the number of participants, programming language and the duration for each study. The reported effects were categorized into *productivity* and generic *quality effects* which included defect rates as well as perceptions of quality.

Jeffries and Melnik summarize that in most of the industrial studies, more development effort was needed when the organizations took test-driven development into use. In the academic environment, effort increases were noticed as well but in some academic experiments test-driven development was seen to lead to reduced effort levels. As for the quality effects, a majority of the industrial studies showed a positive effect on the amount of defects and in certain cases the differences to previous company baselines were quite significant. Fewer academic studies reported reduced defect rates and the results were not quite as significant; in one, defect rates actually went up with test-driven development.

Recently, Rafique and Mišić [4] gathered experiences from 25 test-driven development studies in a statistical meta-analysis which focused on the dimensions of external quality and productivity. Empirical results from existing studies were used as much as data was available from the primary studies. The studies were categorized as academic or industrial and the development method of the respective control groups was noted as well. It seemed to matter which development method the reference group was using in terms of how effective test-driven development was seen to be.

Rafique and Mišić conclude that external quality could be seen to have improved with test-driven development in bigger, and longer, industrial projects but the same effect was not noticed in all academic experiments. For productivity, the results were indecisive and there was a bigger gap between the academic experiments and industrial case studies than with external quality. Desai et al. [5] came to a similar conclusion in their review of academic studies: some aspects of internal and external quality saw improvement but the results for productivity were mixed—experience of the students was seen as a factor in the experiments.

All of the previous reviews offer valuable insights into the effects of test-driven development as a whole by gathering information from a number of existing studies. While the research method is similar to the aforementioned reviews, in this review the idea is to extend previous knowledge by breaking down the quality attributes to more atomic units as far as data is available from the empirical studies. We expect that this will lead to deeper understanding of the effect test-driven development has on various attributes of quality.

### 3 Research Method

Empirical studies in software engineering can be conducted by using research methods which support the collection of empirical findings in various settings [6]. Runeson and Höst write that surveys, case studies, experiments and action research serve different purposes: surveys are useful as descriptive methods, case studies can be used for exploring a phenomena whereas experiments can at best be explanatory while action research is a flexible methodology that can be used

for improving some aspects that are related to the research focus [6]. Test-driven development has been the object of study in a number of research endeavours that have utilized such methods.

Literature reviews can be used for constructing a theoretical framework for a study which helps to formulate a problem statement and in the end identify the purpose of a study [7]. In integrative literature reviews [8], existing literature itself is the object of study and emerging or mature topics can be analyzed in more detail in order to create new perspectives on the topic. Systematic literature reviews [9] have similar objectives as integrative reviews but stress that the review protocols, search strategies and finally both inclusion criteria and exclusion criteria are explicitly defined.

The research method in this study is an integrative literature review which was performed to discover a representative sample of empirical studies about the effects of test-driven development from the industry and academia alike. Creswell [10] writes that literature reviews should start by identifying the keywords to use as search terms for the topic. Keywords that were used were such as *test driven development*, *test driven*, *test first programming* and *test first*. The second step suggested by Creswell is to apply these search terms in practice and find relevant publications from catalogs and publication databases. Several search engines from known scientific publishers were used in the process, namely the keywords were entered into search fields at the digital libraries of the Institute of Electrical Engineers (IEEE), Association of Computer Machinery (ACM), Springer and Elsevier. The titles and abstracts of the first few hundred highest-ranked entries from each service were manually screened. Although the search was repeated several times with the aforementioned keywords with multiple search engines on different occasions, the review protocol wasn't entirely systematic since there wasn't a single, exact, search string and entries were not evaluated if they had a low rank in the search results.

A selection of relevant publications from a larger body requires that there is at least some sort of inclusion and exclusion criteria. Merriam [7] suggests that the criteria can for instance include the consideration of the seminality of the author, date of publication, topic relevance to current research and the overall quality of the publication in question. There was no strict filter according to the year of publication so the main criterion was whether the publication included empirical findings of test-driven development and presented results either from the industry or academia. The quality of publications and general relevance were used as exclusion criteria in some cases. While the objective was to gather all the relevant research, there was a limited number of publications that could be reviewed in greater detail due to the nature of the study.

After applying the criteria, 19 publications remained to be analyzed further. These publications, listed in Table 1, were conference proceedings and journal articles that had relevant empirical information about test-driven development. In 2009, Turhan et al. [2] identified 22 publications in their systematic literature review of test-driven development and 7 of these publications are also included in



**Table 1.** An overview of the publications included in the review and the effects of test-driven development on quality factors

Author Name and Year	Context	Defects	Coverage	Complexity	Coupling	Cohesion	Size	Effort	External Quality	Productivity	Maintainability
Bhat and Nagappan 2006 [12]	Industry	x <sup>+</sup>	x					x	x		
Canfora et al. 2006 [13]	Industry								x <sub>-</sub>	x <sub>-</sub>	
Dogša and Batič 2011 [14]	Industry	x <sup>+</sup>	x			x		x <sub>-</sub>	x <sup>+</sup>	x <sub>-</sub>	x <sup>+</sup>
George and Williams 2003 [15]	Industry		x						x <sub>-</sub>	x <sup>+</sup>	x <sub>-</sub>
Geras et al. 2004 [16]	Industry	x	x					x		x	
Maximilien and Williams 2003 [17]	Industry	x <sup>+</sup>				x					
Nagappan et al. 2008 [18]	Industry	x <sup>+</sup>	x			x		x <sub>-</sub>			
Williams et al. 2003 [19]	Industry	x <sup>+</sup>				x					
Janzen and Saedian 2008 [20]	Industry/Academia	x	x <sup>+</sup>	x	x	x <sup>+</sup>					
Madeyski and Szała 2010 [21]	Industry/Academia							x		x	
Müller and Höfer 2007 [22]	Industry/Academia	x				x		x	x	x	
Desai et al. 2009 [23]	Academia		x					x	x	x	
Gupta and Jalote 2007 [24]	Academia							x <sup>+</sup>	x	x	
Huang and Holcombe 2009 [25]	Academia							x <sub>-</sub>	x	x	
Janzen and Saedian 2006 [26]	Academia	x	x	x		x		x	x	x	
Madeyski 2010 [27]	Academia	x				x					
Pančur and Ciglarič 2011 [28]	Academia	x	x						x	x	
Vu et al. 2009 [29]	Academia	x	x	x <sup>+</sup>		x		x	x	x	
Wilkerson et al. 2012 [30]	Academia	x <sub>-</sub>						x			

this integrative review but some of the previously identified publications remain outside the analysis.

The findings of the studies of this review were used to construct a map of quality attributes and the perceived effects noted in each study. Construction of the quality map proceeded so that an attribute was added to the map if a particular study contained empirical data about the attribute so the attributes were not predetermined. This map lead to a more detailed analysis of the individual attributes. The literature review itself was completed in 2012 [11].

## 4 Effects of Test-Driven Development

Test-driven development has had a role in certain empirical studies but how are the different software engineering areas affected in practice? This section describes the results of the literature review and gathers the information about the perceived effects from the empirical studies included in the review.

Quality attributes of products, processes and resources can be related to the internal quality of objects or to the external quality which requires the consideration of not just the object but its external circumstances as well [31]. These

attributes can be either direct and readily measured or indirectly composed of several different attributes.

In the review, empirical findings for 10 different, internal and external, quality attributes were gathered from the 19 reviewed research reports which covered the effects of test-driven development. The extracted attributes or qualities which are analyzed further below include the following: the amount and density of defects, code coverage, code complexity, coupling, cohesion, size, effort, external quality, productivity and finally maintainability.

An overview of the results indicates that various effects on the quality attributes have been recorded in the test-driven development studies. The quality map which summarizes the effects is illustrated in Table 1 where the included publications are sorted by author and the study context which here is either industry or academia or a mixture of both. Case studies and experiments with industrial professionals in their own environment are considered industrial studies. Controlled and quasi-controlled experiments with student subjects are categorized as academic.

Across all the studies in the review, there are a total of 73 reported quality attribute effects that are shown in the quality map. Out of these effects, 12 have been labeled as significant positive effects whereas there are 9 effects labeled significantly negative. The rest of the 52 effects are either neutral or inconclusive. Significant positive effects mean that there was a statistically significant difference between the development practices compared in the study and the difference was in favor of test-driven development or there was enough significant qualitative data that suggested a meaningful difference. Significant negative effects means that test-driven development was in similar terms considered worse than other development practices it was compared against. In the quality map, significant positive effects are denoted by a superscript plus symbol  $x^+$  and significant negative effects by a subscript minus symbol  $x_-$ ; an undecorated  $x$  symbol signifies inconclusiveness, neutrality or the fact that the attribute was merely mentioned in the study results but not necessarily contemplated further as was the case with the size of code products in many reports. The following in-depth analysis of the effects and individual quality attributes shows what kind of role test-driven development can play in the software engineering process. A coarse-grained aggregation of the results is given at the end of the section.

#### 4.1 Defect Density and the Number of Defects

In test-driven development, test code is written before the implementation which could theoretically lead to a reduced number of defects as fewer sections of code remain untested. Indeed, several industrial case studies show a relatively large reduction of defects compared to sister projects in the organizations where the trials were carried out.

Defect densities have reduced in a number of organizations and projects of different size. At IBM, a team that employed test-driven development for the first time was able to create software that had only half the defect density of previous comparable projects, and on average there were only around four defects

per one thousand lines of code [15]. While the overall amount of reported defects dropped, there were still the same relative amount of severe defects as with the previous release that didn't use test-driven development methods [19]. Microsoft development teams were able to cut down their defect rates, as well, and the selected test-driven projects had a reduced defect density of sixty to ninety percent compared to similar projects [18]. In the telecommunications field, test-driven development seemed to slightly improve quality by reductions in defect densities before and especially after releases [14].

Outside the industry, test-driven development hasn't always lead to drastically better outcomes than other development methods. Geras et al. experimented with industry professionals [16] but in a setting that resembled an experiment rather than a case study and didn't notice much of a difference in defect counts between experimental teams that were instructed to use or not to use test-driven development in their task. Student developer teams reported in the research of Vu et al. [29] fared marginally better in terms of the amount of defects when teams were using test-driven development but it seems that process adherence was relatively low and the designated test-driven team wrote less test code than the non-test-driven team. In another student experiment, Wilkerson et al. [30] noticed that code inspections were more effective at catching defects than test-driven development; that is, more defects remained in the code that was developed using test-driven development than in the code that was inspected by a code inspection group.

## 4.2 Code Coverage

Covering a particular source code line requires that there is a corresponding block of executable test code that in turn executes the source code line. Besides the basic statement coverage, it is also possible to measure how well the test code covers code blocks or branches in the code [32]. Without test code, there is no coverage and the code product must be tested with other methods or left untested. Test-driven development should encourage the developers to write scores of tests which should lead to a high code coverage.

The message from the industry seems to be that development teams were able to achieve good coverage levels when they were using test-driven development. At Microsoft and IBM, block coverages were up to 80 and 90 percent for several projects which took advantage of test-driven development although for one larger project the block coverage was around 60 percent [18]. In the longitudinal case study of Janzen et al. [20], coverage for the products of an industry partner were reported to be on the same high levels as the projects at Microsoft and IBM. George and Williams observed similar coverage ratings in a shorter industry experiment earlier [19].

When development teams are writing tests after the implementation and not focusing on incremental test-driven development, coverage ratings tend to be lower although the team's prior exposure to test-driven development can affect the way the developers behave in future projects [20]. Experience of the developers in general seems to be a factor in explaining how individual developers

work and how well they're able to adhere to the test-driven development process [22,27]. Students who are more unfamiliar with the concept might not be able to achieve as high a coverage as their industry peers [29]. Very low coverage rates might be a sign that design and implementation is not done incrementally with the help of tests.

Good coverage doesn't necessarily mean that the tests are able to effectively identify incorrect behavior. Mutation testing consists of transforming the original source code with mutation operators; good tests should detect adverse mutations to the source code [32]. Pančur and Ciglarič [28] noticed in their student experiment that even though the branch coverage was higher for the test-driven development students, the mutation score indicator was actually worse than the score of other students who were developing their code with a test-last approach. In another student experiment, the mutation score indicators were more or less equal [27].

### 4.3 Complexity

Complexity measures of code products can be used to describe individual code components and their internal structure. For instance, McCabe's complexity [33] is calculated from the relative amount of branches in code while modern views of complexity take the structure of methods and classes into account [34]. Because code is developed in small code fragments with test-driven development, a reduction in the complexity of code is possible.

Reductions in complexity of classes have been observed both in the industry and academia but not all results are conclusive. Classes created by industrial test-driven developers seem to be more coherent and contain fewer complex methods [20] or the products seem less complex overall [14]. Student developers have on occasion constructed less complex classes with test-driven development [20,29] but in some cases the differences in complexity between development methods have been small [28].

### 4.4 Coupling and Cohesion

Coupling and cohesion are properties of object-oriented code constructs and measure the interconnectivity and internal consistency of classes [34]. Through incremental design, test-driven development could encourage developers to create classes which have more distinct roles.

Relatively few test-driven development studies have reported coupling or cohesion metrics: coupling measures, for instance, are not entirely straightforward to analyze as classes can be naturally interconnected in an object-oriented design pattern [20]. In an industrial case study, Janzen et al. noted that coupling was actually greater in the project which utilized test-driven development [20] and the experiences were similar in another student study [26]. As for cohesion, classes constructed by test-driven developers can have fine cohesion figures but the effect is not reported to be constant across all cases and at times cohesion has been lower when test-driven development has been used [20].

## 4.5 Size

Besides examining the structure of code products and the relationships of objects, it is possible to determine the size of these elements. The amount of source code lines is one size measure which can be used to characterize code products. Test-driven development involves writing a considerable amount of automated unit tests that contribute to the overall size of the code base but the incremental design and implementation could have an effect on the size of the classes and methods written.

The ratio between test source code lines and non-test source code lines is one way to look at the relative size of test code. The studies from the industry show that the ratios can be substantial in projects where test-driven development is used. At Microsoft, the highest ratings were reported to be at 0.89 test code lines/production code lines, lowest at 0.39 test code lines/production code lines for a larger project and somewhere in between for other projects depending on the size of the project [18]. The numbers from IBM fall within this range at around 0.48 test code lines/production code lines [17]. Without test-driven development and with proper test-last principles, it is possible to reach fair ratios but the ratios tend to fall behind test-driven projects [14]. For student developers, the ratios have been observed to be on the same high level as industry developers [27] or somewhat lower, albeit in one case students were able to achieve a ratio of over 1 test code lines/production code lines without test-driven development [29]. It seems to be apparent that with test-driven development, the size of the overall code base increases due to the tests written if for every two lines of code there is at least a line of test code.

The size of classes and methods has in certain cases been affected under test-driven development. In the longitudinal industry study of Janzen [20], classes and methods were reported to be smaller although the same didn't apply to several other case studies mentioned in the report. Similarly, students wrote lighter classes and methods in one case [20] but not in another [26]. Madeyski and Szala found in a quasi-controlled experiment that there was less code per user story when the developer was designing and implementing the stories with test-driven development [21]. Müller and Höfer conclude from an experiment that test-driven developers with less experience don't necessarily create test code which has a larger code footprint but there might be some differences in the size of the non-test code in favor of the experts [22].

## 4.6 Effort

Effort is a process quality attribute [31] and here, it can be defined as the amount of exertion spent on a specific software engineering task. Typically, there is a relation between effort and duration and with careful consideration of the context the duration of a task could be seen as an indicator for effort spent. Test-driven development changes the design and implementation process of code products so the effort of these processes might be affected. The effects might not be limited to these areas as the availability of automated tests is related to verification and validation activities as well.

Writing the tests seems to take time or then there are other factors which affect the development process as there have been experiences which suggest that the usage of test-driven development increases the effort. At Microsoft and IBM, managers estimated that development took about 15 to 30 percent longer [18]. Effort also increased in the study of Dogša and Batič [14] where the development took around three to four thousand man-hours more in an approximately six-month project; the developers felt the increase was at least partly due to the test-driven practices. George and Williams [15] and Canfora et al. [13] came to the same conclusion in their experiments that developers used more time with test-driven development. Effort and time spent on testing has also been shown to increase in academical experiments [25]. However, the correlation between test-driven development and effort isn't that straightforward as Geras et al. [16] didn't notice such a big difference and students have been shown to get a faster start into development with test-driven development [24].

Considering effort, it is not enough to look at the initial stages of development, as more effort is put on the development of the code products in later stages of the software life cycle when the code is being maintained or refactored for other purposes. Here, the industrial case study of Dogša and Batič [14] provides interesting insights into the different stages of development. Even though the test-driven development team had used more time in the development phase before the major release of the product, maintenance work on the code that was previously written with test-driven development was substantially easier and less time-consuming. The observation period for the maintenance period was around nine months, and during this time the test-driven team was quickly making up for the increased effort in the initial development stage, although they were still several thousand man-hours behind the aggregated effort of the non-test-driven teams when the observation ended.

#### 4.7 External Quality

External quality in this review refers to qualitative, subjective and environment specific information from the empirical studies that doesn't easily convert to quantifiable metrics. For instance, mixed-method research approaches have been applied in several studies which have utilized various surveys and interviews to gather subjective opinions about test-driven development.

When customers are shown products which have been developed with test-driven development, they don't necessarily perceive a shift in quality. Huang and Holcombe [25] interviewed customers of a student project with a detailed questionnaire and asked how they felt about the quality of the product after having used it for a month. According to the results, the development method didn't affect the quality that the customers saw: test-driven products got similar ratings as the products of other development methods.

Perhaps developers who participate in the design and implementation of software can better explain what the true effects of test-driven development are? Dogša and Batič [14] investigated this question with a survey for the industry developers who took part in the study and they felt that they were able to provide

better code quality through test-driven development. The students in the study of Gupta and Jalote [24] had the sense that test-driven development improved the testability of their creation but at the same time reduced the confidence in the design of the system.

#### 4.8 Productivity

Productivity is an indirect resource metric based on the relation between the amount of some tangible items produced and the effort required to produce the output. The resources can, for instance, be developers while the output can be products resulting from development work like source code or implemented user stories. Productivity is an external attribute which is sensitive to the environment [31]. Test-driven development seemed to be a factor in the increased effort of the developers as previously described but could the restructured development process and the tests written somehow accelerate the implementation velocity of the user stories or affect the rate by which developers write source code lines?

There have been a number of studies which have featured test-driven development and productivity. Dogša and Batič [14] reported that the industrial test-driven developer team produced code at a slightly lower rate than the other teams involved in the study and the developers also thought themselves that their productivity was affected by the practices required in test-driven development. In the experiment of Madeyski and Szała [21], there were some signs of increased productivity but it was noted that there were certain validity threats for this single-developer study. Student developers who used test-driven development in the study of Gupta and Jalote [24] were on average on the same level or a bit faster in producing source code lines than student teams developing without test-driven development. In the study of Huang and Holcombe [25], students were also faster with test-driven development, although the difference didn't exceed statistical significance. But then again there have been test-driven student teams whose productivity has been lower in terms of implemented features or source code lines [29]. In some cases, no differences between the productivity of student teams have been found [28].

The time it takes to produce one line of code might depend on the type of code being written as well. Müller and Höfer [22] examined the productivity rates when experts and students were developing code in a test-driven development experiment and noticed that both experts and students wrote test code faster than they wrote the implementation code. Experts were generally faster in writing code than students but both groups of developers wrote test code three times faster reaching maximum rates of 150 lines of code per hour. Test-driven development involves writing a lot of test code but based on this result, writing an equal amount of test code doesn't take as long as writing implementation code which is something to consider.

## 4.9 Maintainability

Maintainability is a property that is related to some of the evolution aspects of software: the easiness of finding out what to change from existing code, the relative effort to make a change and the sustained confidence that everything works well after the change with sufficient mechanisms to verify the effects [35]. An array of automated tests might at least help to increase the testability and stability of software which implies that test-driven has a chance to affect maintainability.

Few empirical studies about test-driven development mention maintainability and there seems to be room for additional research in this area. The industrial case study of Dogša and Batič [14] considers maintainability and the nine months maintenance period seems long enough to draw some initial conclusions. As previously described, serving the change requests for the code that had been developed with test-driven development took less time and was thus more effortless. In addition, when interviewed, developers in the study answered to a closed question that the development practice helped them to make the software more maintainable. While more research could verify whether the effect is of a constant nature, the idea is still encouraging.

## 4.10 Aggregation of Results

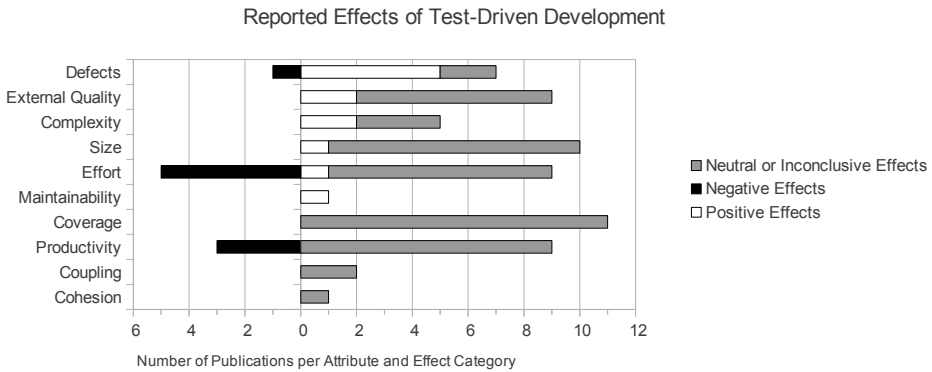
A summary of the reported effects and the frequency of individual quality attributes in research reports is shown in Figure 1. In the figure, the attributes are ordered by the number of publications showing significant positive effects associated to a particular attribute. Besides showing the positive effects, the figure illustrates the number of publications showing neutral or inconclusive effects and significant negative effects as reported in the respective publications. It should be considered that this is a coarse-grained aggregation of the study results that ignores the different contexts. A summary tailored for a specific context might look different. However, the study gives an overview of the trends and can be seen as a starting point for deeper analysis and interpretation.

Many of the significant positive effects are associated with defects but there are some positive effects related to external quality, complexity, maintainability and size. The negative effects are mostly related to effort and productivity although in one study test-driven development was seen to reduce effort. There are also several studies showing no effect with respect to effort. In case of effort and productivity, this might indicate that there are hidden context factors that have an influence on the effect of test-driven development. Code coverage is a common attribute that is mentioned in the studies but it is rarely highlighted as a key dependent variable in the studies or the results have been found inconclusive; the same can be said for size.

## 4.11 Limitations

The results of the individual studies have a limited scope of validity and cannot be easily generalized and compared. Therefore, the findings presented in this





**Fig. 1.** The occurrence of positive, neutral and negative effects for each quality attribute as reported by the test-driven development publications included in the review

article need a careful analysis of the respective contexts before applying in other environments. The completeness of the integrative literature review was based on the ranking algorithm of the search engines and might have been enforced more strictly. Other threats to validity concern the use of qualitative inclusion and exclusion criteria as well as the selection of databases, search terms, and the chosen timeframe. Due to these factors, there could be a selection bias related to the selection of the publications. This needs to be taken into care when interpreting and using the results of this integrative literature review.

## 5 Conclusion

This integrative literature review analyzed the effects of test-driven development from existing empirical studies. The detailed review collected empirical findings for different quality attributes and found out varying effects to these attributes. Based on the results, prominent effects include the reduction of defects and the increased maintainability of code. The internal quality of code in terms of coupling and cohesion seem not to be affected so much but code complexity might be reduced a little with test-driven development. With all the tests written, the whole code base becomes larger but more source code lines are being covered by tests. Test code is faster to write than the code implementing the test but many of the studies report increased effort in development.

The quality map constructed as part of the review shows some possible directions for future research. One of the promising effects was the increased maintainability and reduced effort it took to maintain code later but at the time of the review there was only a single study from Dogša and Batič [14] which had specifically focused on maintainability. This could be one of the areas for further research on test-driven development.

## References

1. Beck, K.: Test-Driven Development: By Example. Addison-Wesley (2003)
2. Turhan, B., Layman, L., Diep, M., Erdogmus, H., Shull, F.: How Effective is Test-Driven Development. In: Oram, A., Wilson, G. (eds.) *Making Software: What Really Works, and Why We Believe It*, pp. 207–219. O'Reilly (2010)
3. Jeffries, R., Melnik, G.: Guest Editors' Introduction: TDD—The Art of Fearless Programming. *IEEE Software* 24(3), 24–30 (2007)
4. Rafique, Y., Mistic, V.: The Effects of Test-Driven Development on External Quality and Productivity: A Meta Analysis. *IEEE Transactions on Software Engineering* 39(6), 835–856 (2013)
5. Desai, C., Janzen, D., Savage, K.: A Survey of Evidence for Test-Driven Development in Academia. *SIGCSE Bulletin* 40(2), 97–101 (2008)
6. Runeson, P., Höst, M.: Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering* 14, 131–164 (2009)
7. Merriam, S.B.: *Qualitative Research: A Guide to Design and Implementation*. John Wiley & Sons (2009)
8. Torraco, R.J.: Writing Integrative Literature Reviews: Guidelines and Examples. *Human Resource Development Review* 4(3), 356–367 (2005)
9. Kitchenham, B., Charters, S.: *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Technical Report Version 2.3, Keele University and University of Durham (July 2007)
10. Creswell, J.W.: *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Sage Publications (2009)
11. Mäkinen, S.: *Driving Software Quality and Structuring Work Through Test-Driven Development*. Master's thesis, Department of Computer Science, University of Helsinki (October 2012)
12. Bhat, T., Nagappan, N.: Evaluating the Efficacy of Test-driven Development: Industrial Case Studies. In: *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, ISESE 2006*, pp. 356–363. ACM, New York (2006)
13. Canfora, G., Cimitile, A., Garcia, F., Piattini, M., Visaggio, C.A.: Evaluating Advantages of Test Driven Development: A Controlled Experiment with Professionals. In: *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, ISESE 2006*, pp. 364–371. ACM, New York (2006)
14. Dogša, T., Batič, D.: The Effectiveness of Test-Driven Development: An Industrial Case Study. *Software Quality Journal* 19, 643–661 (2011)
15. George, B., Williams, L.: An Initial Investigation of Test Driven Development in Industry. In: *Proceedings of the 2003 ACM Symposium on Applied Computing, SAC 2003*, pp. 1135–1139. ACM, New York (2003)
16. Geras, A., Smith, M., Miller, J.: A Prototype Empirical Evaluation of Test Driven Development. In: *Proceedings of the 10th International Symposium on Software Metrics, METRICS 2004*, pp. 405–416 (September 2004)
17. Maximilien, E., Williams, L.: Assessing Test-Driven Development at IBM. In: *Proceedings of the 25th International Conference on Software Engineering, ICSE 2003*, pp. 564–569 (May 2003)
18. Nagappan, N., Maximilien, E., Bhat, T., Williams, L.: Realizing Quality Improvement through Test Driven Development: Results and Experiences of Four Industrial Teams. *Empirical Software Engineering* 13, 289–302 (2008)

19. Williams, L., Maximilien, E.M., Vouk, M.: Test-Driven Development as a Defect-Reduction Practice. In: Proceedings of the 14th International Symposium on Software Reliability Engineering, ISSRE 2003, pp. 34–45 (November 2003)
20. Janzen, D.S., Saiedian, H.: Does Test-Driven Development Really Improve Software Design Quality? *IEEE Software* 25(2), 77–84 (2008)
21. Madeyski, L., Szala, Ł.: The Impact of Test-Driven Development on Software Development Productivity — An Empirical Study. In: Abrahamsson, P., Baddoo, N., Margaria, T., Messnarz, R. (eds.) *EuroSPI 2007*. LNCS, vol. 4764, pp. 200–211. Springer, Heidelberg (2007)
22. Müller, M., Höfer, A.: The Effect of Experience on the Test-Driven Development Process. *Empirical Software Engineering* 12(6), 593–615 (2007)
23. Desai, C., Janzen, D.S., Clements, J.: Implications of Integrating Test-Driven Development Into CS1/CS2 Curricula. In: Proceedings of the 40th ACM Technical Symposium on Computer Science Education, SIGCSE 2009, pp. 148–152. ACM, New York (2009)
24. Gupta, A., Jalote, P.: An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development. In: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, ESEM 2007, pp. 285–294 (September 2007)
25. Huang, L., Holcombe, M.: Empirical Investigation Towards the Effectiveness of Test First Programming. *Information and Software Technology* 51(1), 182–194 (2009)
26. Janzen, D., Saiedian, H.: On the Influence of Test-Driven Development on Software Design. In: Proceedings of the 19th Conference on Software Engineering Education and Training, CSEET 2006, pp. 141–148 (April 2006)
27. Madeyski, L.: The Impact of Test-First Programming on Branch Coverage and Mutation Score Indicator of Unit Tests: An Experiment. *Information and Software Technology* 52(2), 169–184 (2010)
28. Pančur, M., Ciglarič, M.: Impact of Test-Driven Development on Productivity, Code and Tests: A Controlled Experiment. *Information and Software Technology* 53(6), 557–573 (2011)
29. Vu, J., Frojd, N., Shenkel-Therolf, C., Janzen, D.: Evaluating Test-Driven Development in an Industry-Sponsored Capstone Project. In: Proceedings of the Sixth International Conference on Information Technology, ITNG 2009, pp. 229–234. New Generations (April 2009)
30. Wilkerson, J., Nunamaker, J.J., Mercer, R.: Comparing the Defect Reduction Benefits of Code Inspection and Test-Driven Development. *IEEE Transactions on Software Engineering* 38(3), 547–560 (2012)
31. Fenton, N.E., Pfleger, S.L.: *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company, Boston (1997)
32. Pezzè, M., Young, M.: *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, Chichester (2008)
33. McCabe, T.: A Complexity Measure. *IEEE Transactions on Software Engineering* SE 2(4), 308–320 (1976)
34. Chidamber, S., Kemerer, C.: A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20(6), 476–493 (1994)
35. Cook, S., He, J., Harrison, R.: Dynamic and Static Views of Software Evolution. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 592–601 (2001)

# Isolated Testing of Software Components in Distributed Software Systems

François Thillen, Richard Mordinyi, and Stefan Biffi

Christian Doppler Laboratory “Software Engineering Integration for Flexible  
Automation Systems”

Vienna University of Technology  
Vienna, Austria

{firstname.lastname}@tuwien.ac.at  
<http://cdl.ifs.tuwien.ac.at>

**Abstract.** Component-based software engineering emphasizes the composition of software systems through loosely coupled independent components. Although software components are binary units of independent software artifacts, they typically interact with other components as they form a functioning system and thus implicitly define dependency relations. However, in case of distributed component-based software systems current testing strategies either assume total independence of components or require usage of mock-up frameworks which do not facilitate testing of the entire component but only a subset of it. Consequently, the dependency structure between components is not really taken into account. Therefore, those tests are limited in their effectiveness of detecting defects in software systems with distributed components. In this paper the “Effective Tester in the Middle” (ETM) approach is presented, which improves testing of components depending on other distributed components. The approach relies on test scenario specific interaction models and network communication models which facilitate isolated testing of entire components without the need to run the overall system. We evaluate the approach by implementing test scenarios for a system integration platform. The prototypic implementation demonstrates that software testers are able to create unit test like integration tests with minimal effort and that it increases the quality of the system by enabling the injection of fault-messages.

**Keywords:** Component-based Systems, Distributed Components, Component Tests.

## 1 Introduction

Component-based software engineering (CBSE) [1] relies on the existence of independent software components which can be composed to a software system in a loosely coupled manner. Instead of continually custom developing software artefacts, as is largely the case under traditional development processes, CBSE concentrates on assembling prefabricated parts which can be an organization’s

own implementation or a part from professional component vendors [2]. The concept of reusing pre-existing components aims to reduce development time, costs and risks, while developing larger and more complicated systems quickly and with high product quality [3]. Nevertheless, as with any software engineering approach these goals rely on thoroughly tested components to ensure their quality and stability, and thus consequently of the system.

There are several testing approaches and frameworks at software testers' disposal (see section 2.2) to find defects in the code and to inspect the correct behaviour of a software component [4] in distributed software systems. However, while those testing approaches rely on the definition of an independent component [5], software components typically interact with other components as they form a functioning system and thus implicitly define dependency relations [6]. If software testers want to ensure the quality of dependent components by testing the components' interface, they can use of mock up frameworks [7]. However, these do not support testing of the entire component, but only a subset of it (since the rest of the component is mocked) and thus does not take into account any component dependencies. On the other hand integration tests using the component's interface, but it can be performed only if the entire system is up and running. In that case the entire component is tested and its dependencies considered. However, while this is an easy task in a non-distributed environment, it might be a challenge with high effort in a distributed one.

In this paper, the so called "Effective Tester in the Middle" (ETM) approach is presented, which improves testing of distributed components depending on other components in the system by introducing interaction models and network communication models which facilitate isolated testing of the entire component without the need to run the entire system. The proposed approach makes use of established network protocols to filter for requests sent by tested components. The request is analysed and based on a given test scenario and its specific interaction model an appropriate response is returned. Software testers implement test scenarios in unit testing manner and configure request-response interaction models used by the ETM while monitoring network communication. The proposed approach is evaluated by implementing test scenarios for a system integration platform. The evaluation results show that although software testers are able to create unit test like integration tests with minimal effort, a high one-off effort has to be invested into the implementation of network communication models for filtering purposes. Furthermore, results indicate increased effectiveness in error detection, since it may detect errors which cannot be easily detected with current testing frameworks and by enabling the injection of faulty request or response messages.

## 2 Related Work

In this section we summarize related work on dependencies in component based software systems (CBS), on testing approaches and frameworks, and mock-up concepts for such systems to review scientific and industrial landscape.

## 2.1 Dependencies in Component Based Systems

A challenge in CBS [5] is to avoid dependencies between components [8] since according to the definition components should be independent. However, components still need other/external components to fulfil their work [8], [9], and thus to form a running system. These interactions lead to non-technical dependencies [6] between components. Furthermore, components can also have internal dependencies, which means that they not only depend on self-generated elements but also have relations between input and output [9]. The greater the number of dependencies between components, the more complex the system will become [6]. This makes the system harder to modify, verify, and understand and thus leads to poor maintainability. Therefore, it is not only important to know these dependencies in order to verify and being capable of modifying components but also to define the influence of external dependencies on the component's behaviour [9].

## 2.2 Testing Concepts for Component Based Systems

**Web Services:** Web services are integration technologies, which enable a dynamic correlation between components in networks under the use of open standardized internet technologies, like the "Web Services Description Language" (WSDL) for describing the interfaces, or the "Simple Object Access Protocol" (SOAP), the "Hypertext Transfer Protocol" (HTTP), and the "Internet Protocol" (IP) for communication purposes. [10] presents two tools that are using Web services as interfaces between services and client components. These tools offers an easy way for testing Web services in different programming languages. However, Web services are independent and have always a defined endpoint.

**Distributed Components:** One of many testing tool is Jata [11] that is a language for testing distributed components. Jata is mainly integrates the benefits of JUnit and TTCB-3 [12], supports Message orientated middleware and web services. Another more theoretical approach is the decomposition and hybrid approach [13], which introduces a formal way to analyse the correct behaviour of a component. Furthermore, it combines a model-checking techniques and black box testing that is the hydride approach. JRT [3] is a wrapping java component (JRT-Server), which automatically executes test cases (JRTClient) and offers a graphical representation (JRTClientGUI).

**Component Based Systems:** A way of testing component based system is an automated statistical test approach [14]. The approach is based on state-based component models, which are used to create compact interaction test models. The goal is to test system interactions and functionalities that are split in several systems. A general guideline of improving testing in component based systems is presented by Toroi [15]. The goal was to enhance testing methods and the practicality, especially from the integrator view [15].

These testing approaches rely on the original definition, i.e. independent components. When components are dependent and communicating with other components over several ports, then the presented approaches cannot be applied as components need to be independent.

## 2.3 Mock-up Frameworks

Complex software systems may have several dependencies to external systems like JMS middleware or application servers - "All these moving parts can be difficult to manage and provide interacts that are outside the scope of a unit test" [7]. However, to test the interaction in the system, dependencies to other components are not really needed and can be mocked. There are several frameworks like mockito<sup>1</sup>, jMock<sup>2</sup>, EasyMock<sup>3</sup>, or NMockEasyMock<sup>4</sup>, which provide ways to mock-up method calls. These frameworks create mock-ups by implementing an own class (mocked class) from the selected class. The behaviour is modelled over keywords, which allows testing and verification of the correct behaviour of the system. This implies that the behaviour of the component has to be known beforehand. However, the method, which is mocked-up, has to be tested in another test scenario because only the behaviour of it is simulated but the method itself is never executed. This implies that bugs can be hidden behind the mock-up. Nevertheless, this approach is based on testing the interactions of the system and the correct behaviour of methods, which implies that all components, which have dependencies to other external components are mocked-up in the test scenario. In other words, in order to test the integration of the system, the methods/components have to be independent to other components or network access.

## 3 Motivating Scenario

Component-based software engineering aims reusing and assembling software systems of software parts which potentially have been prefabricated by third-parties. A lot of testing approaches have been introduced and presented, which help to find software defects (see section 2).

An example for a component-based software system is the so called Engineering Service Bus (EngSB) [16]. In comparison to the Enterprise Service Bus, which integrates services [17], the EngSB integrates not only different tools and systems but also different steps in the software development life cycle [16] - the platform aims at integrating software engineering disciplines. A motivating set of components is the EngSB, the .Net Bridge [18], and the Engineering Object Editor (EOE) [19]. The .Net Bridge is a communication interface which enables interaction between the EngSB and .Net-based implementations (e.g., EOE),

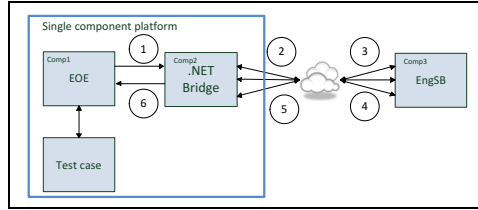
---

<sup>1</sup> <http://code.google.com/p/mockito/>

<sup>2</sup> <http://www.jmock.org/>

<sup>3</sup> <http://www.easymock.org/>

<sup>4</sup> <http://www.nmock.org/>



**Fig. 1.** Communication between EOE, .Net Bridge and the EngSB

by providing means to invoke methods on the EngSB. The EOE is a concrete application that uses the .Net Bridge to communicate with the EngSB.

Figure 1 illustrates the simplified structure of the system, the dependency relations between the components, the flow of information exchange between the components, and the problem space of testing in that environment. The EOE is an Excel-plugin which visualizes data stored in the EngSB. Therefore, whenever the EOE retrieves (Figure 1, 1) data from the EngSB, it forwards the request to the .Net Bridge and then blocks until the data arrives. The .Net Bridge marshals the method request to a JSON object and forwards it to the EngSB via Message-oriented Middleware (MoM) (Figure 1, 2) like ActiveMQ (AQ)<sup>5</sup>. The EngSB receives the request (Figure 1, 3), unmarshalls the JSON object and invokes the corresponding method. The result is marshalled (Figure 1, 4) and transmitted to the .Net Bridge over the MoM. The .Net Bridge unmarshalls the JSON objects (Figure 1, 5) and forwards the result to the EOE by invoking a call back method (Figure 1, 6). Last, the EOE returns from the blocking state and presents the data received. To test this structure, the complete system has to be started, i.e. the EngSB, .Net Bridge, and EOE. However, starting up the entire system for running defined test scenarios costs effort and time.

## 4 Research Issues

In this paper, the so called "Effective Tester in the Middle" (ETM) is introduced, which aims to improve testing of distributed components depending on other components in the system. ETM introduces interaction models and network communication models which facilitate isolated testing of entire components without the need to run the entire system. The key research issue is how to test dependent components in distributed environments in order to find defects effectively and efficiently:

**Modelling Component Dependencies:** Remote dependent components need data from other components to fulfil their work. Therefore, the interaction requires network protocols (e.g., TCP, UDP) to be used, consequently defining a technical dependency. Another dependency refers to the used application protocol, which allows applications to filter information in an efficient way. Therefore,

<sup>5</sup> <http://activemq.apache.org/>



how can dependencies between components be modelled, so that system properties are reflected correctly?

**Effectiveness of the ETM Approach:** An approach to test remote components is to start the entire system and then execute test cases. However, during the software development process it is unlikely that all components are available making it difficult to start the complete system and create correct test cases. The ETM aims to resolve dependencies between components and to isolate them. Therefore, to what extent is the ETM capable of improving effectiveness of component testing?

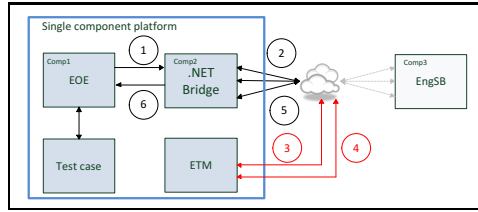
**Efficiency of the ETM Approach:** Components communicate with each other by exchanging messages. How can the ETM improve efficiency of testing components in distributed systems based on the messages exchanged? What kind of additional testing approaches can be easily facilitated and executed in comparison to traditional testing approaches?

## 5 Proposed Solution Approach

The "Effective Tester in the Middle" (ETM) approach relies on test scenario specific interaction models and network communication models. In the following the approach is described in detail, which consists of three main parts:

The ETM core takes care of the communication and handles messages. Since components communicate over a network connection with each other, ETM listens on the Transport Layer for messages. In case of TCP connections, ETM opens a socket first, which listens on the specified port and IP-address. Whenever the socket receives messages, ETM parses through all the configured and specified protocol types and asks if all data has been arrived. If there are still missing messages, the socket combines the old received data with the new ones and parses again through all protocols. In case the protocol replies with all data has been received, the ETM parses through the configuration and forwards the data to each configuration. The configuration analyses the data and checks if the message is complete. Then, the ETM uses the response Transaction model to open a socket (or an existing one) and forward the reply to the client.

The application protocol model references the protocol that the component uses, like SOAP or ActiveMQ. These protocol implementations serialize byte messages to protocol messages and deserialize the message from the received protocol message to a desired type. Every protocol has characteristics, which shows that it is valid. SOAP for example needs an HTML part that includes for example the complete message size. Such characteristics can be used to identify if messages in a byte format are valid. The interaction model represents the message transfer between the components, and needs the following parameters: 1) Port and IP-address from which a client sends messages; 2) A message in a protocol format, which stands for the request message; 3) A list of interaction model referencing a reply; 4) Port and IP-address on which the ETM himself should open to receive requests from a client. These models are used by the ETM to correctly reply to exchanged messages.



**Fig. 2.** ETM in the presented environment

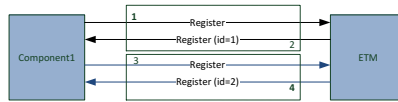
Figure 2 shows the integration of the ETM in the use case presented in section 3. In contrast to the traditional system, communication to the EngSB is simulated by the ETM. The EOE does not know that the EngSB is simulated. This means that check out and check in are still performed the same way (Figure 2, 1 and 6). Also the .Net Bridge does not know that the ETM is the responder (Figure 2, 2 and 5). The ETM catches the communication messages (Figure 2, 3), parses the request, and replies with a corresponding message to the .Net Bridge (Figure 2, 4), which then forwards it to the EOE.

## 6 Prototypic Implementation

The following section describes first the ETM core followed by two examples. In the first example, SOAP messages, based on HTML and XML are sent over TCP, which demonstrate the power of the ETM in a realistic environment. In the second example the ETM simulates the EngSB (Figure 2) that allows it to test the .Net Bridge and the EOE isolated. In this case the ETM simulates the AQ protocol. The sockets have to be configured on a specified IP and Port, which will be provided by the test case and for simplicity reasons, the communication model in this paper is TCP. First the ETM Core is described, which include a new list implementation and the socket handling.

### 6.1 ETM Core, ETM List, and Socket Handling

The ETM core provides the possibility to accept communications, send and receive messages. Mostly, the ETM is reacting on received message but components can wait for method calls from outside. The ETM has all the open socket saved and so allows it to send messages to a specific socket. Like presented in the previous session the interaction model consists these information. When the ETM core receives a message, it forwards the bytes to the list and searches for the corresponding configuration and send the result as answer. Communication between the components works mostly over the network connection and very often components are sending the same message several times. It follows that the answer for the first message should be the first interaction model and the second answer should be the second. This behaviour is presented in Figure 3. The first message from the component1 (Figure 3, 1) is a registration message,



**Fig. 3.** Message Request Order

which is caught by the ETM and a message from the configuration (e.g identifier) is send back (Figure 3, 2). Next, the component1 sends a second registration message (Figure 3, 3), which requires an answer. Corresponding to the example, this should have a different identifier as the previous (Figure 3, 4). This problem requires a new list implementation. From the concrete implementation point of view, every list entry has a counter, which is initialised with zero. When a configuration gets picked this counter get increased (Figure 3, 2 and 4). To find the correct configuration, first the protocol (converted from bytes) is compared with the interaction models and if there more matches, the number of returns (counter) is compared. Most programming languages are handling all the layers up to the transport layer, thus only the socket has to be configured by an endpoint (IP address and port) and the used transport layer (e.g. TCP). The components can communicate over different ports with each other, which make it the simulation challenging. The ETM opens for each port an own thread that handles the communication with the component on that specific port.

## 6.2 Test Case Examples

The first example describes the behaviour of the ETM with SOAP messages. When a socket receives a message, first the ETM tries to identify the used protocol. Second, the bytes are forwarded to the identified SOAP protocol (in this case) and analysed if all required parts are present (e.g. closed Envelope (</Envelope>)). If the bytes are valid, the ETM looks at the SOAP specific interaction configuration (provided by the test case) and search for a response message. Next, the ETM adds to the chosen messages the SOAP protocol information, like header information, and transfers it to the requester. The test case consists of three different parts. First, the configuration of the ETM will be created, followed by the Web service client initialization, while in the third part the method `getData` of the Web Service is invoked. The result of this method call should be 412571. The `ConfigureStartETMWithSOAPProtocol` method configures and starts the ETM (described in Figure 4). In the first step, the request and reply messages are defined, which are used to generate the corresponding SOAP messages. This is done by the implementation of the application protocol. Next, a list of answers respectively replies have to be integrated into the interaction message, which represents a configuration for the ETM. The interaction message contains the IP-Address and Port, on which the ETM should listen or send the message. These interaction messages are then forwarded to the ETM while finally the ETM is started. The request and reply messages generated for this example look like as in Figure 6 and Figure 7. The `GetRequestTestCase1` method

```

private void ConfigureStartETMWithSOAPProtocol()
{
    String requestMsg = GetRequestTestCase1();
    String replyMsg = GetReplyTestCase1();

    IProtocol requestMessage = new SOAPProtocol(requestMsg);
    IProtocol answerMessage = new SOAPProtocol(replyMsg);

    TransactionMessage reply = new TransactionMessage(1866, 0,
        answerMessage, null, IPAddress.Loopback, IPAddress.Loopback);
    TransactionMessage request = new TransactionMessage(51446, 1866,
        requestMessage, reply, IPAddress.Loopback, IPAddress.Loopback);

    ETM = new ETMTCF(new List<TransactionMessage>() { request });
    ETM.Start();
}

```

Fig. 4. ETM configuration

```

public List<InteractionMessage> getConfig(WireFormatInfo wire)
{
    List<InteractionMessage> result =
        new List<InteractionMessage>();
    result.Add(ActiveMQConf.getRemoveInfoAnswer(-1));
    result.Add(ActiveMQConf.getShutdownInfoAnswer(-1));
    result.Add(ActiveMQConf.getKeepAliveAnswer(-1));
    result.Add(ActiveMQConf.getWireFormatAnswer(wire, -1));
    result.Add(ActiveMQConf.getNetBridgeTextMessageAnswer(-1));
    result.Add(ActiveMQConf.getAskedAnswer(-1));
    result.Add(ActiveMQConf.getSessionInfoAnswer(-1));
    result.Add(ActiveMQConf.getProducerInfoAnswer(-1));
    result.Add(ActiveMQConf.getConnectionInfoAnswer(-1));
    result.Add(ActiveMQConf.getConsumerInfoAnswer(-1));
    return result;
}

```

Fig. 5. ActiveMQ configuration

returns the message, which will be invoked by the Web Service client (see Figure 6). In the `GetReplyTestCase1` method the response message is generated, which should be transmitted to the client (see Figure 7). In the last row, the HTML header informations are added, which are generated from the XML part. The third example is based on the presented use case. Not only the component itself has to simulate but also the behaviour of the application protocol (AQ). The AQ protocol is based on commands, which are serialized to bytes forwarded to the transport layer and on the other side deserialised. AQ uses the OpenWire format to communicate with each other. The ETM needs a AQ protocol implementation to be able to communicate with components that uses this MOM. However, AQ provides a class (`OpenWireFormat`) that offers the possibility to serialize and deserialize objects to/from a byte array. To create a valid connection, every response command needs a corresponding `CommandId`, every `MessageDispatcher` a `ConsumerId`, and a `Destination`. The test case architecture is equivalent to the previous example and is presented in Figure 8. First, (Figure 8, 1) the ETM gets configured and started. The configuration creates the interaction models that are used to interact with the AQ client (.Net Bridge). The interaction models are presented in Figure 5, which are related to Figure 10. Every dotted box stands for a communication commands of AQ. For every message that is sent/received from the .Net Bridge, a new producer/consumer is opened. It follows that the commands are the same for all these sockets. In the code, this is specified by setting the socket number -1. The answers for the .Net Bridge has to be configured, which are shown as boxes (solid lines) in Figure 10. The first message from the .Net Bridge is send on the Socket 2, which is a create message. According to the definition of a connector, a void message is needed, i.e. the ETM sends a void message back. This void message is wrapped in an `ActiveMQTextMessage` command, which itself is wrapped in a `MessageDispatcher`. This behaviour is the same for the register, unregister, and delete message. Next, the .Net Bridge has

```

IGNOREFIELD
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <getData xmlns="http://tempuri.org/">
    </s:Body>
  </s:Envelope>

```

Fig. 6. SOAP request message

```

<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <getDataResponse xmlns="http://tempuri.org/">
      <getDataResult>412571</getDataResult>
    </getDataResponse>
  </s:Body>
</s:Envelope>

```

Fig. 7. SOAP response message

to be started (Figure 8, 2). When the registration method was successfully, some method calls on the bridge can be triggered to test the correct behaviour. This is done by forwarding a configuration to the ETM, which finds the correct socket and forwards the request to the .Net Bridge. This request has to be send to the receive queue that is from the definition of the .Net Bridge always on socket 0. To be able to send a valid request to the correct consumer, the ConsumerId and Destination of the receive queue is required. This information is stored in the ETM itself. Next we have to close the .Net Bridge in a correct way that implies to send unregister and delete message. This behaviour is already configured (Figure 9). In the last step (Figure 8, 3) the correct behaviour is tested with normal unit tests. The method call, which has been triggered from the ETM, some variables had been set. These values get test on their correctness with the normal test strategies.

```

public void TestBridge()
{
    ETM = new ETMImplementation(getETMConf());
    ETM.Start(IPAddress.Loopback, 6549); 1
    startBridge();
    ETM.TriggerMessage(ActiveMQConf.
    getNetBridgeInvokeMsgOnReceiveQueue(0,
    getTestCase(), ETM.ReceivedMessages)); 2
    stopBridge();
    AreEqual(domain.message, "TestCase1");
    AreEqual(domain.level, "12");
    AreEqual(domain.name, "Test");
    AreEqual(domain.origin, "123"); 3
   .IsTrue(domain.processId == 123);
    AreEqual(domain.processIdSpecified, true);
}

```

**Fig. 8.** ActiveMQ and .Net Bridge communication

```

private List<InteractionMessage> getETMConf(){
    List<InteractionMessage> result = ActiveMQConfig.getConf();
    result.Add(ActiveMQConfig.
    ConsumerVoidMsg(2, BridgeVoidAnswer()));
    result.Add(ActiveMQConfig.
    ConsumerVoidMsg(4, BrideVoidAnswer()));
    result.Add(ActiveMQConfig.
    ConsumerVoidMsg(6, BrideVoidAnswer()));
    result.Add(ActiveMQConfig.
    ConsumerVoidMsg(7, BrideVoidAnswer()));
    result.Add(ActiveMQConfig.
    ConsumerVoidMsg(9, BrideVoidAnswer()));
    return result;}

```

**Fig. 9.** ActiveMQ ETM configuration method

## 7 Evaluation

The advantage of the ETM is that test cases can be executed at any time and are not dependent to a finished implementation of a component. Furthermore, specified components can be tested without the need to start up the complete system. Therefore, test cases can be created once and then executed any time without any requirements to the system.

### 7.1 ETM and Modeling Component Dependencies

Mocking and the ETM have several things in common - both consists of three parts: configuration, definition of the behaviour, and initiating testing. Like in mocking frameworks, the definition of the behaviour can be very complex and all the aspects have to be considered. The presented interaction models allow testers to define correct behaviour for a specific message at a specific moment. Furthermore, the model allows to define several reply definitions to different endpoints as a reaction to a received message. This allows to model dependencies between the components and thus to reflect the system properties.

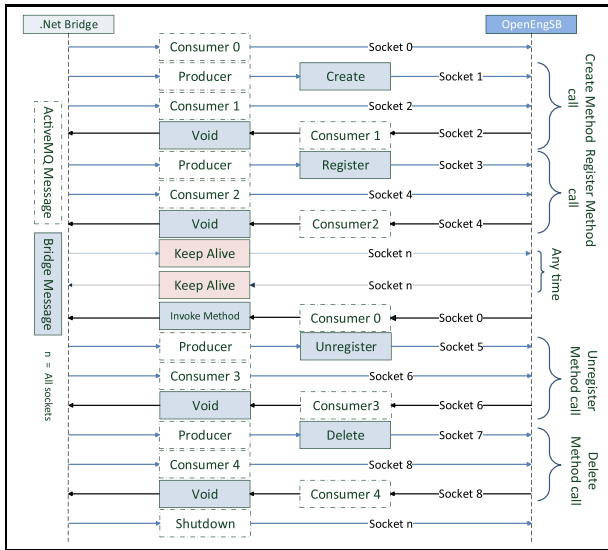


Fig. 10. ActiveMQ Test case example

## 7.2 Effectiveness

The effectiveness is measured, by the different kinds of bugs the ETM can help to find. An advantage of the ETM is that it supports the generation of user defined messages, which may be inconsistent to the definition of the component. ETM therefore enables testing components with fault messages. Since ETM does not execute test cases itself but only simulates other components, the effectiveness depends on the effectiveness of the created test cases. However, this also implies that the ETM can simulate components, which do not exist yet. From the view of the software development process, developers can work independently to the system, which improves the effectiveness of the complete team.

In the following, effectiveness is also discussed by the time of implementing tests and the time needed for execution. The SOAP and ActiveMQ protocols are compared, and together with the complexity and implementation time of the test case the effectiveness of the ETM is illustrated. It needs to be mentioned that the test case using the SOAP protocol is very short, mainly because the component is started in one line only. The complete test case consists of 64 LOC while it took 20 minutes of implementation time. In case of the ActiveMQ protocol reuse of ActiveMQ libraries reduced implementation time. While the protocol code has 82 LOC and took twenty minutes to implement, the behaviour has to be implemented, which contains of 151 LOC and needed about 40 minutes effort. The implementation and creation of the protocol and/or the behaviour of the component costs time. It can be derived that this is just a one time effort. The AQ for example needs in total 20 minutes of implementing the protocol, behaviour and the test case. In contrast, the test case just needs only 15 minutes and 20 LOC. It follows

that all other test cases can use the configuration and the protocol. This improves the effectiveness extremely and it follows that implementing further test cases is very easy. However, the developer of the protocol has to understand the behaviour and the complexity of the protocol to properly handle the exchange of messages.

### 7.3 Efficiency

With the traditional approach, some test cases can be created at any time of the process but the tests fails until all the required components are completely created, which leads to a very late finding of the errors. The ETM is simulating the components and so can also simulate things, which are not implemented yet. With the traditional approach, the components cannot be isolated tested, i.e. the dependencies have to be present and running, which implies that the dependent components have to be error free. This is not guaranteed and so the errors can ours in the component under test and in the dependent components. It follows that the location of the error is challenging to locate. The feature of the tradition approach is that the tests are always communicating with the newest versions of the dependent component. Furthermore, the communication between the components exists and so no simulation has to be created. From the view of performance, the tradition approach implies a start-up time. In the case in which all the dependent components are present, these have to be started, configured and ready before the test case starts. In contrast, the ETM have to be started and next the test cases can be executed. Because the tradition approach does not need an implementation of the behaviour and the protocol, there is no search for a corresponding message. This implies that the communication between the components is faster because the ETM needs some time for converting the message to the chosen protocol and searching the corresponding message. The execution time for the ETM requies 7017 ms without start-up and 7051 ms with start-up and clean-up. The traditional approach (i.e starting all depending components) needs 1516 ms fwithout start-up and 45613 ms with start-up and clean-up. The ETM and the traditional approach are executed with the presented Use case. The start-up for the traditional approach is the following: Starting the OpenEngSB, provide the corresponding domain and execute the test case. This follows that the traditional approach needs a very long time to execute a single test, which are for the use case around 46 seconds. In contrast, the ETM needs for executing a test case 7 seconds, which is very fast compared to the start-up and execution of the tradition approach. A disadvantage is that the ETM needs still 7 seconds without a configuration. The traditional approach is five times faster than the ETM approach. By including the time to implement the protocol and behaviour of a used protocol the ETM needs 93 executions of test cases (Figure 11) to be concurrent to the traditional approach.

The Figure 11 is based on the execution time for the test cases for the presented Use case. Generally, several test cases are create and executed all at the same time. This implies that the peak of 93 is very fast reached and so the ETM justify the implementation time of the protocol and the behaviour.

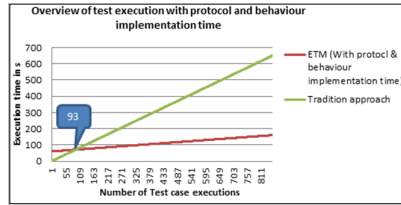


Fig. 11. Execution time dependent to the Numbers of execution

## 8 Discussion

In this section a discussion about the advantages and limitations of the ETM through a comparison to existing concepts is presented. The ETM approach is language independent because it is listening on the transport layer. The transport layer is standardised and so it is offered by a lot of programming language. Furthermore every operation system supports this layer, which implies that components can be tested in different environments, operation systems and languages. Like introduced in section 2, testing is a part of the software engineering process. For every development state a test scenario has to be created.

Like presented in the related work, concepts and approaches already exists to test component based system. Mock-up frameworks, simulates components and allows it to perform tests with the normal test strategies (for example unit tests). This approach is very applicable to every component unless the source code is open. In the other case, it is not possible to use mock-up frameworks because the inner structure has to be known (like white box testing). It is challenging, to use the mock-up framework with the presented approach because the AQ has to be mocked as well. Furthermore, every send and receive method has to be mocked in a way that every message is correctly represented. The test case for the ETM and the Mock-up are the same and both have to simulate the behaviour of the component. It follows that the time to implement the behaviour is very similar. The state-based components approach cannot be used for the Use case because not all of the components are state-based. Compared to the hybrid approach, the presented Use case could not be applied because the basic idea of approach is based on component whose design details are not given. However, the complete design is known in the presented use case. Furthermore, black box testing is just possible with the use case when AQ can synchronises itself. This requires that a dependent component offers an AQ connection. Otherwise the component tries to open a connection which implies an exception. The JRT framework, enables testing of remote server components. The concept is using the black box testing strategy and compares the received result with the expectations. The approach is applicable to server components but is not usable for the presented Use case. This is mainly because the components under test invokes methods on other components and JRT can not handle this kind of method calls (JRT would test the EngSB). The JATA framework is a powerful framework to test components.



It supports Remote Procedure Calls (RPC) and message-based communication. The main different to the ETM is that it does not simulates the dependent component. In contrast, JATA simulates the component and tests the dependent components. It is challenging for the JATA approach to simulate the Use case because the JATA should be used to test the OpenEngSB and not the .Net Bridge. Furthermore, in some use cases a message has to be triggered and send to the component because the component needs some data to full fill the work. In a nutshell, most of the approaches is used to test the here presented dependent components (EngSB) and the ETM can be used for all test cases.

## 9 Conclusion and Further Work

In modern software development processes component based systems become more and more influence. To form a running system components interact with other components and thus implicitly define dependency relations. However, current testing approaches either rely on the total independence of the component or test only a subset of the component. Therefore, those tests are limited in their effectiveness of detecting defects. In this paper the "Effective Tester in the Middle" (ETM) is presented allows the simulation of dependent remote component - it tests the components isolated from the system. The ETM simulates application protocol and provides means to forward test case dependent messages to the component under test. The benefit is higher test coverage and lower effort for implementation.

Future work includes improvements the ETM Core facilitating tests with time-outs and idle times. The intention is to test the reaction of the component with delayed or missing messages.

**Acknowledgment.** This work has been supported by the Christian Doppler Forschungsgesellschaft and the BMWFJ, Austria.

## References

1. Matevska, J.: Rekonfiguration komponentenbasierter Softwaresysteme zur Laufzeit. Vieweg+Teubner, Wiesbaden (2010)
2. Gross, H.G.: Component-Based Software Testing with UML. Springer, Heidelberg (2005)
3. Yao, Y.: A framework for testing distributed software components. *Electrical and Computer Engineering*, 1566–1569 (May 2005)
4. Winkler, D., Hametner, R., Östreicher, T., Bill, S.: A Framework for Automated Testing of Automation Systems (2010)
5. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. In: *Component Software: Beyond Object-Oriented Programming*, 2nd edn. Addison-Wesley Longman Publishing Co., Inc. (2002)
6. Sharma, A., Grover, P.S., Kumar, R.: Dependency analysis for component-based software systems. *ACM SIGSOFT Software Engineering Notes* 34(4), 1 (2009)

7. Minella, M.T.: Pro Spring Batch. Apress (2011)
8. Vieira, M., Richardson, D.: The role of dependencies in component-based systems evolution. *Proceedings of the International Workshop on Principles of Software Evolution - IWPSE 2002*, 62 (2002)
9. Vieira, M., Richardson, D.: Analyzing Dependencies in Large Component-Based Systems, 241 (September 2002)
10. Hamill, P., Alexander, D., Shasharina, S.: Web Service Validation Enabling Test-Driven Development of Service-Oriented Applications. In: *2009 Congress on Services - I*, pp. 467–470 (July 2009)
11. Wu, J., Yang, L., Luo, X.: Jata: A Language for Distributed Component Testing. In: *2008 15th Asia-Pacific Software Engineering Conference*, vol. 60603039, pp. 145–152 (2008)
12. Grabowski, J., Wiles, A., Willcock, C., Hogrefe, D.: On the Design of the New Testing Language TTCN-3, pp. 161–176 (August 2000)
13. Xie, G.: Decompositional verification of component-based systems - a hybrid approach. In: *Proceedings of the 19th International Conference on Automated Software Engineering*, pp. 414–417. IEEE (2004)
14. Bauer, T., Eschbach, R.: Enabling statistical testing for component-based systems. In: Fährnich, K.-P., Franczyk, B. (eds.) *GI Jahrestagung (2)*. LNI, vol. 176, pp. 357–362. GI (2010)
15. Toroi, T.: *Testing Component-Based Systems Towards Conformance Testing*. PhD thesis, University of Kuopio (2009)
16. Biffi, S., Schatten, A.: A Platform for Service-Oriented Integration of Software Engineering Environments. In: *Proceeding of the 2009 Conference on New Trends in Software Methodologies Tools and Techniques Proceedings of the Eighth SoMeT*, pp. 75–92. IOS Press (2009)
17. Chappell, D.A.: *Enterprise Service Bus. Theory in practice*, vol. 4. O'Reilly Media, Inc. (2004)
18. Thillen, F., Mordinyi, R.: M1-TR2011.1.04-NetBridge. Technical report (2012)
19. Mordinyi, R., Pacha, A., Biffi, S.: Quality Assurance for Data from Low-Tech Participants in Distributed Automation Engineering Environments. In: Mammeri, Z. (ed.) *Proceedings of 16th IEEE International Conference on Emerging Technologies and Factory Automation*, pp. 1–4 (2011)

# Automated Test Generation for Java Generics

Gordon Fraser<sup>1</sup> and Andrea Arcuri<sup>2</sup>

<sup>1</sup> University of Sheffield  
Dep. of Computer Science, Sheffield, UK  
`gordon.fraser@sheffield.ac.uk`  
<sup>2</sup> Simula Research Laboratory  
P.O. Box 134, 1325 Lysaker, Norway  
`arcuri@simula.no`

**Abstract.** Software testing research has resulted in effective white-box test generation techniques that can produce unit test suites achieving high code coverage. However, research prototypes usually only cover subsets of the basic programming language features, thus inhibiting practical use and evaluation. One feature commonly omitted are Java's *generics*, which have been present in the language since 2004. In Java, a generic class has type parameters and can be instantiated for different types; for example, a collection can be parameterized with the type of values it contains. To enable test generation tools to cover generics, two simple changes are required to existing approaches: First, the test generator needs to use Java's extended reflection API to retrieve the little information that remains after *type erasure*. Second, a simple static analysis can identify candidate classes for type parameters of generic classes. The presented techniques are implemented in the EVOSUITE test data generation tool and their feasibility is demonstrated with an example.

**Keywords:** automated test generation, unit testing, random testing, search-based testing, EvoSuite.

## 1 Introduction

To support developers in the tedious task of writing and updating unit test suites, white-box testing techniques analyze program source code and automatically derive test cases targeting different criteria. These unit tests either exercise automated test oracles, for example by revealing unexpected exceptions, or help in satisfying a coverage criterion. A prerequisite for an effective unit test generator is that as many as possible language features of the target programming language are supported, otherwise the quality of the generated tests and the usefulness of the test generation tools will be limited.

A particular feature common to many modern programming languages such as Java are *generics* [12]: Generics make it possible to parameterize classes and methods with types, such that the class can be *instantiated* for different types. A common example are container classes (e.g., list, map, etc.), where generics can be used to specify the type of the values in the container. For example, in Java a

`List<String>` denotes a list in which the individual elements are strings. Based on this type information, any code using such a list will know that parameters to the list and values returned from the list are strings. While convenient for programmers, this feature is a serious obstacle for automated test generation.

In this short paper, we present a simple automated approach to generating unit tests for code using generics by statically determining candidate types for generic type variables. This approach can be applied in *random testing*, or in *search-based testing* when exploring type assignments as part of a search for a test suite that maximizes code coverage. We have implemented the approach in the EVOSUITE test generation tool and demonstrate that it handles cases not covered by other popular test data generation tools. Furthermore, to the best of our knowledge, we are aware of no technique in the literature that targets Java generics.

## 2 Background

In this paper, we address the problem of Java generics in automated test generation. To this purpose, this section presents the necessary background information on generics and test generation, and illustrates why generics are problematic for automated test generation.

### 2.1 Java Generics

In Java, generics parameterize classes, interfaces, and methods with type parameters, such that the same code can be instantiated with different types. This improves code reusability, and it helps finding type errors statically.

A generic class has one or more type parameters, similar to formal parameters of methods. When instantiating a generic class, one specifies concrete values for these type parameters. For example, consider the following simplistic implementation of a stack datastructure:

```
public class Stack<T> {
    private T[] data = new T[100];
    private int pos = 0;

    public T pop() {
        return data[pos--];
    }

    public void push(T value) {
        data[pos++] = value;
    }
}
```

The class `Stack` has one *type parameter*, `T`. Within the definition of class `Stack`, `T` can be used as if it were a concrete type. For example, `data` is defined as an array of type `T`, `pop` returns a value of type `T`, and `push` accepts a parameter of type `T`. When instantiating a `Stack`, a concrete value is assigned to `T`:

```
Stack<String> stringStack = new Stack<String>();
stringStack.push("Foo");
stringStack.push("Bar");
String value = stringStack.pop(); // value == "Bar"

Stack<Integer> intStack = new Stack<Integer>();
intStack.push(0);
Integer intValue = intStack.pop();
```

Thanks to generics, the `Stack` can be instantiated with any type, e.g., `String` and `Integer` in this example. The same generic class is thus reused, and the compiler can statically check whether the values that are passed into a `Stack` and returned from a `Stack` are of the correct type.

It is possible to put constraints on the type parameters. For example, consider the following generic interface:

```
public abstract class Foo<T extends Number> {
    private T value;

    public void set(T value) {
        this.value = value;
    }

    public double get() {
        return value.doubleValue();
    }
}
```

This class can only be instantiated with types that are subclasses of `Number`. Thus, `Foo<Integer>` is a valid instantiation, whereas `Foo<String>` is not. A further way to restrict types is the `super` operator. For example, `Foo<T super Bar>` would restrict type variable `T` to classes convertible to `Bar` or its super-classes.

Note also that `Foo` is an abstract class. When creating a subclass or when instantiating a generic interface, the type parameter can be concretized, or can be assigned a new type parameter of the subclass. For example:

```
public class Bar extends Foo<Integer> {
    // ...
}

public class Zoo<U extends Number, V> extends Foo<U> {
    // ...
}
```

The class `Bar` sets the value of `T` in `Foo` to `Integer`, whereas `Zoo` delays the instantiation of `T` by creating a new type variable `U`. This means that inheritance can be used to strengthen constraints on type variables. Class `Zoo` also demonstrates that a class can have any number of type parameters, in this case two.

Sometimes it is not known statically what the concrete type for a type variable is, and sometimes it is irrelevant. In these cases, the *wildcard type* can be used. For example, if we have different methods returning instances of `Foo` and we do not care what the concrete type is as we are only going to use method `get` which works independently of the concrete type, then we can declare this in Java code as follows:

```
Foo<?> foo = ... // some method returning a Foo
double value = foo.get();
```

Generics are a feature that offers type information for static checks in the compiler. However, this type information is not preserved by the compiler. That is, at runtime, given a concrete instance of a `Foo` object, it is not possible to know the value of `T` (the best one can do is guess by checking `value`). This is known as *type erasure*, and is problematic for dynamic analysis tools.

The Java compiler also accepts generic classes without any instantiated type parameters. This is what most dynamic analysis and test generation tools use, and it essentially amounts to assuming that every type variable represents `Object`.

```
Stack stack = new Stack();
stack.push("test");
Object o = stack.pop();
```

As indicated with the wavy underline, a compiler will issue a warning in such a case, as static type checking for generic types is impossible this way.

Besides classes, it is also possible to parameterize methods using generics. A generic method has a type parameter which is inferred from the values passed as parameters. For example, consider the following generic method:

```
public class Foo {
    public <T> List<T> getNList(T element, int length) {
        List<T> list = new ArrayList<T>();
        for(int i = 0; i < length; i++)
            list.add(element);
        return list;
    }
}
```

The method `getNList` creates a generic list of length `length` with all elements equal to `element`. `List` is a generic class of the Java standard library, and the generic parameter of the list is inferred from the parameter `element`. For example:

```
Foo foo = new Foo();
List<String> stringList = foo.getNList("test", 10);
List<Integer> intList = foo.getNList(0, 10);
```

In this example, the same generic method is used to generate a list of strings and a list of numbers.

## 2.2 Automated Test Generation

Testing is a common method applied to ensure that software behaves as desired. Developer testing involves writing test cases that exercise a program through its application programmer interface (API). This has been particularly popularized by the availability of convenient test automation frameworks for unit testing, such as JUnit. Test cases may be written before the actual implementation such that they serve as specification (test-driven development), they can be written while explicitly testing a program in order to find bugs, or they can be written in order to capture the software behavior in order to protect against future (regression) bugs. However, writing test cases can be a difficult and error-prone task, and manually written suites of tests are rarely complete in any sense. Therefore, researchers are investigating the task of automating test generation, such that developer-written tests can be supplemented by automatically generated tests.

Technically, the task of automatically generating unit tests consists of two parts: First, there is the task of generating suitable test inputs, i.e., individual sequences of calls that collectively exercise the class under test (CUT) in a comprehensive way. Second, there is the task of determining whether these tests find any bugs, i.e., the generated test cases require test oracles.

Automatically generated test cases are particularly useful when there are *automated* oracles, such that finding faults becomes a completely automated task. Fully automated oracles are typically encoded in code contracts or assertions, and in absence of such partial specifications, the most basic oracle consists of checking whether the program crashes [11] (or in the case of a unit, whether an undeclared exception occurs).

The alternative to such automated oracles consists of adding oracles in terms of *test assertions* to the generated unit tests. In a scenario of regression testing where the objective is simply to capture the current behaviour, this process is fully automated. Alternatively, the developer is expected to manually add assertions to the generated unit tests. This process can further be supported by suggesting possible assertions [6], such that the developer just needs to confirm whether the observed behaviour is as expected, or erroneous.

Popular approaches to automated unit test generation include *random testing* [11], *dynamic symbolic execution* [7], and *search-based testing* [10]. In random testing, sequences of random calls are generated up to a given limit. A simple algorithm to generate a random test for an object oriented class is to randomly select methods of the class to add, and for each parameter to randomly select previously defined assignable objects in the test, or to instantiate new objects of the required type by calling one of its constructors or factory methods (chosen randomly). Given that random tests can be very difficult to comprehend, the main application of such approaches lies in exercising automated oracles. Dynamic symbolic execution systematically tries to generate inputs that cover all paths for a given entry function by applying constraint solvers on path conditions. Consequently, entry functions need to be provided, for example in terms of parameterized unit tests [13]. Finally, search-based testing has the flexibility

of being suitable for almost any testing problem, and generating unit tests is an area where search-based testing has been particularly successful.

In search-based testing [1,10], the problem of test data generation is cast as a search problem, and search algorithms such as hillclimbing or genetic algorithms are used to derive test data. The search is driven by a fitness function, which is a heuristic that estimates how close a candidate solution is to the optimum. When the objective is to maximize code coverage, then the fitness function does not only quantify the coverage of a given test suite, but it also provides guidance towards improving this coverage. To calculate the fitness value, test cases are usually executed using instrumentation to collect data. Guided by the fitness function, the search algorithm iteratively produces better solutions until either an optimal solution is found, or a stopping condition (e.g., timeout) holds.

The use of search algorithms to automate software engineering tasks has been receiving a tremendous amount of attention in the literature [8], as they are well suited to address complex, non-linear problems.

### 2.3 The Woes of Generics in Automated Testing

So why are generics a problem for test generation? Java bytecode has no information at all about generic types — anything that was generic on the source code is converted to type `Object` in Java bytecode. However, many modern software analysis and testing tools work with Java bytecode rather than sourcecode. Some information can be salvaged using Java *reflection*: It is possible to get the exact signature of a method. For example, in the following snippet we can determine using Java reflection that the parameter of method `bar` is a list of strings:

```
public class Foo {
    public void bar(List<String> stringList) {
        // ...
    }
}
```

However, consider the following variation of this example:

```
public class Foo<T> {
    public void bar(List<T> stringList) {
        // ...
    }
}
```

If we now query the signature of method `bar` using Java reflection, we learn that the method expects a list of `T`. But what is `T`? Thanks to type erasure, for a given object, it is impossible to know for a given instance of `Foo` what the type of `T` is<sup>1</sup>.

---

<sup>1</sup> Except if we know the implementation of `Foo` and `List` such that we can use reflection to dig into the low level details of the list member of `Foo` to find out what the type of the internal array is.



Our only hope is if we know how `Foo` was generated. For example, as part of the test generation we might instantiate `Foo` ourselves — yet, when doing so, what should we choose as concrete type for type variable `T`? We did not specify a type boundary for `T` in the example, and the implicit default boundary is `Object`. Consequently, we have to choose a concrete value for type `T` out of the set of all classes that are assignable to `Object`. In this example, this means *all* classes on the classpath are candidate values for `T`, and typically there are *many* classes on the classpath.

However, things get worse: It is all too common that methods are declared to return objects or take parameters of generic classes where the type parameters are given with a wildcard type. Even worse, legacy or sloppily written code may even completely omit type parameters in signatures. A wildcard type or an omitted type parameter looks like `Object` when we inspect it with Java reflection. So if we have a method that expects a `List<?>`, what type of list should we pass as parameter? If we have received a `List<?>`, all we know is that if we get a value out of it, it will be an `Object`. In such situations, if we do not guess the right type, then likely we end up producing useless tests that end up in `ClassCastExceptions` and cover no useful code. To illustrate this predicament, consider the following snippet of code:

```
@Test
public void testTyping(){
    List<String> listString = new LinkedList<String>();
    listString.add("This is a list for String objects");
    listString.add("Following commented line would not
        compile");
    //List<Integer> listStringButIntegerType =
        listString;
    List erasedType = listString;
    List<Integer> listStringButIntegerType = erasedType;
    listStringButIntegerType.get(0).intValue();
}
```

If we define a parametrized list as to contain only `String` objects, then it is not possible to assign it to a list for integers; the compiler will not allow it. But, if we first assign it to a generic list, then this generic list can be assigned to an integer list without any compilation error. In other words, a reference to an integer list does not give any guarantee that it will contain only integers. This is a particular serious problem for automated test data generation because, if generics are not properly handled, we would just end up in test cases that are full of uninteresting `ClassCastExceptions`. For example, if a method of the CUT takes as input a list of `Strings`, then it would be perfectly legit (i.e., it will compile) to give as input a generic list that rather contains integers.

Considering all this, it is not surprising that research tools on automated test generation have steered clear of handling generics so far.

### 3 Generating Tests for Generic Classes

The problem of generics becomes relevant whenever a test generation algorithm attempts to instantiate a new object, or to satisfy a parameter for a newly inserted method call. For example, this can be the case of a random test generation algorithm exploring sequences of calls, but it can just as well be part of a genetic algorithm evolving test suites. Assume that our test generation algorithm decides to add a call to the method `bar` defined as follows:

```
public class Foo<T> {
    public void bar(T baz) {
        // ...
    }
}
```

The signature of `bar` does not reveal what the exact type of the parameter should be. In fact, given an object of type `Foo`, when we query the method parameters of method `bar` using Java's standard reflection API reveals that `baz` is of type `Object`! Fortunately, the reflection API was extended starting in Java version 1.5, and the extended API adds for each original method a variant that returns generic type information. For example, whereas the standard way to access the parameters of a `java.lang.reflect.Method` object is via method `getParameters`, there is also a generic variant `getGenericParameters`. However, this method only informs us that the type of `baz` is `T`.

Consequently, the test generator needs to consider the concrete instance of `Foo` on which this method is called, in order to find out what `T` is. Assume the test generator decides to instantiate a new object of type `Foo`. At this point, it is necessary to decide on the precise type of the object, i.e., to instantiate the type parameter `T`. As discussed earlier, any class on the classpath is assignable to `Object`, so any class qualifies as candidate for `T`. As randomly choosing a type out of the entire set of available classes is not a good option (i.e., the probability of choosing an appropriate type would be extremely low), we need to restrict the set of candidate classes.

To find a good set of candidate classes for generic type parameters, we can exploit the behaviour of the Java compiler. The compiler removes the type information as part of type erasure, but if an object that is an instance of a type described by a type variable is used in the code, then before the use the compiler inserts a `cast` to the correct type. For example, if type variable `T` is expected to be a string, then there will be a method call on the object representing the string or it is passed as a string parameter to some other method. Consequently, by looking for casts in the bytecode we can identify which classes are relevant. Besides explicit casts, a related construct giving evidence of the concrete type is the `instanceof` operator in Java, which takes a type parameter that is preserved in the bytecode.

The candidate set is initialized with the default value `Object`. To collect the information about candidate types, we start with the dedicated class under test (CUT), and inspect the bytecode of each of its methods for `castclass` or `instanceof` operators. In addition to direct calls in the CUT, the parameters may be used in subsequent calls, therefore this analysis needs to be interprocedural. Along the analysis, we can also consider the precise method signatures, which may contain concretizations of generic type variables. However, the further away from the CUT the analysis goes, the less related the cast may be to covering the code in the CUT. Therefore, we also keep track of the depth of the call tree for each type added to the set of candidate types.

Now when instantiating a generic class `Foo`, we randomly choose values for its type parameters out of the set of candidate classes. The probability of a type being selected is dependent on the depth in the call tree, and in addition we need to determine the subset of the candidate classes that are compatible with the bounds of the type variable that is instantiated. Finally, it may happen that a generic class itself ends up in the candidate set. Thus, the process of instantiating generic type parameters is a recursive process, until all type parameters have received concrete values. To avoid unreasonably large recursions, we put an upper boundary on the number of recursive calls, and use a wildcard type if the boundary has been reached.

## 4 EVOSUITE: A Unit Test Generator Supporting Generics

We have extended the EVOSUITE unit test generation tool [3] with support for Java generics according to the discussed approach. EVOSUITE uses a genetic algorithm (GA) to evolve test suites. The objective of the search in EVOSUITE is to maximize code coverage, so the fitness function does not only quantify the coverage of a given test suite, but it also provides guidance towards improving this coverage. For example, in the case of branch coverage, the fitness function considers for each individual branching statement how close it was to evaluating to true and to false (i.e., its branch distance), and thus can guide the search towards covering both outcomes.

The GA has a population of candidate solutions, which are test suites, i.e., sets of test cases. Each test case in turn is a sequence of calls (like a JUnit test case). EVOSUITE generates random test suites as initial population, and these test suites are evolved using search operators that mimic processes of natural evolution. The better the fitness value of an individual, the more likely it is considered for reproduction. Reproduction applies mutation and crossover, which modify test suites according to predefined operators. For example, crossover between two test suites creates two offspring test suites, each containing subsets from both parents. Mutation of test suites leads to insertion of new test cases, or change of existing test cases. When changing a test case, we can remove, change, or insert new statements into the sequence of statements. To create a new test case, we simply apply this statement insertion on an initially empty sequence until the test has a desired length. Generic classes need to be handled both, when

generating the initial random population, and during the search, when test cases are mutated. For example, mutation involves adding and changing statements, both of which require that generic classes are properly handled. For details on these search operators we refer to [5].

To demonstrate the capabilities of the improved tool, we now show several simple examples on which test generation tools that do not support generics fail (in particular, we verified this on RANDOOP [11], DSC [9], Symbolic PathFinder [2], and PEX [13]).

The first example shows the simple case where a method parameter specifies the exact signature. As the method accesses the strings in the list (by writing them to the standard output) outwitting the compiler by providing a list without type information is not sufficient to cover all the code – anything but an actual list of strings would lead to a `ClassCastException`. Thus, the task of the test generation tool is to produce an instance that exactly matches this signature:

```
import java.util.List;

public class GenericParameter {

    public boolean stringListInput(List<String> list){
        for(String s : list)
            System.out.println(s.toLowerCase());
        if(list.size() < 3)
            return false;
        else
            return true;
    }
}
```

For this class, EVOSUITE produces the following test suite to cover all branches:

```
public class TestGenericParameter {
    @Test
    public void test0() throws Throwable {
        GenericParameter genericParameter0 = new GenericParameter();
        LinkedList<String> linkedList0 = new LinkedList<String>();
        linkedList0.add("");
        linkedList0.add("");
        linkedList0.add("");
        boolean boolean0 = genericParameter0.stringListInput(linkedList0);
        assertEquals(true, boolean0);
    }

    @Test
    public void test1() throws Throwable {
        GenericParameter genericParameter0 = new GenericParameter();
        LinkedList<String> linkedList0 = new LinkedList<String>();
        boolean boolean0 = genericParameter0.stringListInput(linkedList0);
        assertEquals(false, boolean0);
    }
}
```

As second example, consider a generic class that has different behavior based on what type it is initialized to, such that a test generator needs to find appropriate values for type parameter T in order to cover all branches:

```
import java.util.List;

public class GenericsExample<T,K> {

    public int typedInput(T in){
        if(in instanceof String)
            return 0;
        else if(in instanceof Integer)
            return 1;
        else if(in instanceof java.net.ServerSocket)
            return 2;
        else
            return 3;
    }
}
```

Again EVOSUITE is able to create a branch coverage test suite easily:

```
public class TestGenericsExample {
    @Test
    public void test0() throws Throwable {
        GenericsExample<ServerSocket, ServerSocket> genericsExample0 = new
            GenericsExample<ServerSocket, ServerSocket>();
        ServerSocket serverSocket0 = new ServerSocket();
        int int0 = genericsExample0.typedInput(serverSocket0);
        assertEquals(2, int0);
    }

    @Test
    public void test1() throws Throwable {
        GenericsExample<Integer, Object> genericsExample0 = new
            GenericsExample<Integer, Object>();
        int int0 = genericsExample0.typedInput((Integer) 1031);
        assertEquals(1, int0);
    }

    @Test
    public void test2() throws Throwable {
        GenericsExample<String, ServerSocket> genericsExample0 = new
            GenericsExample<String, ServerSocket>();
        int int0 = genericsExample0.typedInput("");
        assertEquals(0, int0);
    }

    @Test
    public void test3() throws Throwable {
        GenericsExample<Object, Object> genericsExample0 = new
            GenericsExample<Object, Object>();
        Object object0 = new Object();
        int int0 = genericsExample0.typedInput(object0);
        assertEquals(3, int0);
    }
}
```

To evaluate the examples, we compare with other test generation tools for Java described in the literature. Research prototypes are not always freely available, hence we selected tools that are not only available online, but also popular (e.g., highly cited and used in different empirical studies). In the end, we selected RANDOOP [11], DSC [9], Symbolic PathFinder [2], and PEX [13].

Like for EVOSUITE, setting up RANDOOP to generate test cases for `GenericsExample` is pretty straightforward. Already after a few seconds, it has generated JUnit classes with hundreds of test cases. However, RANDOOP generated no tests that used a list as input for the `stringListInput` method (0% coverage) or a `ServerSocket` for `typedInput`.

A popular alternative to search-based techniques in academia is dynamic symbolic execution (DSE). For Java, available DSE tools are DSC [9] and Symbolic PathFinder [2]. However, these tools assume static entry functions (DSC), or appropriate test drivers that take care of setting up object instances and selecting methods to test symbolically<sup>2</sup>. As choosing the “right” type for a `GenericsExample` object instantiation is actually part of the testing problem (e.g., consider `typedInput` method), then JPF does not seem to help in this case.

To overcome this issue, we also investigated PEX, probably the most popular DSE tool, but a tool that assumes C#. However, generics also exist in C#, so it is a good opportunity to demonstrate that the problem addressed in this paper is not a problem specific to Java.

We translated the `GenericsExample` class to make a comparison, resulting in the following C# code.

```
using System;
using System.Collections.Generic;

public class GenericsExample<T> {
    public int typedInput(T input){
        if (input is String)
            return 0;
        else if (input is Int32)
            return 1;
        else if (input is System.Collections.Stack)
            return 2;
        else
            return 3;
    }
}
```

Note that we replaced the TCP socket class with a `Stack` class such that the example can also be used with the web interface for PEX, `PexForFun`<sup>3</sup>. Like the other DSE tools, PEX assumes an entry function, which is typically given by a

<sup>2</sup> <http://javapathfinder.sourceforge.net/>, accessed June 2013.

<sup>3</sup> <http://www.pexforfun.com/>, accessed June 2013.

parameterized unit test. For example, the following code shows an entry function that can be used to explore the example code using DSE:

```
using Microsoft.Pex.Framework;

[PexClass]
public class TestClass {

    [PexMethod]
    public void Test<T>(T typedInput) {
        var ge = new GenericsExample<T>();
        ge.typedInput(typedInput);
    }
}
```

This test driver and the `GenericsExample` class can be used with Pex on Pex4Fun, and doing so reveals that PEX does not manage to instantiate `T` at all (it just attempts `null`).

## 5 Conclusions

Generics are an important feature in object-oriented programming languages like Java. However, they pose serious challenges for automated test case generation tools. In this short paper, we have presented a simple techniques to handle Java generics in the context of test data generation. Although we implemented those techniques as part of the EVOSUITE tool, they could be used in any Java test generation tool.

We showed the feasibility of our approach on artificial examples. While EVOSUITE was able to achieve 100% coverage in all these examples quickly, other test generation tools fail — even if they would be able to handle equivalent code without generics (e.g., in the case of RANDOOP).

Beyond the simple examples of feasibility, as future work we will perform large scale experiments to determine how significant the effect of generics support is in practice, for example using the SF100 corpus of open source projects [4].

EVOSUITE is a freely available tool. To learn more about EVOSUITE, visit our Web site:

<http://www.evosuite.org>

**Acknowledgments.** This project has been funded by a Google Focused Research Award on “Test Amplification” and the Norwegian Research Council.

## References

1. Ali, S., Briand, L., Hemmati, H., Panesar-Walawege, R.: A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering (TSE)* 36(6), 742–762 (2010)

2. Anand, S., Păsăreanu, C.S., Visser, W.: JPF-SE: A symbolic execution extension to java pathFinder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 134–138. Springer, Heidelberg (2007)
3. Fraser, G., Arcuri, A.: EvoSuite: Automatic test suite generation for object-oriented software. In: ACM Symposium on the Foundations of Software Engineering (FSE), pp. 416–419 (2011)
4. Fraser, G., Arcuri, A.: Sound empirical evidence in software testing. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 178–188 (2012)
5. Fraser, G., Arcuri, A.: Whole test suite generation. *IEEE Transactions on Software Engineering* 39(2), 276–291 (2013)
6. Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)* 28(2), 278–292 (2012)
7. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: ACM Conference on Programming language design and implementation (PLDI), pp. 213–223 (2005)
8. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45(1), 11 (2012)
9. Islam, M., Csallner, C.: Dsc+mock: A test case + mock class generator in support of coding against interfaces. In: International Workshop on Dynamic Analysis (WODA), pp. 26–31 (2010)
10. McMinn, P.: Search-based software test data generation: A survey. *Software Testing, Verification and Reliability* 14(2), 105–156 (2004)
11. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 75–84 (2007)
12. Parnin, C., Bird, C., Murphy-Hill, E.: Adoption and use of Java generics. *Empirical Software Engineering*, 1–43 (2012)
13. Tillmann, N., de Halleux, N.J.: Pex — white box test generation for .NET. In: International Conference on Tests And Proofs (TAP), pp. 134–253 (2008)



# Constraint-Based Automated Generation of Test Data

Hans-Martin Adorf and Martin Varendorff

Mgm Technology Partners, Frankfurter Ring 105a,  
80807 München, Germany

{Hans-Martin.Adorf, Martin.Varendorff}@mgm-tp.com

**Abstract.** We present a novel method for automatically generating artificial test data that are particularly suited for testing form-centric software applications with several thousand input fields. The complex validation rules for user input are translated to a constraint satisfaction problem (CSP), which is solved using an off-the-shelf SMT-solver. In order to exert pressure onto the software under test, the generated test data have to incorporate extreme and special values (ESVs) for each field. The SMT-solver is aided by a sophisticated graph-based cluster algorithm and by other heuristic methods in order to reduce the complexity of the CSPs. With further optimizations, the test data generator now routinely generates a complete set of test data records for large form-centric applications within less than two hours. The test data generator described here is operationally being used for automated tests of form-centric Web-applications, within an iterative development process emphasizing very early testing of software applications.

**Keywords:** automated test data generation, constraint satisfaction problems, form-centric software applications, functional testing, satisfiability modulo theories, software quality assurance.

## 1 Introduction

The construction of comprehensive test data sets for large software applications is a complex, time consuming, and error-prone process. The sets have to include valid data records in order to support positive functional tests, as well as invalid data for negative tests.

In order to exert pressure onto the application one needs appropriate test coverage of the space of possible input data. Appropriate coverage requires seeking challenging input values for input fields, such as extreme or otherwise special values (ESVs), while simultaneously fulfilling all required validation rules for the input data, or, conversely, explicitly violating some of them. In order to limit the test execution time, the size of the test data set (i.e. the total number of test data records) should be small, which forces one to incorporate as many ESVs as possible into each test data record.

Large applications may require thousands of input values, which are subject to a similar number of validation rules. Test data sets have to be generated frequently. Therefore the automation of the entire data generation process is not only highly desirable, but a necessity for business-critical applications. This paper presents how we solved the problem of generating high-quality artificial test data sets that are mainly used within an efficient automated quality assurance process.

## 2 Testing Form-Centric Software Applications

Below we will concentrate on the quality assurance of “form-centric” software applications. The main purpose of such an application consists in providing users with “free-text” fields on forms for data entry. The input to a form-centric application ought to be validated before it is transferred to a processing system. This validation assures that each field entry is syntactically correct (single-field validation), and that the combination of entries into different fields is consistent (cross-field validation). The combination of definitions of fields and of validation rules is called a “validation rule-base”.

A prime example of a large form-centric application is software supporting a tax declaration, where the taxpayer must enter names, addresses, earnings, calendar dates, etc. into free-text fields. Tax-related applications are particularly demanding, since they may contain up to two dozens of forms, some of which may occur in several instances (e.g. one form per child). Each form contains many fields, which in addition can be repeated (e.g. a list of deductibles). As stated above, the number of accompanying validation rules usually has the same order of magnitude as the number of fields, which can be in the range of several thousands.

Another example of a form-centric application is software supporting an applicant for an insurance policy, or an agent acting as an intermediary between the applicant and an insurance company.

### 2.1 The Test Process

Form-centric applications, as any other software application, must be tested before being deployed in the field. In our case, the high quality demands of our customers have so far been met by executing intensive manual tests, which, due to the large number of fields, are very tiring and costly. Over time, in order to save labor and to reduce costs, manual tests are replaced, as far as possible, by automated tests. For the automation of functional tests mgm has developed a test framework called jFunk [1].

At mgm we follow a software development life cycle emphasizing early and frequent testing of the applications under development (figure 1). A development cycle typically spans across several months. Within such a cycle, stable versions of the software are regularly produced in iterations, and they are tested mainly using automated functional tests. Test results are fed back into the development of the next iteration in order to prevent a build-up of defects within a cycle. The duration of the iterations decreases towards the end of a cycle. Accordingly, the frequency of executing automated tests increases, which requires a timely creation of test data sets.

### 2.2 Requirements for the Test Data Generation

In this section we will take a closer look at the requirements for, and the complexity of, the task of creating test data for a large form-centric application.

The validation rules in a rule base vary in complexity. Simple ones only check the presence of values in one or more fields. Others consist of equalities, inequalities, or disequalities between the values in two or more fields. Even more complex rules use functions of field values within numerical predicates. The most complex rules combine all types of conditions within sizable Boolean expressions.

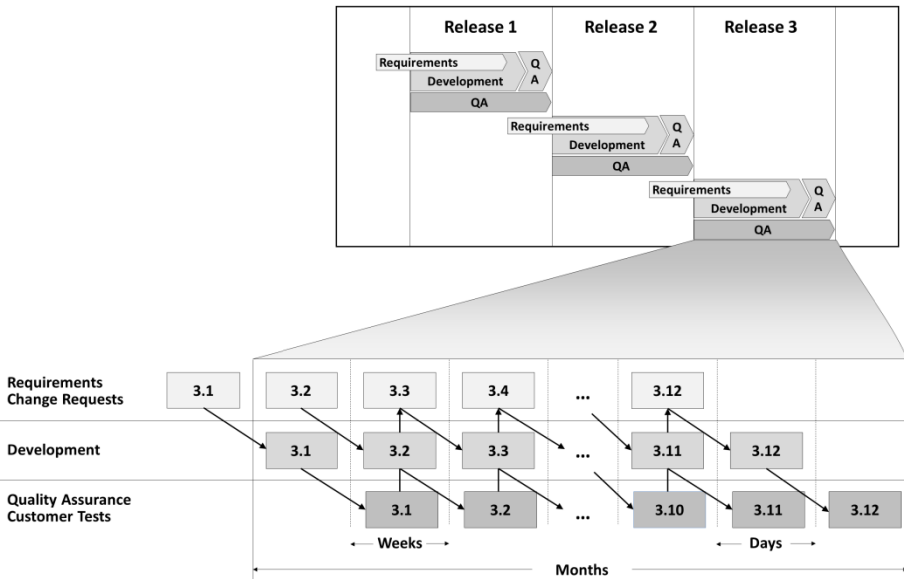


Fig. 1. Mgm’s “Very Early Testing” quality assurance process

For each input item, i.e. a field which, as explained above, may appear in several instances, values need to be generated. The latter must comprise as many predefined extreme and special values (ESVs) as possible. Each ESV should, if feasible at all, occur in at least one test data record. There is no requirement to consider many (all???) combinations of interesting values, as a “combinatorial explosion” would arise. If a test requires a particular combination of values, then these values are predefined for the test data generation process, or such test cases are executed manually.

The execution of functional test cases (each using a single record from the test data set) is time consuming, since each test may run for several minutes. Therefore the size of a set with sufficient test coverage, as defined above, should be minimized. This requirement entails that the ESV density in each test data record should be maximized.

Within a software development cycle the rule-base associated with an application usually undergoes several changes. If such changes are small, a large proportion of existing test data is usually still valid, and can therefore be used for testing the application. But if the rule-base changes are substantial, new test data will have to be generated. This requirement entails that the test data generation process has to be correspondingly fast. For reasons of practicality our goal has been to accomplish a turn-around time of one day between the delivery of a new rule-base and the end of the data generation process.

For the test of large applications it is impossible to *manually* generate a sufficient amount of test data records with the required quality and within such stringent time limits. Therefore an *automated* generation approach is mandatory.

### 3 An Automated Test-Data Generator

The automation of test data generation – known to be a challenging, highly complex task – is not a complete novelty. Early trials, which date back almost four decades, even include the treatment of systems of non-linear equations [2]. Test data generators described in the literature (see e.g. [3]) are often based on a mathematical modeling process comprising the following phases: (1) construct a control flow graph (usually by some form of code analysis), (2) select an execution path, and (3) generate test data for the latter. Each execution path entails a so-called “path predicate”. If the predicate is sufficiently simple, it can directly be submitted to a suitable solver (see e.g. [4], [5]). If there is no solution, the path cannot be followed at run-time.

All these methods have to satisfy some constraint(s), but since solving constraint satisfaction problems (CSPs) is particularly difficult, often one has to resort to heuristic techniques [3].

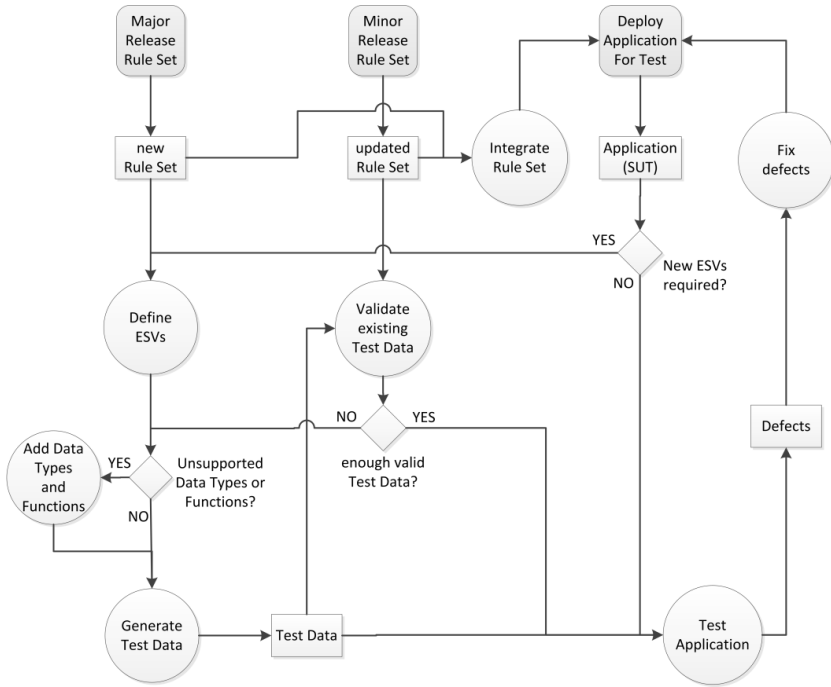
Over the past five years, the quality assurance division at mgm has developed an automated rule-based test data generator (R-TDG) fulfilling the requirements described above. Our test data generator resembles the classical method of generating test data insofar as it also uses predicates and a CSP-solver. However, unlike the classical method, we do not have to perform any code analysis (which is notoriously difficult) in order to generate a predicate. Instead, we are exploiting the validation rule-base that accompanies each of our form-centric applications. From the rule-base a “fundamental predicate” is directly derived, which concisely describes the domain of valid input data. This predicate is systematically varied in order to incorporate all those ESVs that should be present in a comprehensive, valuable set of test data records. Each variation represents a CSP that, probably after some simplifications, can directly be submitted to a CSP-solver, a Satisfiability Modulo Theories (SMT) solver in our case.

In principle, the operating procedure for the R-TDG is straightforward:

1. Define several configuration parameters, including the number of desired instances of the forms and of the field lines.
2. Define required ESVs for all field instances.
3. Add definitions representing new data types and functions.
4. Run the R-TDG and produce random test data records.
5. Validate these records, and feed them into functional tests.

Figure 2 shows how this procedure integrates with the development process of the application and its corresponding rule set. Three triggers may initiate a test data generation process:

1. A major release of a rule set containing new field definitions or constraints: Test data have to be generated for a functional test of the application.
2. A minor release of a rule set containing updated rules and only minor changes in field definitions: Old test data may be reused, or new test data have to be generated for a functional test of the application.
3. The deployment of a new version of the application for a functional test: New ESVs may be necessary, requiring the generation of new test data for a functional test of the application.



**Fig. 2.** Procedure for testing using a new or updated rule set, or a new version of the application. The necessary test data are generated by the R-TDG.

The data-generation procedure within the R-TDG works as follows:

1. Read the configuration.
2. Read the field and rule definitions
3. Translate each field definition into a number of associated variable definitions, and each validation rule into a number of associated constraints.
4. Assemble variables and constraints into a base CSP.
5. Read the field definitions, and generate ESVs for each field.
6. Use ESVs for creating variants of the base CSP.
7. Solve the resulting CSPs using an SMT-solver.
8. Translate the solutions back into the format required by the application.
9. Validate these records.

However, the simplicity of this process is misleading since a number of problems arise which have to be solved before valid test data emerge. Let us discuss these problems in some more detail.

### 3.1 Translating Field and Rule Definitions into Variable and Constraint Definitions

Several configuration parameters govern the whole data generation process. Two of those influence the translation, namely the actual number of instances of the forms

holding the fields (e.g. instances for up to 14 children in a tax declaration), and the actual number of field instances. We refer to the actual numbers of form and field instances as their “multiplicities”.

Consider, for example, the declaration of travel expenses: one “line” (consisting of a set of field instances) might consist of the amount of trips, the origin (i.e. the address of your employer), the destination (i.e. address of your destination), and the distance. A validation rule might restrict the number of trips to 366. When we generate test data, we have to set the desired multiplicity to less than 366 for the fields above.

A form multiplicity of  $m$  combined with a field multiplicity of  $n$  leads to  $m$  times  $n$  input items. The multiplicity values  $m$  and  $n$  can be quite large (1000 and more), but in practice we rarely use values exceeding 2.

An important obstacle to deal with consists in the fact that an input item may be empty. (An item being empty is equivalent to a Java-variable containing the value null.) No SMT-solver known to us can handle variables that have no value. Therefore, for each item, we have introduced a binary “occupation variable” that holds the information whether the corresponding “value variable” is null or not. Such a variable pair is semantically equivalent to a single variable whose value may be unspecified. For instance, a field item of type Boolean is translated into a variable pair that together can represent the three values: *true*, *false*, and *undefined*. We therefore refer to the logic implemented in the rules as “three-valued” logic.

Fields are declared in field definitions which, apart from the data type, hold additional information such as the minimum/maximum field length (number of characters). This information has to be translated into variable and constraint definitions that are comprehensible to the SMT-solver. Each field acts as a template, which is translated into  $2 * m * n$  corresponding variables.

The rules, which form the other half of the input to the R-TDG, are also mere templates since initially only the *maximum* number of instances of the forms and of the fields occurring in those rules are specified. Therefore each abstract rule must be replicated according to the *actual* multiplicity of the forms and fields occurring in that rule. As a consequence, each abstract rule is translated into several concrete constraints. The constraints have to be formulated in such a way that they properly accommodate the value and occupation variables explained above. In effect, each rule is normally translated into  $m * n$  corresponding constraints.

Another important issue to deal with is functions that occur in rule conditions. SMT-solvers do not comprehend proper functions, but only constraints over a very limited set of data types. Therefore each function occurring in a rule condition needs to be translated into a corresponding equivalent constraint.

The three-valued logic vastly complicates all logical expressions and all functions operating on field items. Consider the simple predicate  $z = \max_2(x, y)$ . This expands to 4 cases:

1.  $z = \max(x, y)$ , if both  $x$  and  $y$  have values (here  $\max$  is the ordinary maximum function, which can be resolved into a logical condition),
2.  $z = x$ , if  $x$  has a value and  $y$  does not,
3.  $z = y$ , if  $y$  has a value and  $x$  does not,
4.  $z = lb$ , if neither  $x$  nor  $y$  is defined (here  $lb$  is the lower bound for the variables  $x$  and  $y$ ).

A concise representation of some functions is also an issue. Consider, for example, a predicate  $y = f(x_1, x_2, \dots, x_n)$  which requires that at least one of the number of input items  $x_1, x_2, \dots, x_n$  has to have a non-null value. If for the occupation variables we were using ordinary Boolean data types (taking the values *true* or *false*), a complex and long winding expression for the function  $f$  would result. If we instead use bit data types (taking the values 0 or 1), a very compact formulation of the predicate emerges in form of a numeric inequality.

### 3.2 Representing Data Types Unsupported by SMT-Solvers

The translation process described so far is incomplete. What is missing is a description of how to deal with data types which occur in the rule-bases, but which have no direct counterpart in the SMT-world. A case in point is a decimal number which is an important data type for form-centric applications. Other such data types encountered in our rule-bases are calendar dates and date ranges. Again, no SMT-solver known to us can directly deal with calendar dates. We finally mention the important string data types. While dealing with string data types and string constraints is an important current research topic, none of the off-the-shelf SMT-solvers is capable of dealing with strings.

Below we discuss in some detail how we represent these data types in our CSPs.

#### Decimal Numbers

A decimal number is a rational number that possesses a representation with a finite number of digits after the decimal separator. For instance, a currency amount is a decimal number with at most two decimal digits. On the other hand the rational number  $1/3$  is not a decimal number.

In order to represent a decimal number within a CSP we use a pair consisting of a rational number and an accompanying constraint (called “decimal constraint”). The latter requires that a certain multiple of the number must be an integer. E.g. a standard currency amount with 2 post-decimal digits (say Euros with Cents) multiplied by 100 must be an integer.

Unfortunately those innocent looking decimal constraints can lead to severe performance issues for the SMT-solver.

#### Calendar Dates

Fields with values of type calendar date present another problem since the available SMT-solvers do not encompass the data type “calendar date”. In the rule-base the (external) representation of a date value is always a string, such as “11.03.1956”, but a typical cross field constraint uses functions that require separate access to the day in the month, the month, and the year of the calendar date. Comparisons with other date variables or constants, such as *before*, *at the same time*, or *later*, may have to be performed on parts of a date value, or on the whole value.

In order to enable a CSP-solver to operate on calendar dates, we represent any calendar date by an integer equivalent to a ‘relative’ day, starting from 01.01.1900 (which is day 1). In an imperative or functional programming language it is easy to

implement accessor functions that retrieve the year, the month in the year, or the day in the month from such a relative day. However, we are dealing with a *declarative* CSP-language, and thus these accessors have to be implemented as constraints – a non-trivial task.

We found representations for all required date constraints. However the run-times are sometimes prohibitive. The only solution we have found so far consists in pre-assigning suitable values to a sufficient number of the date variables involved, and thereby remove these variables from the CSP in question. With this drastic measure we have been able to cut down the run-times to reasonable values, at the expense of losing some test-coverage.

### Strings and String-Constraints

In a typical form-centric application a large number of the fields are string fields. Each such field can contain a maximum number of characters, which is a simple single-field constraint. The character set that the user may choose from for his/her input is another constraint. In our applications, many fields are further constrained by one or more regular expressions.

Here is a simple example of the combination of two regular expressions due to different rules: an ID code must match the expression “[0-9]{5}”, but not match “00000”, which might be a pseudo-value reserved for “first time customer who has no ID yet”.

Solving CSPs over string variables is a current research topic (see e.g. [6], [7], and references therein). However, none of the off-the-shelf SMT-solvers presently includes a string data type.

In order to cope with string fields and associated string constraints we have used a heuristic approach that consists of the following elements: for a given field, the field-length constraint, the alphabet (character set) constraint, and any regular expression match constraint are considered as predicates over the field. Each predicate is replaced by the Boolean abstraction of the predicate, i.e. a Boolean variable which holds the logical value of the predicate. The modified CSP is then submitted to the SMT-solver, and the values of the artificial Boolean variables are read off the solution.

For each string field, the solution of the CSP consists of a list of regular expressions along with a list of Boolean values (match flags), which indicate whether the string value for the field should match the expression or not. In order to generate valid strings, fulfilling these predicates, a string generator was developed based on a publically available regular expression package [8]. Our string generator uses the well-known representation of a regular expression as a finite state automaton (see e.g. [9]). By walking the graph representing the automaton, strings can be generated that, in addition to meeting all constraints, may attempt to fulfill further requirements. The most important requirement is to exhaust the given character set as early and as well as possible. Our string generator accomplishes this goal.

The heuristic described above is not an exact method powerful to handle all occurring situations. Once more, the price to pay for our approximation consists in losing some of the viable solutions, and sometimes even generating inconsistencies. Fortunately, most of the time our heuristic works well and produces valid solutions to string constraints.



There are string constraints which, out of principle, cannot be treated by a string generator acting in a post-processing phase. These comprise the equality and inequality constraints between string variables, the substring function, and conversion functions where a suitable string is converted to a number or calendar date. These constraints do actually occur in our rule-bases, and properly solving CSPs that contain them would require a genuine string solver. In cooperation with the Technical University Munich such a string solver has been developed [7], but so far has not yet been incorporated into the productive R-TDG. Therefore in each of those cases a handcrafted workaround is still required.

### 3.3 Dealing with Non-linear Constraints

A major obstacle for almost any SMT-solver is non-linear constraints over numeric variables, such as  $z = x * y$  for some variables  $x$ ,  $y$ , and  $z$  (for an early account see e.g. [2]). Only recently a few SMT-solvers that can solve constraints comprising multinomial expressions have become available (see e.g. [10]).

For the time being, we linearize constraints such as the ones above, by manually replacing a sufficient number of variables by constant values. Of course, this way we lose ESV-coverage, but that is a modest price to pay, until we will be ready to move on to a solver capable of handling non-linear constraints.

### 3.4 Generating Extreme and Special Values

The R-TDG is expected not to produce arbitrary data records, but test data records that put the software under test (SUT) under pressure. Therefore, as explained above, the records have to include ESVs for the input items.

For a numeric input item an important ESV obviously is 0; other ESVs of interest are the minimum and the maximum admissible values. For a calendar date field 28<sup>th</sup> and 29<sup>th</sup> of February, and 1<sup>st</sup> of March are interesting ESVs. Dates at the boundaries of a quarter such as 1<sup>st</sup> of January and 31<sup>st</sup> of March are also interesting values. For a string field, the empty string and a string with the maximum allowed number of characters are interesting ESVs. For string fields it is important that each admissible character occurs in at least one ESV, if feasible at all. Of course, for all input items the null value is an important ESV.

For the production of ESVs we have implemented an automated ESV-generator. For each variable it produces a reasonable set of ESVs on the basis of the variable's generic field-type combined with some additional field-specific parameters such as the minimum/maximum field length.

In addition to predefined ESVs we usually include some random values that are treated in the same way as the deterministic ESVs.

The number of ESVs for a given CSPs is roughly proportional to the number of variables present. The number of ESVs to be generated for a given CSP averages about 3 to 5 times the number of variables contained in the CSP.

## 4 Solving the CSPs Efficiently

The problem of validating a given data set is straightforwardly solved with an algorithm with polynomial computational complexity. However, the associated *inverse* problem of generating test data has a computational complexity that is much higher than that of the *forward* problem. In almost all cases the problem of solving a CSP is NP-complete. It is therefore often difficult to obtain solutions to practical CSPs which, as in our case, may have many thousands of variables and many thousands of constraints.

Usually there no stringent correlation between the run-time and the size of the CSP measured by the number of variables or constraints. Nevertheless, CSPs with roughly the same number of variables and of constraints tend to be more difficult to solve than CSPs with an unbalanced number of variables and constraints. With few constraints compared to the number of variables the solution space is large, and with many more constraints than variables the search space for solutions can usually quickly be restricted, or contradictions are found which lead to an empty solution space.

Efficiency-boosting techniques are essential for a test data generator that is supposed to be useful in practice. It is important to offer reasonable turn-around times that fit to operational schedules. With a combination of measures we were able to reduce the make-span for generating tests data, even for our largest form-centric applications from about a week to about an hour. Some of those measures are explained below.

### 4.1 Partitioning a CSP into Independent Components

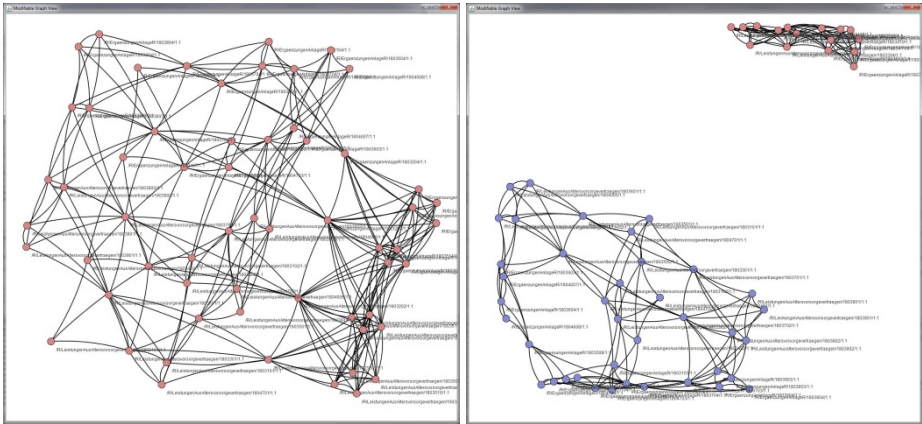
A major breakthrough consisted in partitioning any given CSP into independent CSP-components. Two variables are in the same component when they simultaneously occur only in constraints of this component. The components of a CSP can best be viewed in the undirected *primal constraint graph* in which each vertex corresponds to a variable, and two vertices are linked when the corresponding variables appear together in one of the constraints. Each component-CSP can be solved independently of the others.

One might think that CSP partitioning would be an integral part of any SMT-solver, but apparently that seems not to be the case. Fortunately, our external CSP-partitioning technique offers some advantages:

1. The partial solutions to the component-CSPs can freely be combined to form complete solutions to the parent CSP, i.e. valid test data records.
2. A trivial parallelization (i.e. concurrency) of the solution process becomes feasible, which further reduces the make-span for the test data generation. We are usually employing two to four threads, each operating on a different CSP-component. This approach exploits the resources of a state-of-the-art CPU with four physical (eight virtual) cores quite well. The reduction of the make-span is roughly proportional to the number of threads employed.

## 4.2 Decomposing a CSP via Cluster Analysis

The run-time of solving a given CSP is usually dominated by the run-time of solving the largest component-CSPs. Sometimes the largest component-CSPs still are too complex for our SMT-solver. Here it would be helpful if we could somehow identify one or more central variables which, if “removed” from the CSP, would permit the latter to be decomposed into two or more independent components. Again the child CSPs should more easily be solvable than the parent CSP.



**Fig. 3.** CSP-decomposition accomplished by the “vertex inbetweenness” cluster analysis algorithm. The primal constraint graph for a fairly large component of the CSP corresponding to SUT #1 (see table 1) is shown before (left) and after (right) running the algorithm. The algorithm has removed four central vertices in order to accomplish the decomposition.

In order to tackle this problem we have developed a very fast, concurrent, graph-based “vertex inbetweenness” cluster algorithm that iteratively identifies central graph vertices. The algorithm works on the primal constraint graph of the problematic component-CSP. A central vertex, once identified, can subsequently be removed from the graph (figure 3). After we had developed our algorithm we found that it had already been invented several decades before in the field of sociology [11].

In the context of CSPs a variable removal can be achieved by pre-assigning a value to the corresponding variable. (The pre-assigned value effectively transforms the variable into a constant which therefore disappears from the CSP.) Often the removal of a single variable does not yet allow a decomposition of the CSP. If so, the cluster algorithm has to be iterated until decomposition becomes possible.

The vertex-inbetweenness algorithm is really amazing in identifying the important central vertices, which eventually will allow a decomposition of the CSP into independent CSP-components of roughly comparable size. (The algorithm is greatly superior to e.g. the simple vertex-cut algorithm, which will often split a given CSP into one large and one tiny component.) From observing the effect of the vertex-inbetweenness algorithm, and from analyzing its inner workings, we can state that

it behaves as if it were “goal directed”. It seems to identify a potentially worthy cut consisting of one or more central vertices. During each iteration one of the vertices in the cut-set is removed until the cut has been achieved. Only after having accomplished this task the algorithm considers another potential cut.

The cluster algorithm is *the* power tool in our toolbox which we apply when nothing else helps to reduce the run-time of a critical (i.e. prohibitively long-running) component-CSP. We have had situations where the SMT-solver, working on a relatively small component-CSP, would not return within half a day. This, of course, is unacceptable. The CSP-decomposition, accomplished by applying the cluster algorithm, usually helps to dramatically reduce the make-span which the SMT-solver requires for generating solutions – sometimes by several orders of magnitude. The performance gain again comes at the expense of test coverage, since variable values oftentimes are fixed to constants that may not even be ESVs. However, obtaining test data with reduced test coverage is much better than obtaining test data too late or not at all!

### 4.3 Re-using Partial Solutions

Another very important efficiency boosting technique, which we recently implemented, consists in the following: when, for a component-CSP which is currently being worked on, a partial solution containing some ESVs has been obtained, it can be reused as a starter, when other ESVs are being added. For large rule-bases, reusing partial solutions has decreased the make-span for obtaining complete test data sets by a factor of 30 to 50. To us this large reduction factor came as a welcome surprise.

### 4.4 Handling Decimal Constraints

At first sight decimal constraints appear innocent. However, in practice we all too often suffer from severe efficiency problems. Particularly when the number of post-decimal digits is variable (as is the case for some fields in our SUTs), the run-time for solving the CSPs can increase dramatically. We therefore had to resort to the following heuristic: we solve the CSP without decimal constraints, and wait for a problem with decimal numbers to surface in at least one of the solutions. Only for those problematic variables, where a solution contains a rational number that cannot be represented as a decimal number with the allowed number of post-decimal digits, a decimal constraint is inserted into the CSP. The CSP is then solved once more.

Clearly, this heuristic approach requires two or more solution runs. However, in our experience, the overall efficiency, when using this heuristic, is still a lot higher compared to inserting a decimal constraint for all variables that represent a decimal number.

### 4.5 Timeouts

Sometimes the solution process for a component-CSP is well underway, when all of the sudden the SMT-solver “goes on strike”, meaning, it does not return within acceptable time. In such a situation a timeout is very helpful. When a process,

in which an SMT-solver runs, is overdue, it is cancelled. The latest ESV added to the current component-CSP is considered to be the trouble-maker, and is eliminated from further consideration. The solution process is then restarted, and it is guaranteed that the process not only terminates, but terminates in acceptable time.

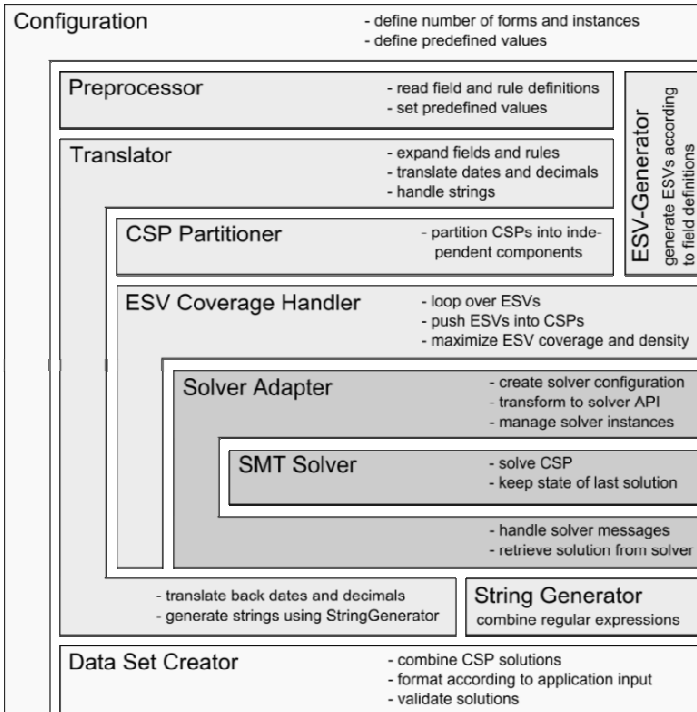


Fig. 4. Architecture of the rule-based test data generator (R-TDG)

## 5 Architecture

A sketch of the architecture of the R-TDG is shown in figure 4. The basic idea is to start with a rule set, translate it to a CSP with variables and constraints, partition the CSP into independent components, solve the CSP by an SMT-solver (which is accessed via a “Solver Adapter”), assemble the solutions of the CSPs, and translate them back to valid test data records. Each architectural component contained in the diagram is accompanied by a short description of its task.

The most complex components are the “Translator” and the “ESV Coverage Handler”. The Translator remedies the discrepancy between the “real world” definitions of fields and rules of the business domain and the available variable types and constraint definitions of an SMT-solver. The ESV Coverage Handler tries to include as many ESVs as are feasible into the resulting test data set, while keeping the overall size of the set at a minimum.

## 6 Results

Table 1 presents the results of generating test data for some large form-centric SUTs. For all SUTs the number of test data records is relatively small considering the number of ESVs that have to be incorporated. In addition, in all cases the make-span of generating test data is between  $\sim 10$  and  $\sim 100$  minutes, well below the one day target that we set out, when we began the development of the R-TDG.

**Table 1.** For several large form-centric software applications under test (SUTs) the table shows the size of the CSP (measured by the number of variables and the number of constraints), the number of CSP-components after partitioning and decomposition, the number of test data records produced, and the make-span of the data generation. Form and field multiplicities were set to two. Two threads were used in parallel.

SUT	# Variables	# Constraints	# Components	# Records	Make-span
#1	11,845	14,307	3309	51	25 min
#2	13,325	17,063	2129	90	92 min
#3	3,128	4,111	374	79	9 min
#4	2,934	3,736	381	86	12 min
#5	4,830	5,968	712	86	11 min
#6	4,199	5,069	452	79	19 min

## 7 Summary

We have presented a novel approach for generating artificial random test data that are suitable for testing large form-centric software applications. These contain up to several thousand fields that the user has to potentially fill in. The very same single-field and cross-field constraints that are used by a validator, for validating the user input to the application, are also used by our test data generator.

The set of base constraints is augmented by simple additional constraints, which insert into the solutions “extreme and special values” (ESVs), whose purpose is to exert pressure onto the software application under test (SUT). The variations of the initial constraint satisfaction problem (CSP) are solved by an off-the-shelf Satisfiability Modulo Theories (SMT) solver. Data types unknown to current SMT-solvers such as decimal numbers, calendar dates, and strings, have to be treated in special ways. Several heuristics, including an effective graph-clustering algorithm, have been put in place in order to enable an efficient generation of test data. Even for large form-centric applications the make-span for data generation has come down to less than 100 minutes.

For about four years, these data are regularly used mainly for automated tests of several large form-centric software applications that are being developed by our company.

## 8 Glossary

CSP	constraint satisfaction problem
ESV	extreme and special value
R-TDG	rule-based test data generator
SMT	satisfiability modulo theories
SUT	software (application) under test

## References

1. Dost, J., Nägele, R.: “jFunk Overview”, mgm technology partners GmbH (2012)
2. Howden, W.E.: Methodology for the Generation of Program Test Data. *IEEE Transactions on Computers* C-24(5), 554–560 (1975)
3. Edvardsson, J.: Survey on Automatic Test Data Generation. In: *Second Conf. on Computer Science and Engineering in Linköping (ECSEL)*, pp. 21–28 (1999)
4. DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering* 17(9), 900–910 (1991)
5. Gotlieb, A., Botella, B., et al.: Automatic test data generation using constraint solving techniques. *ACM SIGSOFT Software Engineering Notes* 23(2), 53–62 (1998)
6. Hooimeijer, P., Veanes, M.: *An Evaluation of Automata Algorithms for String Analysis*. Redmond City, Microsoft Research (2010)
7. Braun, M.: *A Solver for a Theory of Strings*. Fakultät für Informatik, Technische Universität München (2012)
8. Møller, A.: “Automaton.” Aarhus, Basic Research in Computer Science (BRICS) (2009)
9. Brüggemann-Klein, A.: Regular expressions into finite automata. In: Simon, I. (ed.) *LATIN 1992*. LNCS, vol. 583, pp. 87–98. Springer, Heidelberg (1992)
10. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver (2012)
11. Freeman, L.C.: A Set of Measures of Centrality Based on Betweenness. *Sociometry* 40, 35–41 (1977)

# RUP Alignment and Coverage Analysis of CMMI ML2 Process Areas for the Context of Software Projects Execution\*

Paula Monteiro<sup>1</sup>, Ricardo J. Machado<sup>2,4</sup>, Rick Kazman<sup>3</sup>,  
Cláudia Simões<sup>4</sup>, and Pedro Ribeiro<sup>2,4</sup>

<sup>1</sup> CCG-Centro de Computação Gráfica, Guimarães, Portugal

<sup>2</sup> Centro ALGORITMI, Escola de Engenharia, Universidade do Minho, Guimarães, Portugal

<sup>3</sup> University of Hawaii, Honolulu, USA

<sup>4</sup> Departamento de Sistemas de Informação, Universidade do Minho, Guimarães, Portugal

**Abstract.** The simultaneous adoption of CMMI and RUP allows the definition of “what to do” (with the support of CMMI) and “how to do” (with the support of RUP) in the context of executing software development projects. In this paper, our main contribution relates to the alignment of CMMI ML2 with RUP, in the context of executing software projects and the analysis of RUP coverage. We present the alignment for CMMI ML2 process areas, incorporating priority mechanisms. The adopted case study allows the analysis of the way RUP supports CMMI ML2 process areas taking into account the proposed alignment and the theoretical coverage analyzed. For particular process areas, RUP can be considered a good approach for CMMI ML2 implementation.

**Keywords:** RUP, CMMI ML 2, Project Execution.

## 1 Introduction

World organizations are influenced and molded by reference models that rule their activity, size or organizational culture. Regarding software development organizations, reference models such as CMMI, SPICE (ISO/IEC 15504:1998), ISO/IEC 9000, RUP, PMBOK, BABOK, PSP, ISO/IEC 9126, SWEBOK [1-3], amongst many others rule their behaviour and work. Although these reference models act in many different perspectives and sub-fields, their main purpose is to enhance the quality of the developed software according to the final users' needs [4]. The software development organizations need to be aware that the concern with the final product (software) is not enough and that the development process must be involved in improving itself. Development process means all activities necessary for managing, developing, acquiring and maintaining software [5]. The teams must be able to

---

\* This work has been supported by FEDER through Programa Operacional Fatores de Competitividade – COMPETE and by Fundos Nacionais through FCT – Fundação para a Ciência e Tecnologia in the scope of the project: FCOMP-01-0124-FEDER-022674.



evaluate the quality of the process in order to promote the monitoring and thereby detecting deviations in advance.

Using reference models to assess software quality is nowadays not only a minimum requirement for an organization's survival [6] but also a business strategy [7]. The focus of our work is on two reference models for software development processes: RUP (Rational Unified Process) [7] and CMMI (Capability Maturity Model Integration) [8, 9]. RUP and CMMI are used for different purposes: the first is focused on the necessary activity flow and responsibilities, and the other guides and assesses the maturity of the process in use. RUP and CMMI have a common goal: improving software quality and increasing the customer satisfaction. Both can be used together: since they complement each other mutually.

The main purpose of this work is to discuss whether through the adoption of RUP for small projects [10] in the execution of software development projects it is possible to achieve CMMI ML2. Our goal is to understand the support that we can expect from RUP when executing software development projects in a CMMI-compliant perspective. In a previous publication [11] we have presented the CMMI-RUP mapping only for two process areas (PP and REQM) strictly for the context of the restricted effort of elaborating software project proposals (we did not consider the execution phase of the project). The detailed motivation analysis for this work can be found in [11] but generically we can state that we intend to perceive if RUP is a good reference model to support a software development team in getting an alignment with CMMI ML2.

In this paper, we present the detailed mapping of CMMI ML2 (without SAM) PAs into RUP tasks and activities for the execution of software development projects and the analysis of the theoretical RUP coverage that we can expect for each PA. The SAM PA is out of our study since this process area is not mandatory for most of the companies. Since our concern is the project execution, we focus our analysis on the RUP Inception, Elaboration and Construction phases. A case study illustrates the usefulness of our CMMI-RUP mapping, where we interpret the obtained results in terms of the teams' performance while executing one software project and comparing with the theoretical RUP coverage.

## 2 Related Work

In 1991, the Software Engineering Institute (SEI) created the Capability Maturity Model (CMM). It has evolved until the creation of CMMI in 2002 [8, 9, 12, 13], which is more engineering-focused than its antecessors. CMMI provides technical guidelines to achieve a certain level of process development quality, however, it cannot determine how to attain such a level [2]. In November 2010, SEI released the CMMI-DEV v1.3 [9]. An appraisal at ML2 guarantees that the organization's processes are performed and managed according to the plan [6].

Rational Software developed in 1996 a software development framework called RUP. This framework includes activities, artifacts, roles and responsibilities and the best practices recognized for software projects. RUP enables the development team to perform an iterative transformation of the user's requirements into software that suits

the stakeholder's needs [2, 7, 14]. RUP also provides guidelines for “what”, “who” and “when” [2], avoiding an ad-hoc approach [7] that is usually time consuming and costly [6]. RUP divides the life cycle of a development process in four phases: *Inception, Elaboration, Construction and Transition*.

CMMI and RUP intersect each other in regards to software quality and hence customer satisfaction. In addition, both models have been constantly updated, so they do not become obsolete [7] and prevent an ad-hoc and chaotic software development environment [12]. While created by independent entities, they both counted with the participation of experts from the software industry and government [12]. There are many reasons why organizations should use these two frameworks: increased quality, productivity, customer and partners satisfaction; lower costs and time consumed; and better team communication [2, 6, 12]. CMMI-DEV may be used to evaluate an organization's maturity whether it uses or not RUP as a process model.

### 3 ML2 Mappings for Project Execution

Since our goal is to understand what kind of support can we expect from RUP to the execution of software development projects in a CMMI-compliant perspective, we will detail the previous analysis [11] for all the CMMI ML2 process areas at the subpractices level. When performing the previous analysis, we considered five different coverage levels that we will use in this study: *High coverage (H)*: CMMI fully implemented with RUP, which means that there are no substantial weaknesses; *Medium-High coverage (MH)*: CMMI nearly fully implemented with RUP, although some weaknesses can be identified; *Medium coverage (M)*: CMMI mostly implemented with RUP, however additional effort is needed to fully implement this process area using RUP; *Low coverage (L)*: CMMI is not directly supported using RUP, or there is a minimal RUP support; *Not covered (N)*: CMMI is not covered by any RUP.

We consider two different contexts for the execution of software projects: the context #1, where the development team must fully comply with CMMI recommendations, which means the team needs to perform all the subpractices; the context #2, where the development team acts with a strong time or cost bounds constrictions, which means the team may not be able to perform all the subpractices. Teams framed in the context #2 should only perform in what we have called P1 priority subpractices. P1 (higher priority) subpractices are considered mandatory for all the software projects execution (see last column of Table 2). Teams framed in the context #1 should perform P1, P2 and P3 priority subpractices. P3 (lower priority) and P2 (medium priority) subpractices may be skipped, when there is lack of information or of metrics to be thoroughly covered in the project execution. P3 subpractices are the first to be skipped; P2 subpractices may also be skipped in a second analysis.

Taking into account the dependencies analysis published and analyzed in [15] between CMMI ML2 PAs and SPs (see Table 1), we have decided to classify these SPs (all subpractices) with priority P1. Table 1 shows that PP SP1.1, SP1.2, SP1.4, SP2.1, SP2.2, SP2.3, SP2.4, SP2.5, SP2.6, SP3.1 and SP3.2 have priority P1.

**Table 1.** Dependencies Analysis between PAs and Specific Practices based on [15]

Process Area	Depended Process Area	Specific Practice related to
PP	MA CM	SP1.1
	PMC MA PPQA	SP1.2
	MA	SP1.4
	PMC	SP2.1
	PMC	SP2.2
	PMC	SP2.3
	PMC PPQA	SP2.4
	PMC	SP2.5
	PMC	SP2.6
	REQM	SP3.1
	PMC	SP3.2
PMC	REQM MA CM	SP1.1
	REQM	SP1.2
	PP	SP1.3
	MA	SP2.1
	REQM CM	SP2.2
MA	PMC	SP1.2
	PMC	SP1.3
	PMC	SP1.4
CM	PMC MA	SP1.2
	REQM	SP2.1
	REQM	SP2.2
REQM	PP	SP1.3
	MA	SP1.4

The detailed CMMI-RUP for the *Project Planning* PA contains the same required RUP tasks or activities to support each subpractice for the context of elaborating project proposals [11]. However, for the context of projects execution the subpractices priority has been changed. The priority of SP2.3, SP2.4, SP2.5 and SP3.2 for elaborating project proposals was P2 and now for the project execution is P1 since those SPs have dependencies between other ML2 PAs. In this PA, we do not consider P3 subpractices.

The *Requirements Management* (Table 2) PA mapping is quite similar to the previous mapping [11]. The difference is the reconsideration of the subpractices priorities. The priorities of SP1.3 and SP1.4 increased because these SPs have a dependency between the other ML2 PAs. The priority of subpractices SP1.1.3 and SP1.1.4 also increased because, in the project execution, the requirements analysis to ensure that the established criteria are met and the requirements understanding by requirements provider has more prominence that the case of elaborating project proposals.

We detail all the CMMI ML2 process areas (except SAM) like we did to the PP and REQM. For all the subpractices, we identify the tasks and activities, the coverage level of each mapping and the priority of each subpractice.

The *Measurements and Analysis* and the *Project and Monitoring Control* PA present high coverage. The *Measurements and Analysis* process area is composed by eight specific practices, six with high coverage, one (SP2.2) with medium-high coverage and one (SP2.3) with medium coverage. The SP2.2 presents medium-high coverage since the tasks *Monitor Project Status and Assess Iteration* do not guarantee the execution of an initial analysis of the measurements and accomplishment of preliminaries results. The SP2.3 presents medium coverage because none of the RUP

tasks covers the subpractices SP2.3.3 and SP2.3.4. RUP does not have elements that address the need to make the stored measurement data available only for appropriate groups and personnel, and to prevent the inappropriate use of the stored information. The subpractices presenting higher priority are the SP1.2, SP1.3 and SP1.4 subpractices since its priority is imposed by the dependencies between PAs (Table 1).

**Table 2.** Mapping Requirements Management PA for RUP Project Execution

		CMMI		RUP														
Requirements Management (Category: Engineering)	Process Areas	Specific Goals (SG)	Specific Practices (SP)	Subpractices	Task: Review Requirements	Task: Manage Dependencies	Task: Develop Requirements Management Plan	Task: Elicit Stakeholder Requests	Task: Detail the Software Requirements	Task: Project Planning Review	Task: Conduct Review	Task: Submit Change Request	Task: Develop Vision	Task: Review Change Requests	Activity: Manage Change Requests	Activity: Manage Changing Requirements	Priority	
SG1	SP1.1 MH		1				M						M				P1	
			2			H											P1	
			3	H	H			H	H									P1
			4	H	H													P1
	SP1.2 H		1		H	H												P1
			2	H	H	H												P2
	SP1.3 H		1						H							H	H	P1
			2															P1
			3		H										H	H	H	P1
			4					H									H	P1
	SP1.4 H		1		H													P1
			2		H													P1
			3		H													P1
	SP1.5 MH		1	H						H	H							P2
			2	L														P2
			3	H	H													P2
			4									H						

H High coverage, MH High-Medium Coverage, M Medium coverage, L Low Coverage, Not Covered, P1 Priority 1, P2 Priority 2

The *Project Monitoring and Control* is composed by two specific goals. SG1 is composed by seven specific practices, four with high coverage and three with medium-high coverage (SP1.1, SP1.4 and SP1.5). Specific Practice 1.1 is composed by six subpractices, four of them with high coverage. Subpractice SP1.1.2 presents medium coverage because task *Develop Measurement Plan* do not demand the inclusion of the project's cost and expended effort in the project metrics. The other subpractice (SP1.1.5) presents no RUP coverage because RUP do not monitors the skills of the team members. The SP1.4 presents medium-high coverage because two of its three subpractices (SP1.4.1 and SP1.4.3) are not fully implemented with RUP elements. With RUP, we cannot guarantee the review of the data management activities. SP1.5 presents also medium-high coverage because two of its three subpractices (SP1.5.1 and SP1.5.3) are partially implemented with RUP elements. *Task Report Status* does not ensure the review of the stakeholder involvement. The second goal has three specific practices, all with high coverage. The subpractices with higher priority are the subpractices of SP1.1, SP1.2, SP1.3, SP2.1 and SP2.2. These

subpractices should be performed even if the team has some constrictions because they monitor the main performance issues of the project. Furthermore, the dependencies between process areas also impose P1 priority for these specific practices.

The remaining ML2 Process Areas are *Process and Product Quality Assurance* and *Configuration Management*, and presents medium-high RUP compliance. *Process and Product Quality Assurance* is composed by two specific goals, each one with two specific practices. SG1 has one specific practice (SP1.1) with medium-high coverage and the other (SP1.2) with high coverage. The SP1.1 is not fully implemented with RUP because its tasks (in particular the task *Assess Iteration*) do not ensure the noncompliance identification and tracking and the lessons learned identification. The SG2 comprise two specific practices presenting medium coverage. SP2.1 presents medium coverage because RUP tasks do not address how we can ensure the resolution of noncompliance issues. The medium coverage of SP2.2 is triggered by the lack of RUP tasks that guarantee the storage and maintenance of the quality assurance results. This process area does not have priorities imposed by the dependencies between process areas. This process area has dependencies from other ML2 process areas, but the other process areas do not have dependencies from *Process and Product Quality Assurance*. The last process area is the *Configuration Management*. This process area has three specific goals. The first specific goal is divided into three specific practices: SP1.1, SP1.2 and SP1.3 (presenting high coverage). SP1.1 and SP1.2 present medium-high coverage because RUP tasks do not rigorously define the configuration items, components, and related work products that should be maintained under configuration management. RUP does not have mechanisms to create configuration management reports and to guarantee the storage, update, and retrieve of configuration management records. The specific goal 2 and 3 comprise two specific practices each: SP2.1 and SP3.2 with high coverage, and SP2.2 and SP3.1 with low coverage. The low coverage compliance of those specific practices is a consequence of the absence of RUP tasks that guarantee the control changes of the configuration items and the establishment and maintenance of configuration management records describing the configuration items.

#### 4 RUP Coverage Analysis for CMMI ML2 PAs

In this section, we describe the analysis of coverage that each PA can achieve with the adoption of RUP in the execution of software development projects. This study starts with the identification of all RUP tasks and activities mapped into each subpractice of the PA under evaluation. Next, we verify the coverage level for each subpractice. We convert the coverage levels defined in [11] into numeric values: H coverage: between 76% and 100%, by default, H coverage is 100% (this is the ideal coverage); MH coverage: between 51% and 75%, by default MH coverage is 75%; M coverage: between 25% and 50%, by default M coverage is 50%; L coverage: between 1% and 25%, by default L coverage is 25%; Not covered: the coverage is 0%.

We must verify in which RUP phase (*Inception, Elaboration and Construction*) each task is performed. Then, we determine the subpractice coverage in each RUP phase, by calculating the average of the tasks' coverage. We must take into consideration that RUP tasks and activities are not performed in all the RUP phases; some tasks are performed only in the *Inception*, some tasks only in the *Elaboration*, other tasks are performed only in two phases, and so on.

**Table 3.** Elementary Examples of RUP Coverage Analysis for Project Planning PA

Process Areas	CMMI				RUP Elements	RUP Phases			Coverage			Priority	
	Specific Goals (SG)	Weight	Specific Practices (SP)	Subpractices		Inception	Elaboration	Construction	Inception	Elaboration	Construction		
Project Planning Category: Project	SG1	4	SP1.1 H	1	1	Task: Develop Iteration Plan	x	x	x	100	100	100	P1
						Task: Identify and Assess Risks	x	x	x				
						Task: Plan Phases and Iterations	x	x					
			4	1	Task: Plan Phases and Iterations	x	x		67	100	100	P1	
					Task: Architectural Analysis	x	x						
					Task: Incorporate Existing Design Elements		x						
	3	SP1.4 MH	1	1	Task: Plan Phases and Iterations	x	x		50	50	50	P1	

As elementary examples, we describe how we calculate the RUP coverage of *Project Planning* SP1.1.1, SP1.1.4 and SP1.4.1 (see Table 3). Subpractice SP1.1.1 is mapped into the tasks *Develop Iteration Plan, Identify and Assess Risks* and *Plan Phases and Iterations*. The coverage level for this subpractice is high. Hence, these tasks present RUP coverage of 100%. Then, we verify in which RUP phase these tasks are performed. The tasks *Develop Iteration Plan* and *Identify and Assess Risks* are performed in all RUP phases we are considering. The task *Plan Phases and Iterations* is only performed in the *Inception* and *Elaboration* phases. Therefore, this subpractice presents RUP coverage of 100% in the three RUP phases under evaluation. The RUP coverage is the same in all phases because the tasks mapped into this subpractice are all performed for the first time in the same RUP phase (the *Inception*). The subpractice SP1.1.4 is mapped into the tasks *Plan Phases and Iterations, Architectural Analysis* and *Incorporate Existing Design Elements*. The coverage of this subpractice is high. The tasks *Plan Phases and Iterations* and *Architectural Analysis* are performed in the *Inception* and *Elaboration* phases, and the task *Incorporate Existing Design Elements* is only performed in *Elaboration* phase. This subpractice is mapped into three tasks, but two of them are performed in the *Inception* phase, which means we consider a RUP coverage of 67%. In the *Elaboration* and *Construction* phases, we obtain a RUP coverage of 100% because the other task is performed in the *Elaboration* with high coverage. The *Construction* phase is referred here because it inherits the coverage of the previous phases that have accomplished the RUP guidelines. The last example is the subpractice SP1.4.1. This subpractice is mapped into the task *Plan Phases and Iterations*. This task is performed in the *Inception* and *Elaboration* phase. This subpractice presents medium coverage, so the maximum coverage of this subpractice is 50%. Since the task is performed for the first time in the *Inception*, we consider RUP coverage of 50% for all RUP phases.

After assessing the RUP coverage for each subpractice, we calculate the RUP coverage for each Specific Goal, Specific Practice and PA.

We adopt a weighted average to calculate the RUP coverage of each specific goal, specific practice and PA. The subpractices weight is based in the priority level. Higher priority (P1) subpractices correspond to a weight of 1, medium priority subpractices correspond to a weight of 0,5 ( $P2\_weight=P1\_weight/2$ ) and lower priority subpractices correspond to a weight of 0,33 ( $P3\_weight=P1\_weight/3$ ). The SP weight is defined as the sum of its subpractices weight. We calculate two types of PAs coverage, one only with P1 subpractices and the other with all the subpractices.

The RUP coverage for *Project Planning* PA (see Table 4) considering P1 subpractices is 84% in the *Inception* and 90% in the *Elaboration* and *Construction* phases. The RUP coverage, with all *Project Planning* subpractices is 83% in the *Inception* and 89% in the *Elaboration* and *Construction* phases.

The RUP coverage for *Requirements Management* PA (see Table 4) considering P1 subpractices is 57% in the *Inception*, medium-high coverage. In the other two phases, it presents high coverage, achieving 96%. The RUP coverage, with all *Requirements Management* subpractices decreases around 3%. In the *Inception*, the RUP coverage is 53% and in the *Elaboration* and *Construction* is 94%.

Table 4 presents a summary of the RUP coverage for *Project Planning*, *Requirements Management*, *Process and Product Quality Assurance*, *Project and Monitoring Control*, *Measurements and Analysis* and *Configuration Management* process areas.

**Table 4.** Summary of the RUP Coverage for all PAs

Process Areas	CMMI	Coverage		
	Average	Inception	Elaboration	Construction
PP	SG1 Average	88	91	91
	SG2 Average	82	89	89
	SG3 Average	79	86	86
	Average with P1	84	90	90
	Average with P1 + P2	83	89	89
REQM	SG1 Average	53	94	94
	Average with P1	57	96	96
	Average with P1 + P2	53	94	94
CM	SG1 Average	9	74	74
	SG2 Average	3	56	56
	SG3 Average	2	29	29
	Average with P1	6	62	62
	Average with P1 + P2 + P3	6	59	59
MA	SG1 Average	96	96	96
	SG2 Average	65	70	70
	Average with P1	100	100	100
	Average with P1 + P2 + P3	90	91	91
PPQA	SG1 Average	68	83	83
	SG2 Average	26	52	52
	Average with P1	67	83	83
	Average with P1+P2+P3	55	73	73
PMC	SG1 Average	83	88	88
	SG2 Average	100	100	100
	Average with P1	91	91	91
	Average with P1 + P2 + P3	87	91	91

The RUP coverage for *Process and Product Quality Assurance* PA considering P1 subpractices is 67% in the *Inception*, medium-high coverage. In the other two phases, it presents high coverage, achieving 83%. The RUP coverage, with all *Process and Product Quality Assurance* is 55% in the *Inception* and 73% in the *Elaboration* and *Construction* phases.

The RUP coverage for *Project and Monitoring Control* PA considering P1 subpractices is 91% in the *Inception*, *Elaboration* and *Construction* phases. The RUP coverage, with all subpractices is 87% in the *Inception* and 91% in the *Elaboration* and *Construction* phases.

The *Measurements and Analysis* PA considering P1 subpractices achieves the ideal RUP coverage, 100%, in all RUP phases. The RUP coverage with all *Measurements and Analysis* subpractices decreases around 10%, achieving 90% in the *Inception* and 91% in the *Elaboration* and *Construction* phases.

The *Configuration Management* is the process area that presents the lowest RUP coverage. The RUP coverage for *Configuration Management* PA considering P1 subpractices is 6% in the *Inception*, phase and 62% in the *Elaboration* and *Construction* phases. The RUP coverage, with all *Configuration Management* subpractices is 6% in the *Inception* and 59% in the *Elaboration* and *Construction* phases.

## 5 Case Study

A case study was developed to assess the usefulness of the CMMI-RUP mapping to support the execution of CMMI ML2 Process Areas in the context of software development projects execution. This case study was performed at an educational environment and adopted the guidelines established in [16].

The case study involved one hundred and eleven students enrolled in the 8604N5 Software System Development (SSD) from the undergraduate degree in Information Systems and Technology in University of Minho (the first University to offer in Portugal DEng, MSc and PhD degrees in Computing). Students were divided in seven development software teams, each one receiving a sequential identification number (Team 1, Team 2, ..., Team 7). The teams had between 13 and 17 people (1 team with 13, 1 team with 15, 2 teams with 16 and 3 with 17). These teams have to produce a web application that meets the requirements of a real end customer. The teams have some constraints: two interactions with the client; using RUP (only the first three RUP phases); follow CMMI ML2 guidelines; and eighteen weeks for development. All teams have attended a previous course where they were exposed to the RUP concepts. One team (Team 1) was randomly chosen to not adopting the RUP (we call this team the "Control Team") while the other six teams followed the guidelines established by the RUP, executing the phases of *inception*, *elaboration* and *construction*. The control team did not follow any kind of guidelines for organizing themselves in term of roles/responsibilities/team organization.

The students have four assessment milestones of execution and evaluation: Assessment Milestone 1 (M1): relates to the initial project planning, which is part of



the Inception Phase; Assessment Milestone 2 (M2): relates to the Inception Phase; Assessment Milestone 3 (M3): relates to the Elaboration Phase; Assessment Milestone 4 (M4): relates to the Construction Phase.

The assessment of the teams' performance adopted the following two steps: (1) Documental analysis of the produced artifacts to detect compliance with the subpractices of the ML2 process areas. SAM process area is out of the project scope; (2) Elaboration of a survey at the end of each assessment milestone, to check the status of the teams and the team members perception of CMMI practices.

For each team, we calculate the coverage level observed for each subpractice, the corresponding average for each specific practice of CMMI ML2 process areas and the process area average. The coverage level was converted into numeric values: high coverage (H) corresponds to 100%; medium-high coverage (MH) corresponds to 75%; medium coverage (M) corresponds to 50%; low coverage (L) corresponds to 25%, and no coverage (N) correspond to 0%.

The specific practice and process area coverage was calculated using the weighted average. The subpractices weight was based in the level of priority: P1 subpractices correspond to a weight of 1, P2 subpractices correspond to a weight of 0,5 (P2\_weight=P1\_weight/2) and P3 subpractices correspond to a weight of 0,33 (P3\_weight=P1\_weight/3). The specific practice weight was defined as the sum of its subpractices weight.

**Table 5. Teams results for Requirements Management Process Area**

Process Areas	CMMI			Team 1				Team 2				Team 3				Team 4				Team 5				Team 6				Team 7				Priority					
	Specific Practices (SP)	SP Weight	Subpractices Subpractice Weight	M1	M2	M3	M4	M1	M2	M3	M4	M1	M2	M3	M4	M1	M2	M3	M4	M1	M2	M3	M4	M1	M2	M3	M4	M1	M2	M3	M4						
Requirements Management (Category: Engineering)	SP1.1 H	4,00	1	1		M	M	M	M	M	M	M	M	M	M	M	H	H	H	H	MH	H	H	H	H	H	H	H	H	H	H	H	H	H	P1		
			2	1	M	M	M	M		M	M	MH		H	H	H	MH	H	H	H		M	M	M	M	H	H	H	M		H	H	H	H	P1		
			3	1																																P1	
			4	1		L	L	L	L	MH	MH	MH	M	MH	MH	MH	MH	MH	MH	MH	MH	MH	MH	MH	MH	MH	MH	MH	MH	M	M	MH	MH	MH	P1		
	<b>SP1.1 Average</b>			<b>13</b>	<b>31</b>	<b>31</b>	<b>31</b>	<b>19</b>	<b>44</b>	<b>44</b>	<b>56</b>	<b>25</b>	<b>69</b>	<b>69</b>	<b>69</b>	<b>69</b>	<b>69</b>	<b>69</b>	<b>69</b>	<b>69</b>	<b>0</b>	<b>56</b>	<b>56</b>	<b>56</b>	<b>69</b>	<b>69</b>	<b>69</b>	<b>56</b>	<b>31</b>	<b>63</b>	<b>69</b>	<b>69</b>					
	SP1.2 H	1,50	2,05	1	1		L	L	L		MH	MH	MH		MH	MH	MH				MH	MH	MH					M	M	L	H	H	H	P1			
				2	0,5		L	L	L		MH	MH	MH		MH	MH	MH		MH	MH	MH		MH	MH	MH					MH	MH	M	MH	MH	MH	P2	
				3	1																																P1
				4	1	M	MH	MH	MH	M	H	H	H	M	H	H	H	M	H	H	H	H	H	M	MH	MH	MH	MH	M	M	M	M	M	M	M	H	H
	<b>SP1.2 Average</b>			<b>0</b>	<b>25</b>	<b>25</b>	<b>25</b>	<b>0</b>	<b>25</b>	<b>25</b>	<b>75</b>	<b>0</b>	<b>92</b>	<b>92</b>	<b>0</b>	<b>92</b>	<b>92</b>	<b>92</b>	<b>0</b>	<b>25</b>	<b>25</b>	<b>42</b>	<b>0</b>	<b>25</b>	<b>58</b>	<b>33</b>	<b>0</b>	<b>92</b>	<b>92</b>	<b>92</b>							
	SP1.3 H	4,00		1	1		M	MH	MH		MH	MH	MH		MH	MH	MH				MH	MH	MH					M	M	MH	MH	MH	MH	H	P1		
				2	1		L	L	L		MH	MH	MH		MH	MH	MH		MH	MH	MH		MH	MH	MH					L	MH	M	M	H	H	P1	
				3	1		L	L	L		MH	MH	MH		MH	MH	MH		MH	MH	MH		MH	MH	MH					M	M	M	M	M	M	H	P1
				4	1	M	MH	MH	MH	M	H	H	H	M	H	H	H	M	H	H	H	H	H	M	MH	MH	MH	MH	M	M	M	M	M	M	M	H	H
	<b>SP1.3 Average</b>			<b>13</b>	<b>31</b>	<b>50</b>	<b>63</b>	<b>43</b>	<b>44</b>	<b>56</b>	<b>69</b>	<b>13</b>	<b>44</b>	<b>69</b>	<b>81</b>	<b>43</b>	<b>44</b>	<b>69</b>	<b>88</b>	<b>0</b>	<b>44</b>	<b>69</b>	<b>75</b>	<b>13</b>	<b>38</b>	<b>56</b>	<b>75</b>	<b>13</b>	<b>44</b>	<b>69</b>	<b>100</b>						
	SP1.4 H	3,00		1	1					M	M	MH		MH	MH	H	H	H				H	H	H					M	L	M	H	H	P1			
				2	1		L	L	L		MH	MH	MH		MH	MH	MH		MH	MH	MH		MH	MH	MH					M	L	MH	MH	MH	P1		
				3	1		L	L	L		MH	MH	MH		MH	MH	MH		MH	MH	MH		MH	MH	MH					M	L	MH	MH	MH	P1		
				4	1																																P2
	<b>SP1.4 Average</b>			<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>33</b>	<b>33</b>	<b>58</b>	<b>0</b>	<b>17</b>	<b>42</b>	<b>100</b>	<b>0</b>	<b>100</b>	<b>100</b>	<b>100</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>25</b>	<b>0</b>	<b>0</b>	<b>50</b>	<b>25</b>	<b>0</b>	<b>50</b>	<b>83</b>	<b>100</b>						
	SP1.5 H	2,00		1	0,5																														P2		
				2	0,5																															P2	
				3	0,5																																P2
				4	0,5																																P2
<b>SP1.5 Average</b>			<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>				
<b>Average with P1</b>			<b>8</b>	<b>23</b>	<b>29</b>	<b>33</b>	<b>10</b>	<b>38</b>	<b>42</b>	<b>63</b>	<b>13</b>	<b>50</b>	<b>65</b>	<b>83</b>	<b>23</b>	<b>71</b>	<b>79</b>	<b>85</b>	<b>0</b>	<b>33</b>	<b>42</b>	<b>52</b>	<b>27</b>	<b>35</b>	<b>58</b>	<b>52</b>	<b>15</b>	<b>56</b>	<b>75</b>	<b>90</b>	<b>45</b>						
<b>Average with P1+P2</b>			<b>7</b>	<b>20</b>	<b>25</b>	<b>28</b>	<b>9</b>	<b>34</b>	<b>37</b>	<b>54</b>	<b>10</b>	<b>44</b>	<b>56</b>	<b>72</b>	<b>19</b>	<b>61</b>	<b>68</b>	<b>73</b>	<b>0</b>	<b>30</b>	<b>37</b>	<b>46</b>	<b>22</b>	<b>32</b>	<b>51</b>	<b>45</b>	<b>12</b>	<b>49</b>	<b>65</b>	<b>77</b>	<b>39</b>						

As an example of the detailed results obtained in the case study, we present the Table 5 with the teams' results for Requirements Management PA. A similar effort was made for all the teams' results for the CMMI ML2 process areas.

Table 6 presents a summary of the results obtained after the assessment of the RUP coverage for each process areas, in each assessment milestone.

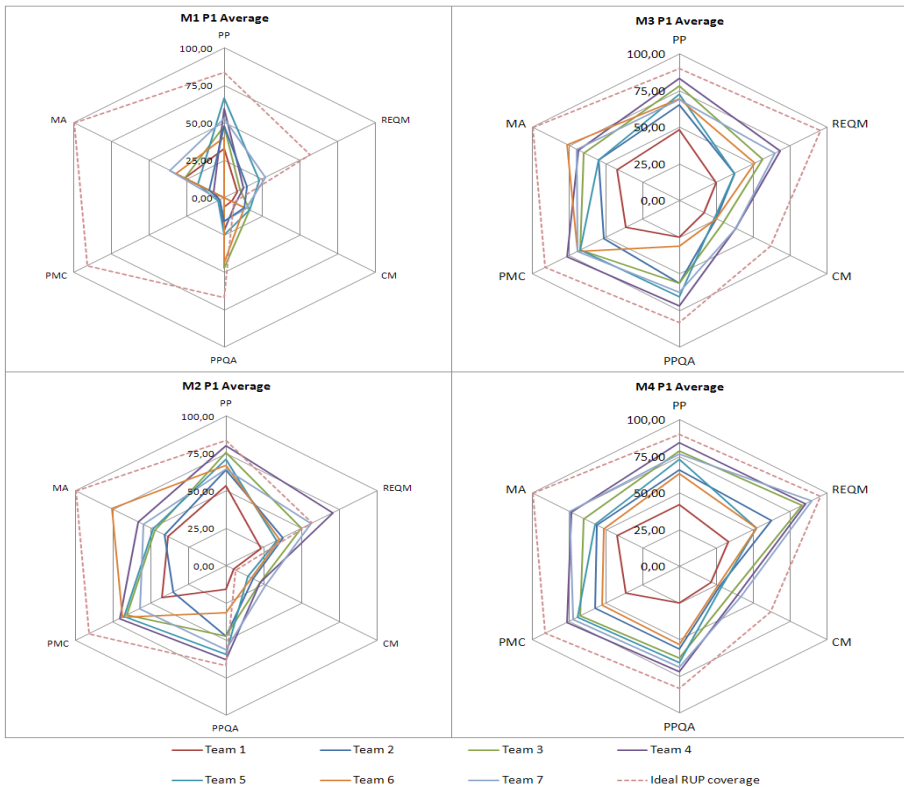
**Table 6.** Teams Assessment

		Team 1	Team 2	Team 3	Team 4	Team 5	Team 6	Team 7	Ideal Coverage
<b>PP (P1 Coverage)</b>									
Inception	M1	33	48	59	67	40	53	48	84
	M2	54	64	76	80	71	67	65	
Elaboration	M3	49	65	78	84	73	69	69	90
Construction	M4	42	66	79	85	73	63	77	90
<b>PP (P1+P2+P3 Coverage)</b>									
Inception	M1	39	59	71	77	50	62	59	83
	M2	59	66	76	85	71	72	72	
Elaboration	M3	53	68	79	86	74	74	74	89
Construction	M4	48	68	79	84	74	63	79	89
<b>REQM (P1 Coverage)</b>									
Inception	M1	8	10	13	23	0	27	15	57
	M2	23	38	50	71	33	35	56	
Elaboration	M3	25	37	56	68	37	51	65	96
Construction	M4	33	63	83	85	52	90	96	96
<b>REQM (P1+P2+P3 Coverage)</b>									
Inception	M1	7	9	10	19	0	22	12	53
	M2	20	34	44	61	30	32	49	
Elaboration	M3	25	37	56	68	37	51	65	94
Construction	M4	28	54	72	73	46	45	77	94
<b>CM (P1 Coverage)</b>									
Inception	M1	5	17	7	17	13	14	13	6
	M2	5	18	23	23	14	17	26	
Elaboration	M3	17	25	30	38	24	25	38	62
Construction	M4	21	27	36	39	29	26	42	62
<b>CM (P1+P2+P3 Coverage)</b>									
Inception	M1	4	18	6	20	16	15	14	6
	M2	5	18	24	25	17	18	28	
Elaboration	M3	16	24	29	37	27	25	38	59
Construction	M4	20	26	34	38	30	28	40	59
<b>PPQA (P1 Coverage)</b>									
Inception	M1	6	47	22	25	44	16	16	67
	M2	16	47	47	63	59	31	56	
Elaboration	M3	25	56	56	72	66	31	63	83
Construction	M4	25	56	63	72	66	53	69	83
<b>PPQA (P1+P2+P3 Coverage)</b>									
Inception	M1	4	31	17	17	30	11	12	55
	M2	15	38	38	49	47	23	45	
Elaboration	M3	22	44	47	59	53	27	52	73
Construction	M4	22	47	52	59	53	42	57	73
<b>PMC (P1 Coverage)</b>									
Inception	M1	0	0	3	4	0	3	3	91
	M2	43	35	66	71	68	69	57	
Elaboration	M3	37	51	68	76	68	69	69	91
Construction	M4	37	57	68	76	69	53	72	91
<b>PMC (P1+P2+P3 Coverage)</b>									
Inception	M1	0	0	3	4	1	7	3	87
	M2	44	34	75	79	74	77	64	
Elaboration	M3	41	58	80	88	75	79	78	91
Construction	M4	41	68	80	88	80	60	82	91
<b>MA (P1 Coverage)</b>									
Inception	M1	26	26	8	18	33	36	10	100
	M2	39	41	48	59	49	76	55	
Elaboration	M3	43	55	65	69	55	76	70	100
Construction	M4	43	56	65	74	58	51	75	100
<b>MA (P1+P2+P3 Coverage)</b>									
Inception	M1	23	24	8	17	29	38	11	90
	M2	33	37	44	55	46	70	51	
Elaboration	M3	38	51	61	63	51	70	63	91
Construction	M4	38	53	61	68	54	47	68	91

In the project planning (M1, Fig. 1), the RUP coverage is much higher than the assessed teams coverage. The main reason for this discrepancy is explained by the fact that, in M1, the teams are only concerned in the project planning execution and

the theoretical RUP coverage for this assessment milestone is the same of the *Inception* phase coverage. Looking to the teams' coverage, we can see that all of them (except the control team) achieve coverage higher 40% when we assess only the P1 subpractices.

Looking to Fig. 1 and Table 6, we can see the huge difference between the *Project Planning* process area and the other ML2 process areas as well as the difference between the P1 average and all subpractices average. The teams' *Project Planning* process area presents higher coverage when we assess all subpractices than it has when we assess only the P1 subpractices. Since in this assessment milestone, the teams are focused in the project planning elaboration they try to implement all subpractices. In M1, two teams (Team 2 and Team 5) have also spent some effort in the implementation of *Process and Product Quality Assurance* (P1 average of 47% and 44%, respectively). However, this effort let them undervalue the *Project Planning* implementation.



**Fig. 1.** CMMI ML2 Assessment (considering P1 subpractices)

At the end of M2 (Fig. 1 and Table 6), the teams' coverage has increased when compared with M1. The coverage of all process areas has increased and approached the theoretical RUP coverage. To emphasize the Team 4, that in the *Project Planning* and *Requirements Management* process areas even exceeded the theoretical coverage

(85% for all *Project Planning* subpractices, 61% for *Requirements Management P1* subpractices and 71% for all *Requirements Management* subpractices). This was a consequence of the teams' constraint of follow CMML ML2 guidelines. They anticipate some of the guidelines that will be implemented by RUP in a subsequent phase. The teams' coverage of *Configurations Management* was also higher than the theoretical RUP coverage, also because the CMMI ML2 constraint. The theoretical RUP coverage is very low in the *Inception* phase because RUP tasks that cover the *Configurations Management* subpractices are performed only in the *Elaboration* and *Construction* phase.

In M3, the theoretical coverage is the maximum coverage that we can achieve if we adopt RUP to implement CMMI. Almost all CMMI ML2 process areas have a theoretical coverage level higher than 75% for both average types. The *Process and Product Quality Assurance* do not achieve a high level for very little it has 73% coverage (for all subpractices average). The *Configuration Management* is the process area with the lowest coverage level; it has a coverage level of 59% for all subpractices and 62% for P1 subpractices. In this phase, we can see that the results of the control team are become quite different from the other teams and considerably lower than the theoretical RUP coverage. Team 4 was the assessed team that achieved the highest level of coverage. This team has achieved the highest level for all process areas except for *Measurement and Analysis*.

In the last assessment milestone, the results are quite similar to the M3 results. There are some slight coverage improvements in the assessed process areas for all teams. The control team performance as we expected is the weakest of all teams. This team had more difficulty in implement CMMI ML2 process areas since they do not follow RUP, and consequently do not have a predefined set of tasks that will help in know how to implement CMMI.

**Table 7.** Summary of RUP Coverage for CMMI ML2

Process Areas	CMMI Average	Coverage					
		Inception		Elaboration		Construction	
		Theoretical Coverage	Teams Coverage Average	Theoretical Coverage	Teams Coverage Average	Theoretical Coverage	Teams Coverage Average
PP	Average with P1	84	70	90	73	90	74
	Average with P1 + P2	83	74	89	76	89	75
REQM	Average with P1	57	47	96	52	96	71
	Average with P1 + P2	53	42	94	52	94	61
CM	Average with P1	6	20	62	30	62	33
	Average with P1 + P2 + P3	6	22	59	30	59	33
MA	Average with P1	100	55	100	65	100	63
	Average with P1 + P2 + P3	90	51	91	60	91	58
PPQA	Average with P1	67	51	83	57	83	63
	Average with P1+P2+P3	55	40	73	47	73	52
PMC	Average with P1	91	61	91	67	91	66
	Average with P1 + P2 + P3	87	67	91	76	91	76

## 6 Conclusion

Customer satisfaction is the most expected outcome by the software development companies. CMMI and RUP intersect in regards to software quality and, therefore, customer satisfaction.

In this study, we have identified the RUP elements that fulfill CMMI ML2 process areas (without SAM) in the context of software projects execution. We have also analyzed the RUP coverage that we can achieve for each CMMI ML2 PA.

With a case study, we have assessed the accomplishment of several teams of the CMMI-RUP mapping. We have found out that the teams adopting RUP have a higher compliance with CMMI ML2 than the control team that did not follow RUP (Table 6). In Table 7, we can compare the theoretical coverage of each CMMI ML2 PA and the teams' coverage average (without the control team).

Fig. 2 presents the comparison of the theoretical RUP coverage and the average of the real results obtained by the teams (without the control team). We can compare the evolution of the teams' coverage average with the theoretical RUP coverage, throughout the RUP phases. We can also compare the teams' average with the theoretical RUP coverage looking only to P1 subpractices and looking to all subpractices, throughout the RUP phases for each CMMI ML2 PA. The theoretical RUP coverage achieves the maximum coverage in the *Elaboration* phase, but the teams' average achieves the maximum coverage only in the *Construction* phase. The teams performance reach almost the previously theoretical estimated coverage, but with some temporal delay; i.e., while the theoretical pick coverage is possible during the Elaboration phase, the real pick coverage is observed during the Construction phase for almost the teams.

As future work, we will detail the CMMI-RUP mapping to the CMMI ML3 process areas.

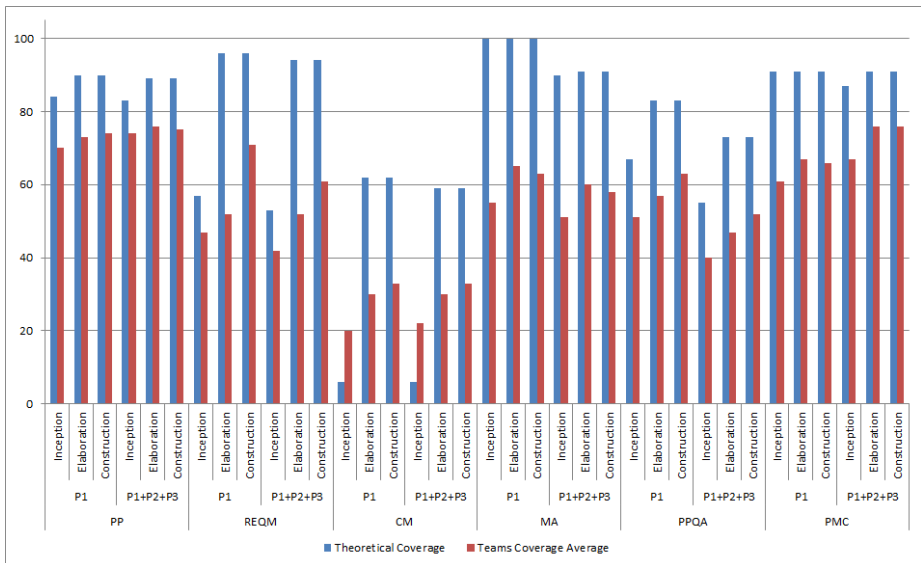


Fig. 2. Comparison between Ideal and Teams Coverage Average by PA and Priority

## References

1. Niazi, M., Wilson, D., Zowghi, D.: Critical success factors for software process improvement implementation: An empirical study. *SPIP* 11, 193–211 (2006)
2. Manzoni, L.V., Price, R.T.: Identifying extensions required by RUP to comply with CMM levels 2 and 3. *IEEE TSE* 29, 181–192 (2003)
3. Marchewka, J.T.: *Information technology project management*. John Wiley and Sons (2009)
4. Chen, C.-Y., Chong, P.P.: Software engineering education: A study on conducting collaborative senior project development. *Journal of Systems and Software* 84, 479–491 (2011)
5. Melo, W.: Enhancing RUP for CMMI compliance: A methodological approach. *The Rational Edge*. IBM (2004)
6. Carvalho, J.P., Franch, X., Quer, C.: Supporting CMMI Level 2 SAM PA with Non-technical Features Catalogues. *SPIP* 13, 171–182 (2008)
7. Kruchten, P.: *The Rational Unified Process: An Introduction*. Addison-Wesley (2003)
8. CMMI Product Team: *CMMI for Development, Version 1.2 (CMU/SEI-2006-TR-008)* (2006)
9. CMMI Product Team: *CMMI for Development, Version 1.3 (CMU/SEI-2010-TR-033)* (2010)
10. IBM, RUP for small projects, version 7.1, <http://www.wthreex.com/rup/smallprojects/> (accessed April 12, 2012)
11. Monteiro, P., Machado, R.J., Kazman, R., Lima, A., Simões, C., Ribeiro, P.: Mapping CMMI and RUP Process Frameworks for the Context of Elaborating Software Project Proposals. In: Winkler, D., Biffel, S., Bergsmann, J. (eds.) *SWQD 2013*. LNBP, vol. 133, pp. 191–214. Springer, Heidelberg (2013)
12. Ahern, D.M., Clouse, A., Turner, R.: *CMMI Distilled: A Practical introduction to Integrated Process Improvement*. Addison - Wesley (2004)
13. Chrissis, M.B., Konrad, M., Shrum, S.: *CMMI: Guidelines for Process Integration and Product Improvement*. Addison-Wesley (2006)
14. IBM, Rational Unified Process: Best practices for software development teams, [http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251\\_bestpractices\\_TP026B.pdf](http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf) (accessed August 29, 2013)
15. Monteiro, P., Machado, R.J., Kazman, R., Henriques, C.: Dependency Analysis between CMMI Process Areas. In: Ali Babar, M., Vierimaa, M., Oivo, M. (eds.) *PROFES 2010*. LNCS, vol. 6156, pp. 263–275. Springer, Heidelberg (2010)
16. Runeson, P., Host, M.: Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 131–164 (2009)

# Directing High-Performing Software Teams: Proposal of a Capability-Based Assessment Instrument Approach

Petri Kettunen

University of Helsinki  
Department of Computer Science  
P.O. Box 68, FI-00014 University of Helsinki, Finland  
petri.kettunen@cs.helsinki.fi

**Abstract.** It is not clearly understood, what high performance means for software development enterprises. Product development team performance has been investigated extensively in various industries, but software development teams and their knowledge-intensive work are still open to even fundamental questions and gaps. Software team performance is relative to the particular context. The performance outcomes of the teams are products of their specific capabilities, provided by the underlying software competencies. This paper proposes a high-performing software team capability analysis approach supported by provisional instrumentation. The goal of such an analyzer is to facilitate software teams and organizations to identify their essential capabilities and – in case of mismatches or weaknesses – to gauge the development of necessary ones. An industrial team case demonstrates how it is able to capture and characterize different team capability traits for performance analysis and improvement with respect to the overall aims of the software organization.

**Keywords:** software teams, performance management, capability development, process improvement, high-performing organization.

## 1 Introduction

Modern high-performing software organizations rely increasingly on capable teams. It follows that the organizational development should focus on their team capabilities. Furthermore, new teams can be configured based on the required key capabilities. Moreover, not just having teams but consciously concentrating on their performance is what brings the business benefits. Such fundamental comprehension would greatly help to leverage high-performing teams to scale up even at enterprise levels.

High-performing teamwork has been investigated in many fields over the years. In particular, the success factors of new product development (NPD) teams are in general relatively well known [1]. However, the specific concerns and intrinsic properties of modern software development teams are essentially less understood in particular in larger scales [2]. It is well known that there may be tremendous productivity differences between different software teams. However, it is not clearly understood, what high performance means for software development enterprises in total, and how exactly such effects and outcomes are achievable in predictable ways.

Our overall research question is thus as follows: How can defined (high) team performance be attained in the particular software development context? This paper tackles that with a design science approach by proposing a holistic capability analysis frame for team-based software organizations. The capabilities are evaluated with our previously developed Monitor instrument (team self-assessment) [3], [4]. Based on that information, we produce the current capability profile of the team with the Analyzer instrument constructed here.

The rest of this paper is organized as follows. The next Section 2 reviews software team performance in general and capability-oriented development views in particular. Section 3 then presents the capability-based team performance analysis approach, followed by a case example in Section 4. Finally, Section 5 discusses the proposition with implications and pointers to further work concluding in Section 6.

## 2 Software Team Performance and Capabilities

Successful software organizations rely more and more on teamwork (Sect. 2.1). However, in order to be able to form and develop such capable software teams (Sect. 2.3), appropriate performance measures and gauging must be set in the context (Sect. 2.2). That raises newfangled research and development opportunities for team performance analysis and improvement (Sect. 2.4).

### 2.1 Software Development Teams in Organizations

Industrial-strength software product development is almost always done in teams, even in globally virtual setups [5]. The increasing demand of software delivered fast on the one hand, and growing complexity of software products and systems on the other hand imposes higher and higher performance requirements for the software development teams. Moreover, considering the lack of available software development resources with respect to the needs, there is a pressing need to be able to form and sustain high-performing teams and to improve their performance further. However, although product development team performance has been investigated extensively in various industries, software development teams and their knowledge-intensive work are still open to even fundamental questions and gaps.

Software teams do not exist in isolation in particular in larger product development enterprises. The context of the team affects the team organization and their performance in several, sometimes subtle and even conflicting ways. It is thus fundamental to understand such factors, too. Moreover, in large-scale organizations there are typically not just single teams but a network of interdependent teams, passing even external company boundaries [6]. That is, the team context is multidimensional and often dynamic. In addition to the context, no two teams are in practice equal inside since teams consist of individual persons with different skills, competencies, and personalities.

In general, it is not reasonable to attempt to define (high) performance of software teams without taking into account the context [2]. The organizational context is especially important for agile software teams, since they are by nature outward-oriented (e.g., close customer-cooperation). Most notably, agility shows on the organizational boundaries of the team (e.g., responsiveness).



## 2.2 Software Team Performance

In general, there is no one universal measure of software team performance. To begin with, software teams can be seen as general work teams and their performance accordingly like dimensioned for instance by Hackman [7]. Typically software team performance is associated with productivity [8]. However, software development teams have usually multiple enterprise stakeholders – including the team members themselves – and consequently multiple different dimensions of performance [9]. Multivariate measures are thus usually more applicable [10]. Different teams may have different performance targets even within same larger organizations [11].

Prior literature has described many such possible software team performance measures [12], [2]. Typical measures used in software-intensive organizations can be categorized as objective measures (e.g., function points, use case points, defect rates) / subjective measures (ratings by team members and external stakeholders, e.g., perceived end product quality, teamwork satisfaction), and product performance (e.g., user-perceived functionality, reliability, maintainability) / process performance measures (e.g., development schedule, budget).

High-performing product teams strive for developing the right things (products/services providing optimal value), and getting them released well at the right time (effective and disciplined delivery) [13]. In particular, high-performing agile teams are sensitive to their environment, flexible, and responsive to changing customer needs [14]. Software team performance in terms of value creation efficiency can be assessed with value stream mapping/analysis (VSM/A) and more generally value network analysis (VNA) methods [15]. High performance can then be defined in terms of optimal value creation (benefits vs. costs) [13], [16]. Performance in terms of agility can be measured with multiple different scales [17].

Although it is difficult to define general-purpose performance metrics for specific software teams, the measurement systems can be developed based on existing general-purpose frameworks to begin with [18]. Notably, software teams may sometimes perceive their internal performance differently than the team externally observable outcomes exhibit. Moreover, the perceived value of the product may vary between different stakeholders. It is imperative to know, who judges the success and when (e.g., development cost vs. profits, purchase value vs. use value) [19]. Finally, although financial performance measures are still the most obvious ones in industrial enterprise teamwork, recently additional dimensions have been proposed – such as triple bottom line covering also social and environmental elements [20].

## 2.3 Team Capabilities

The resource-based view (RBV) is a well-known approach for organizational development [21]. Our work presented in this paper builds on those grounds in general. More specifically, we are analyzing high-performing teams in terms of their capabilities. The premise here is that there should be a conscious fit between the specific performance needs and the capabilities of the team.

In general, the term capability is used in various ways in extant organizational development and management literature. Here we take the basic stance that they are qualities, abilities, and features that can be used and developed (potential). More

systematically, the following definition (by replacing ‘person’ with ‘team’) is congruent [22]:

- *Capabilities describe the skills and abilities, aptitudes and attitudes needed by a person to achieve high performance in a specific role.*

A closely related term is competence. Sometimes they are used interchangeably. However, in this work, we consider competencies as components and building blocks of capabilities [23]. This line of thinking meets also for instance the traits of the Performance Prism framework, which considers capabilities as combinations of people, practices, technology, and infrastructures enabling to perform business processes – such as product development [24]. Software-related competences (e.g., business and engineering) have been categorized in various different ways [25].

In general, just having right competencies is not enough to make a capable team [26]. For instance high-performing agile software teams have certain distinct capabilities with respect to performance [27]. One of the key capabilities is the ability to perform fast development cycles with frequent customer feedback.

## 2.4 Capability Evaluation and Development

Software organization capabilities are typically seen from the process-oriented viewpoint. For instance, the Capability Maturity Model (CMM) models define capable processes as ones that “can satisfy its specified product quality, service quality, and process performance objectives” [28]. In a similar vein, the Software Process Improvement and Capability Determination (SPICE) model gauges software organization capabilities to deliver software products [29]. By and large, the basic purpose of those process-oriented capability models is to evaluate software suppliers with respect to predefined process dimensions and their expected practices. In addition to such specific models, there are also many general-purpose organizational development frameworks with supporting assessments instruments (e.g., [30], [31], [32]).

At the software team level, the Team Software Process (TSP) is one of the most well-established performance development approaches [33]. The TSP has subsequently been coupled with the Capability Maturity Model Integration for Development (CMMI-DEV) [34].

In particular agile software teams do by definition self-reflective continuous capability evaluation and improvement [35], [36]. Although agile software team development has often been seen contradictory to the organizational capability models (chiefly the CMM models) there are certain current attempts to bridge such gaps [37].

The aim of this investigation is to build contextual and situation-aware understanding of the key capability elements of high-performing software development teams. The design rationale is to cover broadly all key areas to gauge them systematically. Such awareness will make it possible to build future navigation aids for team performance development.

### 3 Instrument Design and Analysis Principles

In order to address the research objectives reviewed in Sect. 2, we propose a capability-based monitoring and analysis approach for high-performing software teams (Sect. 3.1). This approach is supported by provisional instrumentation with a team self-assessment Monitor and an accompanying Analyzer (Sect. 3.2).

#### 3.1 Approach

This paper extends our previous works. The Monitor instrument was initially developed to profile different characteristics of high-performing software teams [3]. The capability-oriented approach was introduced with the Agile capability [4]. This paper continues developing the approach and building the instrumentation for the other recognized capabilities.

The overall standpoint of our team analysis approach is that, for each particular software team, there is a performance ideal in its specific organizational context (desired state). The current state of the team may deviate from that for various reasons. The objective is then to understand the current position of the team and the capabilities to be developed and improved in order direct the team to reach the desired state (gap analysis).

The research method and design are as follows. High performance requires capable teams. However, different teams may have different performance targets, and emphasize different key capabilities (e.g., in fast-moving customer goods vs. industrial automation systems). Once the team recognizes its particular capability needs and weights, it can self-assess them and improve the identified gaps.

Our connection between capabilities and (high) team performance is as follows:

- Ideally, each team performs to the best of its capabilities. The team and its management should know them.
- However, the actual realization of the capabilities may be incomplete and possibly hindered by impediments. The team should recognize them.

In order to realize that line of thinking, we apply design science to construct actionable artifacts. The attributes of the capabilities are measured by the Monitor (team self-assessment). Based on that information, we produce the current capability profile of the team with the Analyzer (Sect. 3.2). We can then discuss that together with the team, whether the team have sufficient and fit capabilities for the desired (high) performance, which capabilities should be directed to be improved in the future, and what potential obstacles and impediments should be removed in order to get the full benefits of the capabilities.

The Monitor-Analyzer instrumentation presented here covers currently six typical capability areas. The Monitor items are mainly based on software team performance management research literature. Typically such investigations (reductionism) study certain different factors and their performance effects with correlation hypothesis. Our work is based on combining such factors under the capability profiles. The profiles thus give suggestions, how the team may potentially perform with respect to the different capability traits.

Currently the analysis comprises the following specific capabilities: Agile, Lean, Business Excellence, Operational Excellence, Growth, Innovativeness. They have been selected based on general-purpose organizational performance development models to begin with (e.g., [24], [30], [31]). We have then coupled them with software team performance management literature (Sect. 2.2), in particular with respect to Agile and Lean capabilities. Table 1 presents their overall rationales. However, we do not claim that these are all key capabilities for any particular team. Nevertheless, considering generalization, we presume this initial set to be a plausible starting point.

**Table 1.** Capability-based performance reasoning

CAPABILITY	Performance Traits
Agile	<ul style="list-style-type: none"> <li>• conscious sensitivity and responsiveness to customer and environment needs and changes</li> </ul>
Lean	<ul style="list-style-type: none"> <li>• continuous system-wide value-orientation</li> </ul>
Business Excellence	<ul style="list-style-type: none"> <li>• effectiveness, result-orientation</li> <li>• systemic benefits</li> </ul>
Operational Excellence	<ul style="list-style-type: none"> <li>• sustainable efficiency</li> <li>• consistent predictability</li> </ul>
Growth	<ul style="list-style-type: none"> <li>• active learning and improvement</li> <li>• advancement (including assets, capital)</li> </ul>
Innovativeness	<ul style="list-style-type: none"> <li>• creative exploration and exploitation</li> <li>• foresight</li> </ul>

This work does not propose any particular quantitative formulas for determining the level of the team capabilities (such as capability index). Instead, we rely on the expert judgment of the team itself supported by visual plotting of systematized item combinations as sourced in the team self-assessment (Monitor). The suggested heuristic reasoning is as follows: If the indicator items associated with a particular capability appear to be positive, the current level of the team with respect of that capability may be high. Conversely, if there are some negative signs and/or large variations between the individual team member ratings of the items, the level of capability may be lower.

### 3.2 Analyzer

Our earlier investigations have sensed high-performing software teams with a self-assessment Monitor instrument [3]. The Monitor instrument captures a wide set of team performance attributes. By selecting and combining distinct subsets of them, we can produce capability profile views of the team. This is the design rationale of the Analyzer instrument [4].

The Analyzer aggregates certain subsets of the Monitor questionnaire items and recombines them for the selected team capability indicators. Certain items are coupled to multiple capabilities. Table 2 presents a subset of the current constituting indicator items (currently 6) for each included capability (c.f., Table 1) with their underlying rationales.

**Table 2.** Capability indicator items (partial)

ITEMS	Rationale
<b>Agile</b>	
How do you rate the following organizational factors in your context? <ul style="list-style-type: none"> <li>• <i>The organization is flexible and responsive to customer needs.</i></li> </ul>	The whole organization is encouraged to think in customer-oriented ways. The mindset is towards leveraging the organizational strengths and capital (“can do”).
How do you rate the following concerns? <ul style="list-style-type: none"> <li>• <i>Our team is capable of quick round-trip software engineering cycles (design-build-test-learn).</i></li> </ul>	Iterative development and consequently responsiveness are enabled. Continuous (fast) delivery of valuable software is realized.
<b>Lean</b>	
How do you rate the following aspects from your point of view? <ul style="list-style-type: none"> <li>• <i>We see how our products bring benefits (value).</i></li> </ul>	The importance and usefulness of the product to customer/user needs and problems are prompted.
How do you appraise the following team outcomes and impacts? <ul style="list-style-type: none"> <li>• <i>Outputs meet the organizational standards and expectations.</i></li> </ul>	The software implementation is consistently assessed against defined quality criteria (e.g., reliability, response time).
<b>Business Excellence</b>	
How do you rate the following concerns? <ul style="list-style-type: none"> <li>• <i>How often are you able to see the software (product) in actual use?</i></li> </ul>	The actual business and use context and how the user operations utilize the software execution results are observed in real time. The value and fitness of the product solution to its purpose and concept of operations is assessed.
How do you appraise the following team outcomes and impacts? <ul style="list-style-type: none"> <li>• <i>Getting the business benefits (value)</i></li> </ul>	The value creation and capture drives the software development. The business mindset is incorporated to the software teams.
<b>Operational Excellence</b>	
How important are the following for your team? <ul style="list-style-type: none"> <li>• <i>Getting the products done well (effective and disciplined delivery)</i></li> </ul>	The software production is results-driven. The activities are aligned towards the delivery targets.
How do you rate the following concerns? <ul style="list-style-type: none"> <li>• <i>Our team is fully integrated with the surrounding organization.</i></li> </ul>	The software development is clearly positioned in the total value stream and able to implement its role in the flow.
<b>Growth</b>	
How do you rate the following aspects from your point of view? <ul style="list-style-type: none"> <li>• <i>We have a clear, compelling direction that energizes, orients the attention, and engages our full talents.</i></li> </ul>	Finding (the) most significant and meaningful problems and opportunities to be solved with software is emphasized.
How important are the following for your team? <ul style="list-style-type: none"> <li>• <i>The software (design) is easily upgradable and flexible for future development.</i></li> </ul>	The customer space uncertainties and opportunities are taken into account in design decisions and preparations (e.g., architectural choices). Technical debt is avoided.

**Table 2.** (Continued.)

<b>Innovativeness</b>	
How important are the following for your team? • <i>Thinking the total product / service / system</i>	The role of the software in the product solutions is comprehended. The positioning of the products with the customer/user systems is motivated.
How important are the following aspects for you in your work? • <i>Developing the particular product or service (innovation)</i>	The purpose of the product drives the software development.

In addition, Table 3 shows a subset of potential impediment items which typically prevent from achieving full performance with the capabilities in Table 2.

**Table 3.** Capability demotivator items (partial)

<b>ITEMS</b>	<b>Rationale</b>
How do you rate the following concerns? • <i>How often are there communication and coordination breakdowns?</i>	Gaps and glitches tend to cause delays, inefficient knowledge-sharing, and distracted decisions hurting all software capabilities.
How do you rate the following organizational factors in your context? • <i>People have time to "think" (no excessive stress, pressures).</i>	Sustainable complex software work requires certain slack time. Rushing does not foster excellence in software capabilities.

We have a tool-assisted implementation (spreadsheet) of the Analyzer with visual plotting for evaluating the individual capabilities. For each indicative item, the distributions of the selected Monitor question items are shown with graphs. The tool provides such a view for each individual capability (e.g., Agile) like shown in Figure 1.

## 4 Case Example

We have been using the Monitor instrument regularly since the initial establishment [3]. This section illustrates a case for applying the Analyzer with the detailed items of the current Monitor realization.

We have executed several team performance investigations with various industrial software development organizations. There is one such case team included here. They have applied the Monitor in two rounds in 2012-2013. Considering their key demographic information, the team develops integrated system components in a medium-size global company. Most of the team members (including some subcontractors) were experienced and the team had worked together for longer time. The system has a long life-span.

Figure 1 presents the subset of the Monitor data of the case team as viewed by the Analyzer. The organization of the table is as follows: The question blocks (6+1) are

the currently incorporated indicating items of the defined capabilities in Table 2 and the demotivators in Table 3. They have been extracted from the Monitor. The data shows the number of responses of the team self-ratings like described in our initial publication of the Monitor [3]. The responses were anonymous. Note that some of the team members chose not to respond to all question items.

In essence, the Analyzer views tabulated in Figure 1 exhibit, how the case team perceived certain key aspects contributing to their capabilities with respect to Table 1. We can now reflect the different capabilities as follows (italics as in Tables 2-3):

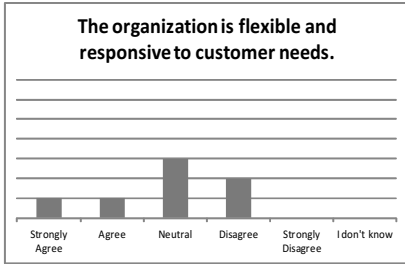
- Agile: For the system component team, the customer appears to be more distant. They do not especially see the organization to be '*flexible and responsive to changing customer needs*'. However, they do have the core ability to conduct '*quick round-trip software engineering cycles*' for responsiveness.
- Lean: The team understands how the product creates '*benefits (value)*'. In addition, they know that their work should fulfill the '*organizational standards*'.
- Business Excellence: As a component team, not everybody can observe the '*software (product) in actual use*'. This might be an impediment for customer-orientation. Nevertheless, they do appreciate the '*business benefits*'.
- Operational Excellence: Being a part of a larger of the system product, the team may not see its role in the total '*delivery*' chain quite clearly. They appear to be not so strongly '*integrated*', which may cause for instance delays in their teamwork.
- Growth: A component team may have trouble having a clear '*direction*'. This could be discouraging, and prevent them from seeing future growth opportunities. In addition, the current software architecture may not readily support '*future development*'.
- Innovativeness: Not everybody in the team is equally interested in '*developing the particular product*'. This may make them less amenable for pursuit of new ideas for the product. However, they do have the mindset towards considering the '*total product*'.
- DE: The team appears to have problems with '*communication and coordination*'. However, they do not seem to be under excessive '*time*' pressure, so that may not be explained by hurry.

When we conducted such reflective discussions together with the team, one consecutive improvement action was to reorganize the team into two smaller subteams in order to mitigate the perceived communication and coordination problems (see the 'DE' part). The perceptions of the reorganized team indicate that this change was favorable (see Figure 1). The response rate was not 100%, though.

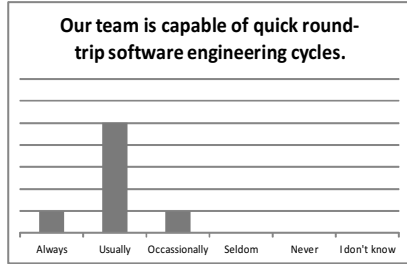
In conclusion, the purpose of our approach and instrumentation is neither to measure the team's performance nor to give normative means to achieve high ends. Instead, like illustrated in this one case example, the general idea is to highlight key performance influencing factors and potential impediments for the team. However, it is for the team itself to judge and rank them for improvement actions. Similar general tactics have been proposed elsewhere [36].

**Agile**

How do you rate the following organizational factors in your context?

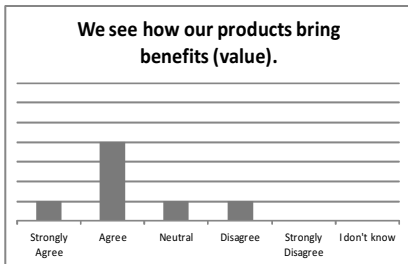


How do you rate the following concerns?



**Lean**

How do you rate the following aspects from your point of view?

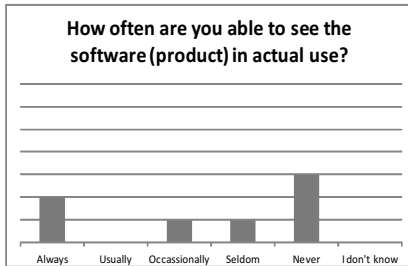


How do you appraise the following team outcomes and impacts?

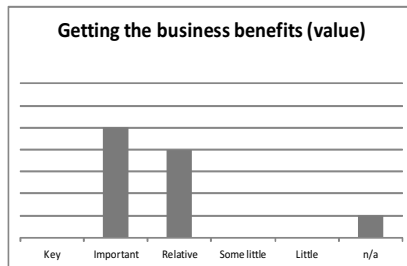


**Business Excellence**

How do you rate the following concerns?



How do you appraise the following team outcomes and impacts?



**Operational Excellence**

How important are the following for your team?



How do you rate the following concerns?

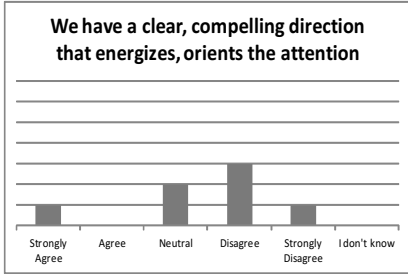


**Fig. 1.** Analyzer views of the case team monitor data

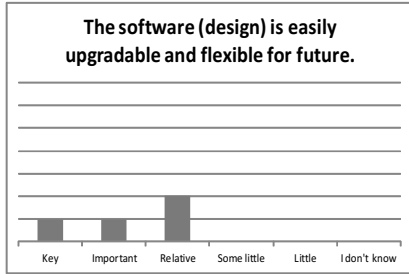


**Growth**

How do you rate the following aspects from your point of view?

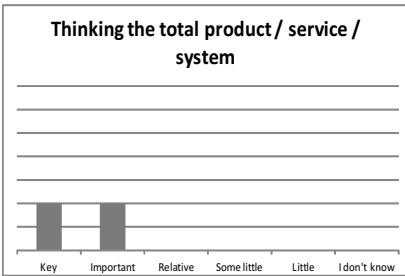


How important are the following for your team? <sup>1)</sup>

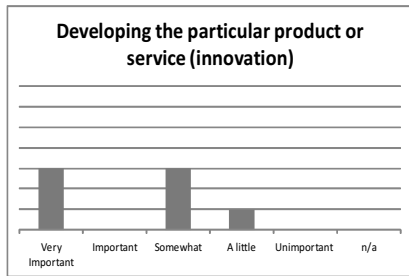


**Innovativeness**

How important are the following for your team? <sup>1)</sup>

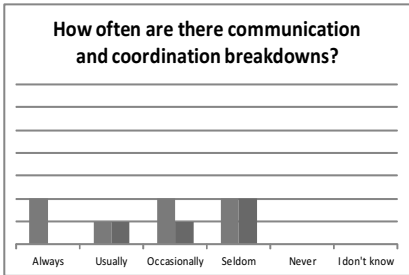


How important are the following aspects for you in your work?



**DE**

How do you rate the following concerns? <sup>2)</sup>



How do you rate the following organizational factors in your context?



1 Responses of the second query round

2 Responses of the first and second query rounds

**Fig. 1. (Continued.)**

**5 Discussion**

The Monitor-Analyzer is primarily a diagnostic instrument. It does not advocate direct solutions or particular software practices. The general design idea is to illuminate the important capability areas for specific software teams and organizations to work on.

In its current stage of development, the Monitor-Analyzer has not been validated for prediction [38]. For instance, with respect to the Agile capability, the Analyzer view based on the team Monitor instrument self-rating information merely suggests that the team may perform high in terms of agility [4]. However, that is not measured here. Specific performance measures could then be some of the ones summarized in Sect. 2.2. As of this writing, we do not have such measurement data readily available.

The validation can be improved by asking the team about their level of agreement on their perceived capability profile indicated by the Analyzer. This would also provide some triangulation of the survey results.

Furthermore, the self-assessment could be repeated longitudinally following the triggered improvement actions to see their performance effects. The team itself can thereby keep building its own capability profile view over time. Moreover, if the Analyzer view indicates that the team itself perceives to have some weaknesses in its current capabilities, there is a risk of lower performance. Such considerations should then be taken into account when anticipating (if not predicting) the teams' future performance. The key is that the team itself recognizes its own level of capability with respect to the expectations.

The team case presented in Sect. 4 exemplifies those considerations. Like illustrated there, the Analyzer views should be evaluated in conjunction with the actual context and situational factors of the team. In general, deeper analysis based on the survey information only is not recommended.

Like recognized already in our initial works with the Monitor, the team self-rating survey has certain inherent limitations and constraints such as lack of common terminology, trust, honesty, self-assessment biases, and survey method limitations [3]. However, most of those limitations and risks can be mitigated by face-to-face discussions with the team members (e.g., clarifying potential misunderstandings). In reality, such reflective dialogue is anyway required to be able to engage the team and stimulate their performance improvements.

All things considered, based on the limited, primarily survey-based case evidence, we are not yet in a position to draw firm conclusions about the generalization of our results. However, the main idea of our approach and instrumentation is to be valid for the particular teams and organizations, and the local validity is for them to judge.

Overall, we see the following prospective thread for further research and development of the approach and instrumentation:

- By conducting more case studies with different teams, the actual expressive strength of the selected Monitor items could be weighted more systematically. This would also strengthen the validation.
- Following that, the current configuration of the Analyzer can be evaluated further with respect to the indicating items (currently 6, but could vary) of each capability descriptor. In addition, potential new capability views could be considered (e.g., Flexibility, Resilience). This is in particular if the specific performance needs of the team are not fully followed by the currently included capabilities. Moreover, potential linkages between the different elementary items within the capabilities and combining capabilities (bundling) such as Agile & Lean would deserve further investigations.

The next advancement would be to construct a Navigator instrument based on the current Monitor-Analyzer constellation with the following line of thinking. The stakeholders of the specific software team shall be identified along with their performance expectations. The Analyzer as presented in this paper provides a profile of the software team's current capabilities. The capabilities are then compared and developed against the aligned expectations of the stakeholders. Both the stakeholder expectations and the associated capability views of the Analyzer should eventually be gauged with appropriate measurements.

For example, in our case team organization (Sect. 4), consistent product performance is the key customer satisfaction factor. Consequently, this would put weight on the Lean and also Operational Excellence capabilities of the software team.

## 6 Conclusion

This paper tackles the research question of how specific software team performance can be directed in the organizational context. We have presented a capability-based software team performance assessment and improvement approach. The approach is supported by provisional design scientific Monitor-Analyzer instrumentation.

The key research design principle of our team Monitor-Analyzer approach has been not to limit to any one particular discipline (e.g., computer science). Instead, we take a holistic view of software teams consisting of individuals in their organizational and business contexts.

This work contributes primarily for practitioners. The Monitor-Analyzer instruments are readily available (as prototype tools). In addition, it promotes potential topical research directions for team performance management given that software teams even in traditional organizations work increasingly in new set-ups (e.g., offshoring) and radically new ways of teamwork are emerging in creative organizations (e.g., game companies). Moreover, the Monitor-Analyzer can be used as research instruments for action research to trigger more theoretical research questions.

By and large, our Monitor-Analyzer approach strives for addressing the following strategic issues in the software organization [39]: What is the intended performance (success) for the team / organization? What are the key capabilities needed for the success? What are the individual competences and organizational characteristics needed to support the capabilities? Like illustrated in the case example, with such understanding the software organization can gauge its teams for achieving the overall (business) goals of the organization. Moreover, the organization can direct its development activities according to the capabilities of the software teams with such profound understanding of its team-based strengths in the competitive environment.

**Acknowledgements.** This work was supported by Cloud Software Program which is funded by DIGILE (former TIVIT, Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT) and TEKES.

## References

1. Cooper, R.G., Edgett, S.J.: *Lean, Rapid, and Profitable New Product Development*. BookSurge Publishing, North Charleston (2005)
2. McLeod, L., MacDonnell, S.G.: Factors that Affect Software Systems Development Project Outcomes: A Survey of Research. *ACM Computing Surveys* 43(4) (2011)
3. Kettunen, P., Moilanen, S.: Sensing High-Performing Software Teams: Proposal of an Instrument for Self-monitoring. In: Wohlin, C. (ed.) *XP 2012. LNBP*, vol. 111, pp. 77–92. Springer, Heidelberg (2012)
4. Kettunen, P.: The Many Facets of High-Performing Software Teams: A Capability-Based Analysis Approach. In: McCaffery, F., O'Connor, R.V., Messnarz, R. (eds.) *EuroSPI 2013. CCIS*, vol. 364, pp. 131–142. Springer, Heidelberg (2013)
5. Kleinschmidt, E., de Brentani, U., Salomo, S.: Information Processing and Firm-Internal Environment Contingencies: Performance Impact on Global New Product Development. *Creativity and Innovation Management* 19(3), 200–218 (2010)
6. Kettunen, P.: *Agile Software Development in Large-Scale New Product Development Organization: Team-Level Perspective*. In: Dissertation. Helsinki University of Technology, Finland (2009)
7. Hackman, J.R.: *Leading Teams: Setting the Stage for Great Performances*. Harvard Business School Press, Boston (2002)
8. Petersen, K.: Measuring and predicting software productivity: A systematic map and review. *Information and Software Technology* 53, 317–343 (2011)
9. Chenhall, R.H., Langfield-Smith, K.: Multiple Perspectives of Performance Measures. *European Management Journal* 25(4), 266–282 (2007)
10. Stensrud, E., Myrtveit, I.: Identifying High Performance ERP Projects. *IEEE Trans. Software Engineering* 29(5), 398–416 (2003)
11. Berlin, J.M., Carlström, E.D., Sandberg, H.S.: Models of teamwork: ideal or not? A critical study of theoretical team models. *Team Performance Management* 18(5/6), 328–340 (2012)
12. Kasunic, M.: *A Data Specification for Software Project Performance Measures: Results of a Collaboration on Performance Measurement*. Technical report TR-012, CMU/SEI (2008)
13. Winter, M., Szczepanek, T.: Projects and programmes as value creation processes: A new perspective and some practical implications. *International Journal of Project Management* 26, 95–103 (2008)
14. Ancona, D., Bresman, H.: *X-Teams: How to Build Teams that Lead, Innovate, and Succeed*. Harvard Business School Press, Boston (2007)
15. Allee, V.: Value Network Analysis and value conversion of tangible and intangible assets. *Journal of Intellectual Capital* 9(1), 5–24 (2008)
16. Buschmann, F.: Value-Focused System Quality. *IEEE Software* 27(6), 84–86 (2010)
17. Anderson, D.J.: *Agile Management for Software Engineering*. Prentice Hall, Upper Saddle River (2004)
18. Staron, M., Meding, W., Karlsson, G.: Developing measurement systems: an industrial case study. *J. Softw. Maint. Evol.: Res. Pract.* 23, 89–107 (2010)
19. Agresti, W.W.: Lightweight Software Metrics: The P10 Framework, pp. 12–16. *IT Pro* (September–October 2006)
20. Osterwalder, A., Pigneur, Y.: *Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers*. John Wiley & Sons, New York (2010)

21. Tonini, A.C., Medina, J., Fleury, A.L., de Mesquita Spinola, M.: Software Development Strategic Management: A Resource-Based View Approach. In: Proc. PICMET, pp. 1072–1080 (2009)
22. Professional Staff Core Capability Dictionary. University of Adelaide, Australia (2010)
23. Day, G.S.: The Capabilities of Market-Driven Organizations. *Journal of Marketing* 58, 37–52 (1994)
24. Neely, A., Adams, C., Crowe, P.: The performance prism in practice. *Measuring Business Excellence* 5(2), 6–13 (2001)
25. von Hertzen, M., Laine, J., Kangasharju, S., Timonen, J., Santala, M.: Drive For Future Software Leverage: The Role, Importance, and Future Challenges of Software Competences in Finland. *Review* 262. Tekes, Helsinki (2009)
26. Downey, J.: Designing Job Descriptions for Software Development. In: Barry, C., et al. (eds.) *Information Systems Development: Challenges in Practice, Theory, and Education*, vol. 1, pp. 447–460. Springer Science+Business Media (2009)
27. Conboy, K., Fitzgerald, B.: Toward a conceptual framework for agile methods: a study of agility in different disciplines. In: Mehandjiev, N., Brereton, P. (eds.) *Workshop on Interdisciplinary software engineering research (WISER)*, pp. 37–44. ACM, New York (2004)
28. CMMI for Development. Technical report, CMU/SEI-2010-TR-033. Software Engineering Institute, Carnegie Mellon University, USA (2010)
29. Guidance on use for process improvement and process capability determination. Information technology, Process assessment, Part 4: 15504-4, ISO/IEC (2009)
30. EFQM Excellence Model. EFQM Foundation, Belgium (2012)
31. Baldrige National Quality Program: Criteria for Performance Excellence. National Institute of Standards and Technology (NIST), Gaithersburg, MD (2012)
32. Drexler, A., Sibbet, D.: *Team Performance Model (TPModel)*. The Grove Consultants International, San Francisco (2004)
33. Humphrey, W.S.: *Introduction to the Team Software Process*. Addison Wesley Longman Inc., Reading (2000)
34. Humphrey, W.S., Chick, T.A., Nichols, W.R., Pomeroy-Huff, M.: *Team Software Process (TSP) Body of Knowledge (BOK)*. Technical report, CMU/SEI-2010-TR-020. Software Engineering Institute, Carnegie Mellon University, USA (2010)
35. Pikkarainen, M.: *Towards a Framework for Improving Software Development Process Mediated with CMMI Goals and Agile Practices*. Dissertation, University of Oulu, Finland (2008)
36. Moe, N.B., Dingsøy, T., Røyrvik, E.A.: Putting Agile Teamwork to the Test – An Preliminary Instrument for Empirically Assessing and Improving Agile Software Development. In: Abrahamsson, P., Marchesi, M., Maurer, F. (eds.) *XP 2009*. LNBP, vol. 31, pp. 114–123. Springer, Heidelberg (1975)
37. Glazer, H.: Love and Marriage: CMMI and Agile Need Each Other. *CrossTalk* 23(1), 29–34 (2010)
38. Fenton, N.E., Pfleeger, S.L.: *Software Metrics: A Rigorous & Practical Approach*. International Thompson Computer Press (1996)
39. Wirtenberg, J., Lipsky, D., Abrams, L., Conway, M., Slepian, J.: The Future of Organization Development: Enabling Sustainable Business Performance Through People. *Organization Development Journal* 25(2), 11–22 (2007)

# Author Index

- Abrahamsson, Pekka 100  
Adorf, Hans-Martin 199  
Angelis, Lefteris 63  
Arcuri, Andrea 185  
  
Bauer, Thomas 135  
Biff, Stefan 170  
Book, Matthias 115  
  
Chatzipetrou, Panagiota 63  
  
Elberzhager, Frank 135  
  
Fraser, Gordon 185  
  
Gencel, Cigdem 100  
Grapenthin, Simon 115  
Gruhn, Volker 115  
  
Kalchauer, Alexander 34  
Kalinowski, Marcos 12  
Karapiperis, Christos 63  
Kazman, Rick 214  
Kettunen, Petri 229  
  
Lang, Sandra 34  
  
Machado, Ricardo J. 214  
Mäkinen, Simo 155  
Marciuska, Sarunas 100  
Mendes, Emilia 12  
Monteiro, Paula 214  
Mordinyi, Richard 170  
Münch, Jürgen 155  
  
Palampouiki, Chrysa 63  
Peischl, Bernhard 34  
  
Ribeiro, Pedro 214  
Rodela Torrents, Vanesa 34  
Rosbach, Alla 135  
  
Schmid, Klaus 85  
Simões, Cláudia 214  
Sneed, Harry M. 48  
  
Thillen, François 170  
Travassos, Guilherme Horta 12  
  
Varendorff, Martin 199  
  
Wilhelm, Reinhard 1