Bertrand Meyer Jerzy R. Nawrocki
Bartosz Walter (Eds.)

Balancing Agility and Formalism in Software Engineering

Second IFIP TC 2 Central and East European Conference
on Software Engineering Techniques, CEE-SET 2007
Poznań, Poland, October 10-12, 2007
Revised Selected Papers

Springer

Volume Editors

Bertrand Meyer
ETH Zentrum, Department of Computer Science
Clausiusstr. 59, 8092 Zürich, Switzerland
E-mail: bertrand.meyer@inf.ethz.ch

Jerzy R. Nawrocki
Poznań University of Technology, Institute of Computing Science
ul. Piotrowo 2, 60-965 Poznań, Poland
E-mail: jerzy.nawrocki@put.poznan.pl

Bartosz Walter
Poznań University of Technology, Institute of Computing Science
ul. Piotrowo 2, 60-965 Poznań, Poland
E-mail: Bartosz.Walter@cs.put.poznan.pl

Library of Congress Control Number: 2008932464

CR Subject Classification (1998): D.2, K.6.3, K.6, K.4.3, F.3.2, H.2, J.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-85278-6 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-85278-0 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12446049 06/3180 5 4 3 2 1 0

Preface

The origins of CEE-SET go back to the end of the 1990s, when the Polish Information Processing Society together with other partners organized the Software Engineering Education Symposium, SEES 1998, sponsored by CEPIS, and the Polish Conference on Software Engineering, KKIO 1999 (the latter has become an annual event). A few years later KKIO changed to an international conference on Software Engineering Techniques, SET 2006, sponsored by Technical Committee 2 (Software: Theory and Practice) of the International Federation for Information Processing, IFIP [http://www.ifip.org/]. In 2007 the conference got a new name: second IFIP TC2 Central and East-European Conference on Software Engineering Techniques, CEE-SET 2007. It took place in Poznan, Poland, and lasted for three days, from October 10 to 12, 2007 (the details are on the conference web page http://www.cee-set.org/2007). The conference aim was to bring together software engineering researchers and practitioners, mainly from Central and East-European countries (but not only), and allow them to share their ideas and experience. The special topic for 2007 was "Balancing Agility and Formalism in Software Engineering."

The conference was technically sponsored by:

- IFIP Technical Committee 2, Software: Theory and Practice
- Gesellschaft für Informatik, Special Interest Group Software Engineering
- John von Neumann Computer Society (NJSZT), Hungary
- Lithuanian Computer Society
- Polish Academy of Sciences, Committee for Informatics
- Polish Information Processing Society
- Slovak Society for Computer Science

Financial support was provided by IBM Software Laboratory in Krakow, Microsoft Research, Microsoft Polska, Polish Information Processing Society, and the XPrince Consortium.

The conference program consisted of 3 keynote speeches given by Scott W. Ambler (IBM, Canada), Bertrand Meyer (ETH Zurich), and Dieter Rombach (Fraunhofer IESE, Kaiserslautern), 21 regular presentations selected from 73 submissions (success rate was about 28%), and 15 work-in-progress presentations. The International Program Committee nominated the following three regular papers for the Best Paper Award:

- Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi: "A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team"
- Adam Trendowicz, Michael Ochs, Axel Wickenkamp, Juergen Muench, Yasushi Ishigai, and Takashi Kawaguchi: "An Integrated Approach for Identifying Relevant Factors Influencing Software Development Productivity"

– Mesfin Mulugeta, and Alexander Schill: "A Framework for QoS Contract Negotiation in Component-Based Applications"

After the above paper presentations, the Best Paper Award Committee chaired by Miklos Biro (IFIP TC2 member) decided that the IFIP TC2 Manfred Paul Award should go to the authors of the second paper: A. Trendowicz, M. Ochs, A. Wickenkamp, J. Muench, Y. Ishigai and T. Kawaguchi.

The IFIP TC2 Manfred Paul Award is an annual award. "The award is made for a published paper, and consists of a prize of 1024 euros and a plaque or certificate. The award is named after Manfred Paul, who was chairman of TC2 from 1977 to 1986 and the representative for Germany from 1973." [http://www.ifip.or.at/awards.htm].

Another award at CEE-SET 2007 was the Best Presentation Award. Here the jury was the audience of the conference. Two presenters won the competition: Miroslaw Ochodek and Ramin Tavakoli Kolagari.

This volume contains two of the three keynotes and all the regular presentations given at the conference. We believe that publishing these high-quality papers will support a wider discussion on balancing agility and formalism in software development and, more generally, on software engineering techniques.

May 2008

Bertrand Meyer
Jerzy Nawrocki
Bartosz Walter

Organization

Program Chairs

Bertrand Meyer
Jerzy Nawrocki

Program Committee

Pekka Abrahamsson
Vincenzo Ambriola
Uwe Assmann
Hubert Baumeister
Maria Bielikova
Stefan Biffl
Miklos Biro
Pere Botella
Albertas Caplinskas
Radovan Cervenka
Paul Clements
Jutta Eckstein
Evram Eskenazi
Gabor Fazekas
Kurt Geihs
Janusz Górski
Paul Grünbacher
Nicolas Guelfi
Tibor Gyimothy
Heinrich Hussmann
Zbigniew Huzar
Stefan Jähnichen
Paul Klint
Jan Kollar
Laszlo Kozma
Henryk Krawczyk
Leszek Maciaszek
Jan Madey
Lech Madeyski
Zygmunt Mazur
Juergen Muench
Pavol Navrat
Barbara Paech
Andras Pataricza
Alexander K. Petrenko
Frantisek Plasil
Erhard Ploedereder
Klaus Pohl
Saulius Ragaisis
Felix Redmil
Karel Richta
Krzysztof Sacha
Wilhelm Schaefer
Helen Sharp
Giancarlo Succi
Tomasz Szmuc
Andrey Terekhov
Bartosz Walter
Jaroslav Zendulka
Krzysztof Zieliński

Local Organization

Michał Jasiński
Jan Kniat
Ewa Łukasik
Łukasz Macuda
Bartosz Michalik
Ewa Nawrocka
Miroslaw Ochodek
Łukasz Olek

Magdalena Olek
Joanna Radke
Bartosz Walter
Adam Wojciechowski

External Reviewers

Marc Alier
Richard Atterer
Alexei Barantsev
Jerzy Błaszczyński
Bartosz Bogacki
Alfredo Capozucca
Łukasz Cyra
Matthias Gehrke
Joel Greenyer
Timea Illes-Seifert
Janusz Jabłonowski
Agata Janowska
Olek Jarzębowicz
Michał Jasiński
Bartosz Michalik
Jakub Miler
Raimund Moser
Miroslaw Ochodek
Łukasz Olek
Maciej Piasecki
Błażej Pietrzak
Gergely Pintér
Andreas Pleuss
Benoît Ries
Zdzisław Spławski
Łukasz Szala
Dániel Tóth
Artur Wilczek
Dietmar Winkler
Adam Wojciechowski
Michael Zapf
Sergei Zelenov

Table of Contents

Agile Software Development at Scale

Scott W. Ambler

Practice Leader Agile Development, IBM Rational
scott_ambler@ca.ibm.com

Abstract. Since 2001 agile software development approaches are being adopted across a wide range of organizations and are now being applied at scale. There are eight factors to consider – team size, geographical distribution, entrenched culture, system complexity, legacy systems, regulatory compliance, organizational distribution, governance and enterprise focus – when scaling agile. Luckily a collection of techniques and strategies exist which scale agile approaches, including considering the full development lifecycle, Agile Model Driven Development (AMDD), continuous independent testing, adopting proven strategies, agile database techniques, and lean development governance. It is possible to scale agile approaches, but you will need to look beyond the advice provided by the "agile in the small" literature.

1 Introduction

Agile software development is being adopted by both the majority of organizations, a recent survey [1] shows that 69% of organizations are taking agile approaches on one or more projects, and by a wide range of organizations. The same survey also indicated that organizations are attempting large agile projects, several respondents indicated that they were not only doing but successful with agile project teams of over 200 people, and many indicated that they were applying agile in distributed environments. Agile project teams also appear to have higher rates of success than do traditional teams [2] indicating that agile approaches are likely here to stay.

Agile techniques have clearly been proven in simple settings and we're seeing that many organizations are now applying agile at scale. In this paper I explore the factors surrounding apply agile techniques at scale, large or distribute teams are just two of the many issues which agile teams now face, and overview a collection of techniques which I've applied successfully in practice.

2 Scaling Factors

When you read some of the agile literature it sounds rather naïve at times. Although we would all love nothing more than to work with small, co-located, closely-knit teams of highly-skilled professionals who are building brand new systems it rarely seems to be the case in practice. Instead one or more "scaling factors" seems to ruin this perfect scenario for us. When you think about scaling agile approaches the first factors that you consider are team size and geographical distribution [3], and although these are clearly important scaling factors they're not the only ones. At IBM Rational we have found that when applying agile strategies at scale you are likely to run into one or more of the following complexity factors:

1. Team size. Large teams will be organized differently than small teams, and they'll work differently too. Strategies that work for small co-located teams won't be sufficient for teams of several hundred people.
2. Geographical distribution. Some members of a team, including stakeholders, may be in different locations. Even being in different cubicles within the same building can erect barriers to communication, let alone being in different cities or even on different continents.
3. Entrenched culture. Most agile teams need to work within the scope of a larger organization, and that larger organization isn't always perfectly agile. The existing people, processes, and policies aren't always ideal. Hopefully that will change in time, but we still need to get the job done right now.
4. System complexity. The more complex the system the greater the need for a viable architectural strategy. An interesting feature of the Rational Unified Process (RUP) [4] is that its Elaboration phase's primary goal is to prove the architecture via the creation of an end-to-end, working skeleton of the system. This risk-reduction technique, described later in this paper, is clearly a concept which Extreme Programming (XP) [5] and Scrum [6] teams can clearly benefit from.
5. Legacy systems. It can be very difficult to leverage existing code and data sources due to quality problems. The code may not be well written, documented, or even have tests in place, yet that doesn't mean that your agile team should rewrite everything from scratch. Some legacy data sources are questionable at best, or the owners of those data sources difficult to work with, yet that doesn't given an agile team license to create yet another database.
6. Regulatory compliance. Regulations, including the Sarbanes-Oxley act, BASEL-II, and FDA statutes can increase the documentation and process burden on your projects. Complying with these regulations while still remaining as agile as possible can be a challenge.
7. Organizational distribution. When a team is made up of people working for different divisions, or from different companies (such as contractors, partners, or consultants), then management complexity rises.
8. Degree of governance. If you have one or more IT projects then you have an IT governance process in place. How formal it is, how explicit it is, and how effective it is will be up to you. Agile/lean approaches to governance are based on collaborative approaches which enable teams to do the right thing, as opposed to traditional approaches which implement command-and-control strategies [7]. More on this later.
9. Enterprise focus. It is possible to address enterprise issues, including enterprise architecture, portfolio management, and reuse within an agile environment. The Enterprise Unified Process (EUP) extends evolutionary processes such as RUP or XP to bring an enterprise focus to your IT department [8].

The point is that agile is relative, that different environments will require different strategies to scale agile approaches effectively. This implies that you need to have a collection of techniques at your disposal.

3 Strategies for Scaling Agile Approaches

It is not only possible to scale agile software development approaches, the strategies to do so already exist. These strategies are:

- Consider the full system lifecycle
- Agile Model Driven Development (AMDD)
- Continuous independent testing
- Risk and value-driven development
- Agile database techniques
- Lean development governance.

3.1 Consider the Full System Lifecycle

Figure 1 depicts a system development lifecycle (SDLC) which shows the phases and major activities involved with the development and release into production of a system following an agile approach [9]. There are four phases to this SDLC, taking their name from the phases of the Unified Process [4, 8]:

- Inception. This is the initial phase of the project where you gain initial funding, perform initial requirements and architecture envisioning, obtain initial resources, and set up your environment. The goal is to define a firm foundation for your project team. This phase is also referred to as Iteration 0, Sprint 0, and Warm Up by various agile methods.
- Elaboration & Construction. During this phase you develop working software which meets the needs of your stakeholders. This phase is also referred to as Development, Construction, and Implementation by various agile methods.
- Transition. During this phase you do the work required to successfully deploy your system into production. This includes finalizing testing, finalizing documentation, baselining your project work products, training end users, training operations and support staff, and running pilot programs as necessary. This phase is also referred to as Release, Deployment, or End Game by various agile methods.
- Production. During this phase, typically the majority of the system lifecycle, you operate and support the system and your end users (hopefully) use it. This phase is also referred to as Maintenance and Support by some agile methods.

There are several reasons why it is important to adopt the lifecycle of Figure 1. First, too many agile teams focus on the construction aspects of the SDLC without taking into account the complexities of initiating a project, deploying into production, or even running the system once it is in production. The risks addressed by these phases are critical regardless of scale, but increase in importance in proportion to the rise in complexity resulting from the scaling factors mentioned earlier. Second, the lifecycle explicitly includes important scaling techniques such as initial requirements and architecture envisioning as well as continuous independent testing.

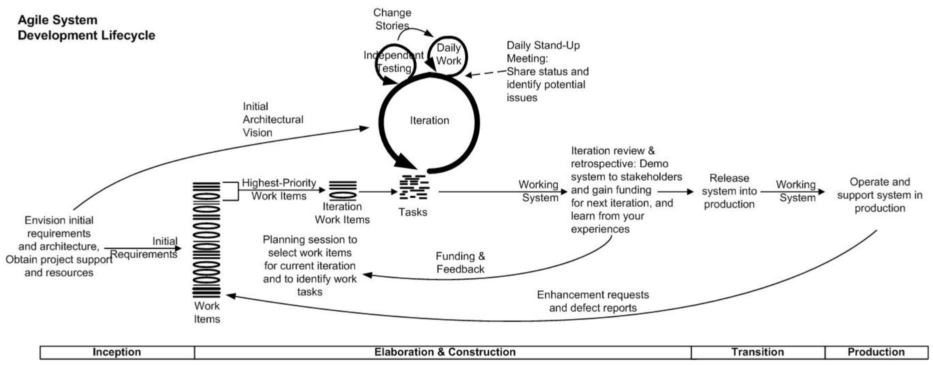

Agile System Development Lifecycle
Change Stories
Independent Testing
Daily Work
Daily Stand-Up Meeting: Share status and identify potential issues
Iteration
Initial Architectural Vision
Highest-Priority Work Items
Iteration Work Items
Tasks
Working System
Iteration review & retrospective: Demo system to stakeholders and gain funding for next iteration, and learn from your experiences
Release system into production
Working System
Operate and support system in production
Envision initial requirements and architecture, Obtain project support and resources
Initial Requirements
Work Items
Planning session to select work items for current iteration and to identify work tasks
Funding & Feedback
Enhancement requests and defect reports
Inception
Elaboration & Construction
Transition
Production

Fig. 1. The Agile System Development Lifecycle (SDLC)

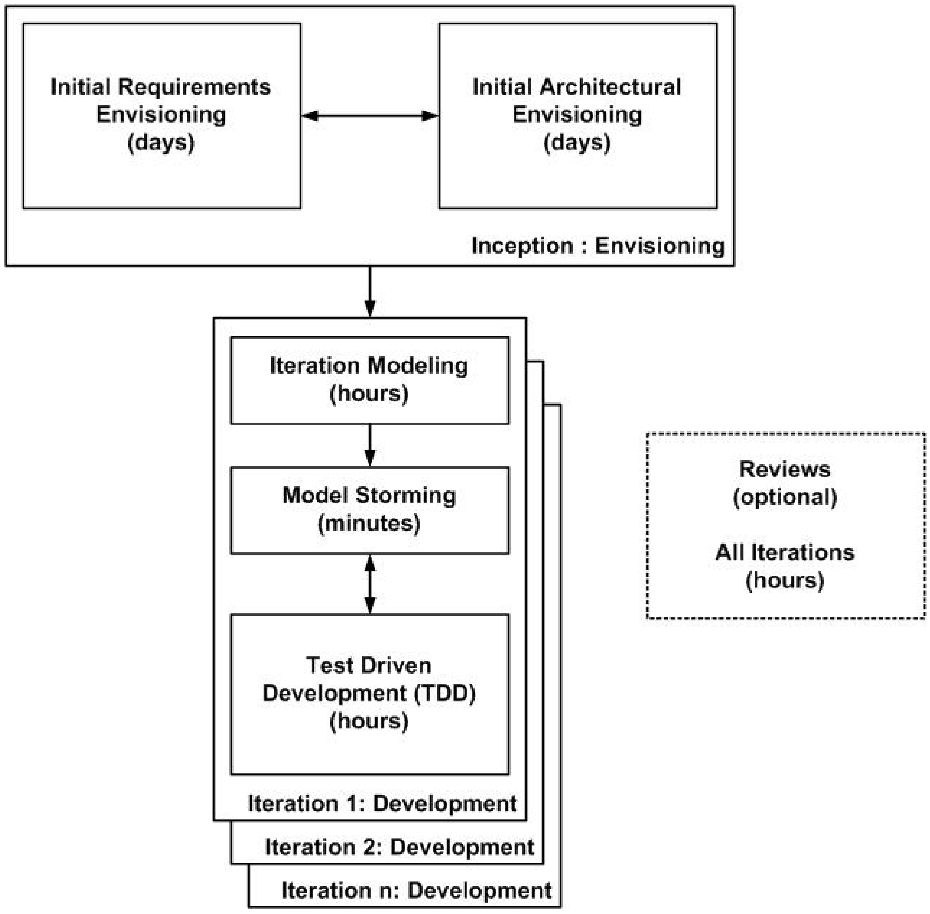


Fig. 2. The Agile Model Driven Development (AMDD) lifecycle for a project

3.2 Agile Model Driven Development (AMDD)

As the name implies, Agile Model Driven Development (AMDD) is the agile version of Model Driven Development (MDD). MDD is an approach to software development where extensive models are created before source code is written. With traditional MDD a serial approach to development is often taken where comprehensive models are created early in the lifecycle. With AMDD you create agile models which are just barely good enough for the current situation at hand to that drive your overall development efforts. Figure 2 depicts the AMDD lifecycle for a project.

As you can see in Figure 2 there are four critical modeling and specification activities:

1. Envisioning. The envisioning effort is typically performed during the first week of a project, the goal of which is to identify the scope of your system and a likely architecture for addressing it. To do this you will do both high-level requirements and high-level architecture modeling. The goal isn't to write detailed specifications but instead to explore the requirements and come to an overall strategy for your project. For short projects (perhaps several weeks in length) you may do this work in the first few hours and for long projects (perhaps on the order of twelve or more months) you may decide to invest two weeks in this effort due to the risks inherent in over modeling.
2. Iteration modeling. At the beginning of each Construction iteration the team must plan out the work that they will do that iteration, and an often neglected aspect of this effort is the required modeling activities implied by the technique. Agile teams implement requirements in priority order, as you can see with the work item stack of Figure 1, pulling an iteration's worth of work off the top of the stack. To do this you must be able to accurately estimate the work required for each requirement, then based on your previous iteration's velocity (a measure of how much work you accomplished) you pick that much work off the stack. To estimate a work item effectively you will need to think through how you intend to implement it, and very often you'll model (often using inclusive tools such as whiteboards or paper) to do so.
3. Model storming. Model storming is just in time (JIT) modeling: you identify an issue which you need to resolve, you quickly grab a few team mates who can help you, the group explores the issue, and then everyone continues on as before. These "model storming sessions" are typically impromptu events, one project team member will ask another to model with them, typically lasting for five to ten minutes (it's rare to model storm for more than thirty minutes). The people get together, gather around a shared modeling tool (e.g. the whiteboard), explore the issue until they're satisfied that they understand it, then they continue on (often coding). Extreme programmers (XPers) would call modeling storming sessions stand-up design sessions or customer Q&A sessions.
4. Test-driven development (TDD). TDD is a technique where you write a single test and then just enough production code to fulfill that test [11]. Not only are you validating your software to the extent of your understanding of the stakeholder's intent up to that point, you are also specifying your software on a JIT basis. In short, with TDD agile teams capture detailed specifications in the form of executable tests instead of static documents or models.

Sinaalto and Abrahamsson [12] found that TDD may produce less complex code but that the overall package structure may be difficult to change and maintain. In other words, they found that although TDD is effective for "design in the small" that it is not effective for "design in the large". AMDD enables you to scale TDD through initial envisioning of the requirements and architecture as well as just-in-time (JIT) modeling at the beginning and during construction iterations.

AMDD also helps to scale agile software development when the team is large and/or distributed and when "the team" is the entire IT effort at the enterprise level. Figure 3 shows an agile approach to architecture at the program and enterprise levels [13, 14]. The architecture owners, the agile term for architects, develop the initial architecture vision through some initial modeling. They then become active participants on development teams, often taking on the role of architecture owner or technical team lead, and thereby help the team implement their part of the overall architecture. They take issues, and what they've learned from their experience on the project teams, back to the architecture team on a regular basis (at least weekly) to evolve the architecture artifacts appropriately.

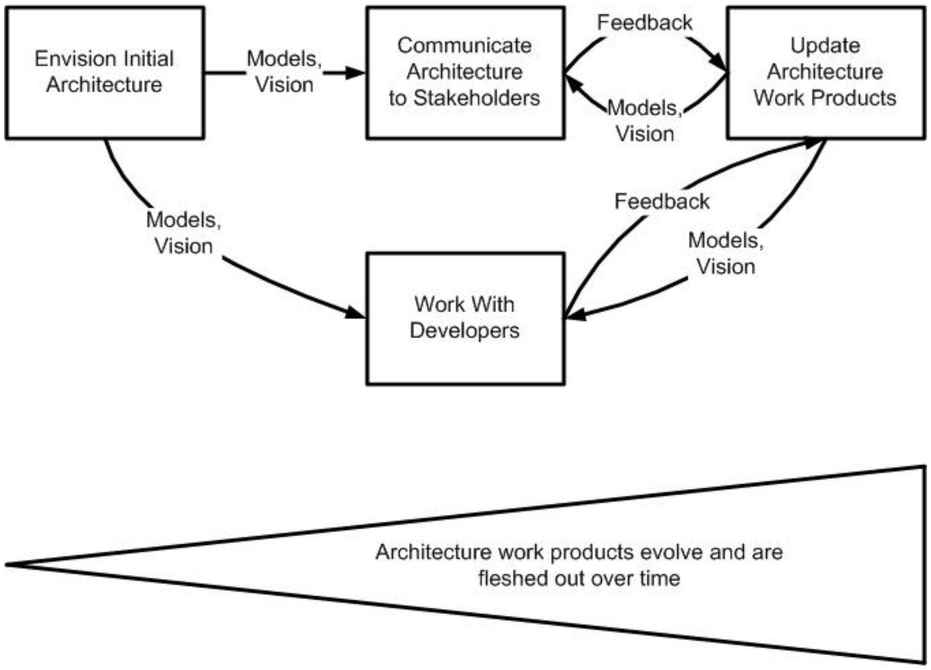

Envision Initial Architecture
Models, Vision
Communicate Architecture to Stakeholders
Feedback
Models, Vision
Update Architecture Work Products
Models, Vision
Feedback
Models, Vision
Work With Developers
Architecture work products evolve and are fleshed out over time

Fig. 3. An agile approach to program/enterprise architecture

3.3 Continuous Independent Testing

Although AMDD scales the specification aspects of TDD, it does nothing for the validation aspects. TDD is in effect an approach to confirmatory testing where you validate the system to the level of your understanding of the requirements. This is the equivalent of "smoke testing" or testing against the specification – while important, it isn't the whole validation picture. The fundamental challenge with confirmatory testing, and hence TDD, is that it assumes that stakeholders understand and can describe their requirements. Although iterative approaches increase the chance of this there are no guarantees. A second assumption of TDD is that developers have the skills to write and run the tests, skills that can be gained over time but which they may not have today.

The implication is that you need to add independent, investigative testing practices into your software process [15]. The goal of investigative testing is to explore issues that your stakeholders may not have thought of, such as usability issues, system integration issues, production performance issues, security issues, and a multitude of others. Agile teams, particularly those working at scale, often having a small independent test team working in parallel with them, as you can see depicted in Figure 1. The development team deploys the current working build into the testing sandbox, an environment which attempts to simulate the production environment. This deployment effort occurs at least once an iteration, although minimally I suggest doing so at least once a week if not nightly (assuming your daily build was successful).

The independent testers don't need a lot of details: The only documentation that they might need is a list of changes since the last deployment so that they know what to focus on first, because most likely new defects would have been introduced in the implementation of the changes. They will use complex, and often expensive, tools to do their jobs and will usually be very highly skilled people.

When the testers find a potential problem, it might be what they believe is missing functionality or it might be something that doesn't appear to work properly, which they write up as a "change story". Change stories are basically the agile form of a defect report or enhancement request. The development team treats change stories like requirements—they estimate the effort to address the requirement and ask their project stakeholder(s) to prioritize it accordingly. Then, when the change story makes it to the top of their prioritized work-item list, they address it at that point in time. Any potential defect found via independent investigative testing becomes a known issue that is then specified and validated via TDD. Because TDD is performed in an automated manner to support regression testing, the implication is that the investigative testers do not need to be as concerned about automating their own efforts, but instead can focus on the high-value activity of defect detection.

3.4 Risk and Value-Driven Development

The explicit phases of the Unified Process (UP) and their milestones are important strategies for scaling agile software development to meet the real-world needs of modern organizations. The UP lifecycle is risk and value driven [16]. What this means is that UP project teams actively strive to reduce both business and technical risk early in the lifecycle while delivering concrete feedback throughout the entire lifecycle in the form of working software. Where agile processes such as XP and Scrum are clearly value driven, they can be enhanced to address risk more effectively. This is particularly important at scale due to the increased risk associated with the greater complexity of such projects.

Each UP phase addresses a different kind of risk:

1. Inception. This phase focuses on addressing business risk by having you drive to scope concurrence amongst your stakeholders. Most projects have a wide range of stakeholders, and if they don't agree to the scope of the project and recognize that others have conflicting or higher priority needs you project risks getting mired in political infighting.
2. Elaboration. The goal of this phase is to address technical risk by proving the architecture through code. You do this by building and end-to-end skeleton of your system which implements the highest-risk requirements. These high-risk requirements are often the high-business-value ones anyway, so you usually need to do very little reorganization of your work items stack to achieve this goal.
3. Construction. This phase focuses on implementation risk, addressing it through the creation of working software each iteration. This phase is where you put the flesh onto the skeleton.
4. Transition. The goal of this phase is to address deployment risk. There is usually a lot more to deploying software than simply copying a few files onto a server, as I indicated above. Deployment is often a complex and difficult task, one which you often need good guidance to succeed at.
5. Production. The goal of this phase is to address operational risk. Once a system is deployed your end-users will work with it, your operations staff will keep it up and running, and your support staff will help end users to be effective. You need effective processes in place to achieve these goals.

The first four phases end with a milestone review, which could be as simple as a short meeting, where you meet with prime stakeholders who will make a “go/no-go" decision regarding your system. They should consider whether the project still makes sense, perhaps the situation has changed, and that you're addressing the project risks appropriately. This is important for “agile in the small” but also for “agile in the large” because at scale your risks are often much greater. These milestone reviews enable you to lower project risk. Although agile teams appear to have a higher success rate than traditional teams, some agile projects are still considered failures [2]. The point is that it behooves us to actively monitor development projects to determine if they're on track, and if not either help them to get back on track or cancel them as soon as we possibly can.

3.5 Agile Database Techniques

Data is an important aspect of any business application, and to a greater extent of your organization’s assets as a whole. Just as your application logic can be developed in an agile manner, so can your data-oriented assets [13]. To scale agile effectively, all members of the team must work in an agile manner, including data professionals. The following techniques enable data professionals to be active members of agile teams:

1. Database refactoring. A database refactoring is a simple change to a database schema that improves its design while retaining both its behavioral and informational semantics [17]. A database schema includes both structural aspects such as table and view definitions as well as functional aspects such as stored procedures

and triggers. A database refactoring is conceptually more difficult than a code refactoring; code refactorings only need to maintain behavioral semantics while database refactorings also must maintain informational semantics. The process of database refactoring is the act of applying database refactorings in order to evolve an existing database schema, either to support evolutionary/agile development or to fix existing database schema problems.
2. Database testing. Databases often persist mission-critical data which is updated by many applications and potentially thousands if not millions of end users. Furthermore, they implement important functionality in the form of database methods (stored procedures, stored functions, and/or triggers) and database objects (e.g. Java or C# instances). The best way to ensure the continuing quality of these assets, at least from a technical point of view, is to have a full regression test suite which you can run on a regular basis.
3. Continuous database integration. Continuous integration is a development practice where developers integrate their work frequently, at least daily, where the integration is verified by an automated build. The build includes regression testing and possibly static analysis of the code. Continuous database integration is the act of performing continuous integration on your database assets. Database builds may include the creation of the database schema from scratch, something that you would only do for development and test databases, as well as database regression testing and potential static analysis of the database contents. Continuous integration reduces the average amount of time between injecting a defect and finding it, improving your opportunities to address database and data quality problems before they get out of control.
4. Agile data modeling. Agile data modeling is the act of exploring data-oriented structures in an iterative, incremental, and highly collaborative manner. Your data assets should be modeled, via an AMDD approach, along with all other aspects of what you are developing.

3.6 Lean Development Governance

Governance is critical to the success of any IT department, and it is particularly important at scale. Effective governance isn't about command and control, instead the focus is on enabling the right behaviors and practices through collaborative and supportive techniques. It is far more effective to motivate people to do the right thing than it is to try to force them to do so. Per Kroll and myself have identified a collection of practices that define a lean approach to governing software development projects [7]. These practices are:

1. Adapt the Process. Because teams vary in size, distribution, purpose, criticality, need for oversight, and member skillset you must tailor the process to meet a team's exact needs. Repeatable results, not repeatable processes, should be your true goal.
2. Align HR Policies With IT Values. Hiring, retaining, and promoting technical staff requires different strategies compared to non-technical staff.

3. Align Stakeholder Policies With IT Values. Your stakeholders may not understand the implications of the decisions that they make, for example that requiring an "accurate" estimate at the beginning of a project can dramatically increase project risk instead of decrease it as intended.
4. Align Team Structure With Architecture. The organization of your project team should reflect the desired architectural structure of the system you are building to streamline the activities of the team.
5. Business-Driven Project Pipeline. Invest in the projects that are well-aligned to the business direction, return definable value, and match well with the priorities of the enterprise.
6. Continuous Improvement. Strive to identify and act on lessons learned throughout the project, not just at the end. Embedded Compliance. It is better to build compliance into your day-to-day process, instead of having a separate compliance process that often results in unnecessary overhead.
7. Continuous Project Monitoring. Automated metrics gathering enables you to monitor projects and thereby identify potential issues so that you can collaborate closely with the project team to resolve problems early.
8. Flexible Architectures. Architectures that are service-oriented, component-based, or object-oriented and implement common architectural and design patterns lend themselves to greater levels of consistency, reuse, enhanceability, and adaptability.
9. Integrated Lifecycle Environment. Automate as much of the "drudge work", such as metrics gathering and system build, as possible. Your tools and processes should fit together effectively throughout the lifecycle.
10. Iterative Development. An iterative approach to software delivery allows progressive development and disclosure of software components, with a reduction of overall failure risk, and provides an ability to make fine-grained adjustment and correction with minimal lost time for rework.
11. Pragmatic Governance Body. Effective governance bodies focus on enabling development teams in a cost-effective and timely manner. They typically have a small core staff with a majority of members being representatives from the governed organizations.
12. Promote Self-Organizing Teams. The best people for planning work are the ones who are going to do it.
13. Risk-Based Milestones. You want to mitigate the risks of your project, in particular business and technical risks, early in the lifecycle. You do this by having throughout your project several milestones that teams work toward.
14. Scenario-Driven Development. By taking a scenario-driven approach, you can understand how people will actually use your system, thereby enabling you to build something that meets their actual needs. The whole cannot be defined without understanding the parts, and the parts cannot be defined in detail without understanding the whole.
15. Simple and Relevant Metrics. You should automate metrics collection as much as possible, minimize the number of metrics collected, and know why you're collecting them.
16. Staged Program Delivery. Programs, collections of related projects, should be rolled out in increments over time. Instead of holding back a release to wait for a

subproject, each individual subprojects must sign up to predetermined release date. If the subproject misses it skips to the next release, minimizing the impact to the customers of the program.

17. Valued Corporate Assets. Guidance, such as programming guidelines or database design conventions, and reusable assets such as frameworks and components, will be adopted if they are perceived to add value to developers. You want to make it as easy as possible for developers to comply to, and more importantly take advantage of, your corporate IT infrastructure.

4 Conclusion

It is definitely possible to scale Agile software development to meet the real-world complexities faced by modern organizations. Based on my experiences, I believe that over the next few years we'll discover that agile approaches scale better than traditional approaches. Many people have already discovered this, and have adopted some or all of the strategies outlined in this paper, but as an industry I believe that there isn't yet sufficient evidence to state this as more than opinion. Time will tell.

References

1. Ambler, S.W.: Dr. Dobb's Journal Agile Adoption Survey 2008 (2008). Accessed on March 22, 2008, `http://www.ambysoft.com/surveys/agileFebruary2008.html`
2. Ambler, S.W.: Dr. Dobb's Journal Project Success Rates Survey 2007 (2007). Accessed on March 22, 2008, `http://www.ambysoft.com/surveys/success2007.html`
3. Eckstein, J.: Agile Software Development in the Large: Diving into the Deep. Dorset House Publishing, New York (2004)
4. Kruchten, P.: The Rational Unified Process: An Introduction, 3rd edn. Addison Wesley Longman, Reading (2004)
5. Beck, K.: Extreme Programming Explained—Embrace Change. Addison-Wesley Longman, Reading (2000)
6. Schwaber, K.: The Enterprise and Scrum. Microsoft Press, Redmond (2007)
7. Kroll, P., Ambler, S.W.: Lean Development Governance (2007). Accessed on March 22, 2008, `https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?lang=en_US&source=swg-ldg`
8. Ambler, S.W., Nalbone, J., Vizdos, M.J.: The Enterprise Unified Process: Extending the Rational Unified Process. Pearson Education, Upper Saddle River (2005)
9. Ambler, S.W.: The Agile System Development Lifecycle (SDLC) (2005). Accessed on March 22, 2008, `http://www.ambysoft.com/essays/agileLifecycle.html`
10. Ambler, S.W.: Agile Model Driven Development (AMDD) (2003). Accessed on March 22, 2008, `http://www.agilemodeling.com/essays/amdd.htm`
11. Astels, D.: Test Driven Development: A Practical Guide. Prentice Hall, Upper Saddle River (2003)
12. Sinaalto, M., Abrahamsson, P.: Does Test Driven Development Improve the Program Code? Alarming Results from a Comparative Case Study. In: CEE-SET 2007 Conference Proceedings (2007)
13. Ambler, S.W.: Agile Database Techniques: Effective Strategies for the Agile Software Developer. Wiley, New York (2003)

14. McGovern, J., Ambler, S.W., Stevens, M.E., Linn, J., Sharan, V., Jo, E.K.: The Practical Guide to Enterprise Architecture. Prentice Hall PTR, Upper Saddle River (2004)
15. Ambler, S.W.: Agile Testing Strategies. Dr. Dobb's Journal, January 2007 (2007). Accessed on March 22, 2008, `http://www.ddj.com/development-tools/196603549`
16. Kroll, P., MacIsaac, B.: Agility and Discipline Made Easy: Practices from OpenUP and RUP. Addison Wesley Longman, Reading (2006)
17. Ambler, S.W., Sadalage, P.J.: Refactoring Databases: Evolutionary Database Design. Addison Wesley, Boston (2006)

Bio

Scott W. Ambler is Practice Leader Agile Development with IBM Rational. Scott has a Master of Information Science from the University of Toronto and is author of several books including Agile Modeling, Agile Database Techniques, and Refactoring Databases. Scott helps organizations around the world to improve their software processes. He is a Senior Contributing Editor with Dr. Dobb's Journal (www.ddj.com) and writes about strategies for scaling software development at www.ibm.com/developerworks/blogs/page/ambler.

Formalisms in Software Engineering: Myths Versus Empirical Facts

Dieter Rombach and Frank Seelisch

Fraunhofer Institute for Experimental Software Engineering
{dieter.rombach,frank.seelisch}@iese.fraunhofer.de

Abstract. The importance of software grows in all sectors of industry and all aspects of life. Given this high dependability on software, the status of software engineering is less than satisfactory. Accidents, recall actions, and late projects still make the news every day. Many of the software engineering research results do not make it into practice, and thereby the gap between research and practice widens constantly. The reasons for not making it into practice range from isufficient commitment for professionalization of software development on the industrial side, to insufficient consideration for practical scale-up issues on the research side, and a tremendous lack of empirical evidence regarding the benefits and limitations of new software engineering methods and tools on both sides. The major focus of this paper is to motivate the creation of credible evidence which in turn will allow for less risky introduction of new software engineering approaches into practice. In order to overcome this progress hindering lack of evidence, both research and practice have to change their paradigms. Research needs to complement each promising new software engineering approach with credible empirical evidence from in vitro controlled experiments and case studies; industry needs to baseline its current state of the practice quantitatively, and needs to conduct in vitro studies of new approaches in order to identify their benefits and limitations in certain industrial contexts.

Keywords: Computer science, [empirical] software engineering, software development, empiricism, empirical evidence.

1 Introduction

Software plays an ever increasing role in all aspects of our lives. The mere amount of code embedded in modern, highly integrated products should mandate that the development of software follow widely agreed principles. Furthermore, these principles ought to ensure quality goals that have been specified in advance. The function of software in life supporting systems as well as in numerous standard applications which allow to perform business processes more efficiently, proves that software has become vital in all respects.

However, we witness serious system failures resulting from faulty software, sometimes with tremendous consequences: People die, assets are being lost, and products need to be recalled by renowned OEMs. Software engineering today, seen as a practically highly relevant engineering discipline, is not mature enough considering the role it plays.

Likewise, the relationship between computer science and software engineering needs to be stated more precisely and lived, so that theoretical research in computer science can be translated into practical methods and procedures that can be utilized by software engineers in software development projects.

The major claim of this work is that typical shortcomings in the practical work of software engineers as we witness them today, result from missing or unacknowledged empirical facts. Discovering facts by empirical studies is the only way to gain insights in how software development projects should be run best, i.e., insights in the discipline of software engineering. Empirical facts will in turn motivate computer science research.

This paper is organized as follows. The following Chap. 2 discusses the role of software in industry and society, and adds an economic standpoint. How do companies estimate the importance of software for their business? What are proven economic guidelines for realizing software development projects?

Chapter 3 summarizes the typical practical problems resulting from immature software engineering, and identifies the shortcomings of software engineering in practice. It comes up with general explanations for the problems we are currently confronted with when developing software in critical settings, e.g., within tight schedules. Moreover, we argue that the current situation is likely to become even more critical, as software systems become more and more complex.

Having detailed practical problems of software engineering, Chap. 4 addresses the possible solutions to these problems offered by research. Starting with general principles of computer science we go top-down towards software engineering as its practical toolbox for developing software, and finally to the most important ingredient of software engineering: empirical facts.

Chapter 5 illustrates the need for more empirical software engineering, both generally and by means of the concrete example of the special inspection technique *reading*.

We close with an outlook on next steps.

2 The Role of Software in Industry

In the previous section, the spectrum of common problems due to software failures has been detailed. From an economic standpoint, these problems imply direct costs as well as great efforts for repairing, fitting, bug-fixing and necessary changes in software versions, products, and processes. In order to save precious resources, *companies need to enforce the usage of best practice methods and procedures of software engineering.*

On the other hand, software engineering has been offering and still offers completely new ways for developing products.

2.1 Software as Driver for Innovation

Due to its enabling role, software has become a major factor in today's industry. Industrial leaders in the automotive business, in the field of medical devices, and logistics estimate that 80% of all innovation is directly triggered by software.

However, most non-IT companies do not reflect these estimates in their organizational structure: Software is important but IT sub-organizations currently do not seem to be. It is a common model to have these sub-organizations raise their funds within the company instead of being given basic funding.

We expect that this organizational setup adds to the current problems in ongoing software projects, as strategic planning cannot be sufficiently considered.

2.2 Economic Aspects of Software Development

In the context of globalization, industrial companies focus more than ever on the goal parameters *quality, cost,* and *time to market.* Especially quality promises to offer better chances to establish unique selling propositions in the lucrative upper market segments. For most Western economies, this may offer a way to legitimate the much higher level of salaries compared to Asian economies.

In terms of software engineering, this means that development projects will also primarily be managed and steered by these parameters. And especially concerning the parameter *time to market* companies face a classical trade-off: Is it more important to deliver a new software version fast, while taking into account that more effort will have to be spent on bug-fixing? Or, is it the goal of the company to minimize costs over the entire software life cylce?

According to a model for business development by Stalk, Evans and Shulman [1] companies need to adher to the following, very roughly sketched strategic roadmap:

1. They need to be clear about their overall strategic goals.
2. They need to identify supporting business processes that guarantee the given strategic goals.
3. They need to prioritize the most important sub-processes and continually invest into them.

In the context of software development, the business processes are software development processes. The highest priority processes to be invested into in the case of safety goals, could be design and verification processes. Investment into a software engineering subprocess P means to invest into the creation of empirical evidence regarding its effectiveness f with respect to some goal G in the context of the given environment C. G could be any one of the afore mentioned goals quality, cost, or time.

Thus, the challenge could be phrased as identifying the following function:

$$G == f(P, C), \tag{1}$$

where "==" stands for an empirically based relationship.

As a consequence of the processes defined under 2. and the selection made under 3., a number of software development projects with management attention will normally be started. Note that the above question whether a software version should be delivered quickly or whether the company should aim at minimizing total cost along the product life cycle, is *not* answered by the above generic roadmap. From an IT perspective, this decision has implications for software attributes such as adaptability and sustainability: The former strategy will in general lead to a higher adaptability, whereas the latter naturally leads to more sustainable software solutions.

3 Practice of Software Engineering

3.1 Problems

Today's software engineering practice is mainly characterized by the following two phenomena.

Schedule and Budget Overrun: Taking a closer look at software development projects in industrial settings reveals that schedule and budget overrun is not at all uncommon. In some projects, rates of schedule overrun of up to 150% have been witnessed.

Safety Criticality: Besides these mere management obstacles, accidents with sometimes dramatic consequences are a much more serious problem. Generally speaking, here, software failures imply safety-critical situations while handling products with embedded software, e.g., cars, trains, or airplanes. Whenever serious safety problems had been detected, OEMs had to recall their products, which typically results in a great loss of assets both economically and in terms of company reputation and product image.

3.2 Reasons for Current Problems

Non-Compliance with Best-Practice Principles of Software Development such as:

– encapsulation,
– information hiding,
– proven architectural patterns,
– traceability, e.g., diversion of documentation versus code over time.

The year 2000 problem (2YK) is a prominent example of poor encapsulation and information hiding. Generally speaking, information hiding will lead to small interfaces between program modules. This does not only increase readability and manageability of code but also enables a potentially higher reuse of these modules, most likely at a lower cost.

Non-Compliance with Best Practice Principles of Process Design such as:

– review and inspection techniques for an earlier defect detection; see [2] and Sect. 5.2,

– best practice process patterns,
– best practice process design tools, e.g., Waterfall or V-Model.

Non-Existence of Credible Evidence Regarding the Effects of Methods and Tools, i.e.,

– missing empirical facts,
– missing context information,
– missing certainty information.

In most problematic software development projects, teams will have to deal with a combination of both kinds of non-compliance, and will additionally suffer from missing context information, or empirical facts.

Figure 1 refines formula (1) given in Sect. 2.2. It illustrates how the precision of the prediction of project development time T for a process model P may depend on context C: Without any context knowledge, prediction across projects may be off by as much as 150%; see [3]. By taking into consideration parameters such as size of the system to be developed and experience of the developers, one can reduce the variance significantly; in many cases to a single-digit variance.

3.3 Future Trends and Challenges

The outlined current situation is not likely to improve by itself. Actually, there are some trends which will most probably aggravate the setting.

Embedded Systems and Increasing Complexity: More and more software is being embedded in life-supporting systems, e.g., medical devices used in operation theatres. Software is key to realizing new functions without including too many additional hardware components. Thereby, systems are becoming more and more complex. We have begun to build highly integrated *systems of systems* in which the same piece of software implements more than just one system function.

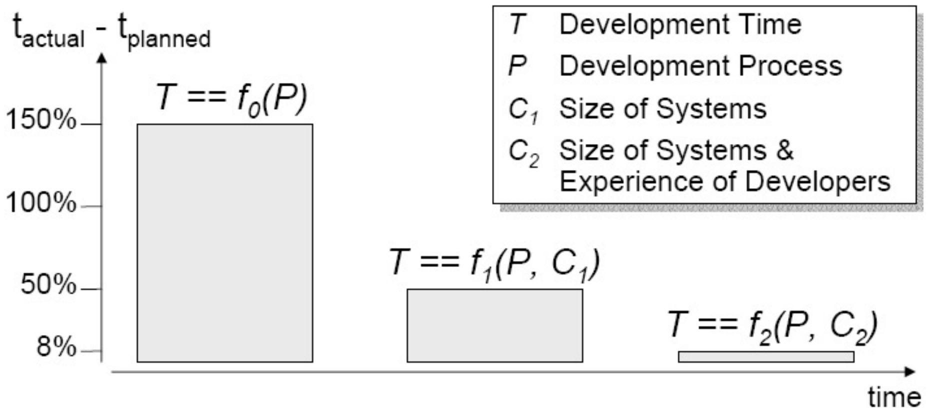


Fig. 1. Relationship between Context Understanding and Predictive Capability

Cross-Disciplinary Systems: Software systems find their way into domains that have previously been dominated by classically engineered solutions. Here, classical disciplines and software engineering already begin to form new hybrid disciplines for which only few skilled engineers are available.

Ubiquitious Computing and Ambient Intelligence: With ubiquitious computing and ambient technologies we witness a miniaturization combined with a remarkable multiplication of new, spontaneously connecting components. These concepts drive the development of highly flexible, service oriented software architectures which ensure high degrees of robustness.

Loss of Direct Control: Ubiquitious Computing and Ambient Intelligence will also push the development of globally interconnected information systems, and systems with autonomous control that are able to spontaneously establish networks and reorganize themselves. The human user will thus no longer be able to directly control these systems.

It is hard to forecast what in detail these trends are going to imply with respect to the development of software.

Most likely, development standards will need to become more definite and resilient, in order to ensure that software operate as specified and be safe, secure, and trustworthy.

4 Research in Software Engineering

4.1 Computer Science and Software Engineering

Computer science is the well-established science of computers, algorithms, programs, and data structures. Just like physics, its body of knowledge, can be characterized by facts, laws, and theories. But, whereas physics deals with natural laws of our physical world, computer science is a body of *cognitive laws*; cf. [4].

Additionally, when software engineering deals with the creation of large software artifacts, then its role is more similar to mechanical and electrical engineering where the goal is to create large mechanical or electronic artifacts.

Figure 2 shows the positioning of computer science and software engineering in the landscape of sciences.

Each science provides a kernel set of principles that gives rise to a set of practical methods for manipulating the world in one or the other useful way. For example, civil engineering methods including statics calculations can be applied to build a bridge; here the kernel set of laws to do so correctly stems from mathematics and physics.

In this sense, software engineering can be seen as the analog set of methods for developing software, based on fundamental results from computer science. For instance, research in computer science gave rise to functional semantics. From that formal foundation, software engineering derived inspection techniques such as *stepwise abstraction* and *cleanroom development*, and proved furthermore the practicability of these methods by means of empirical studies.

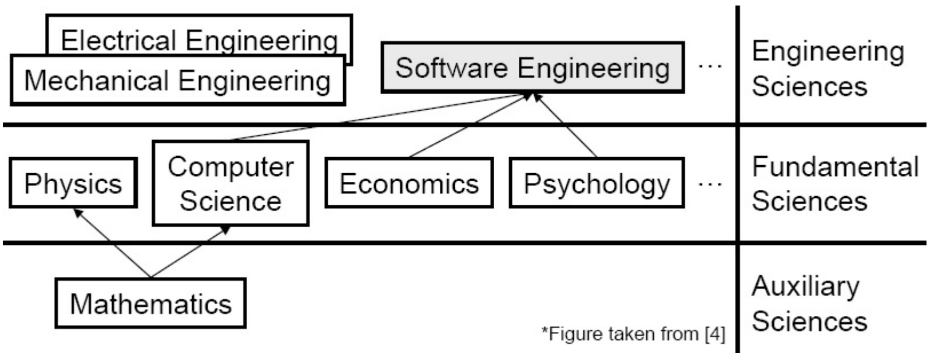


Fig. 2. Positioning Software Engineering in the Landscape of Sciences

Software engineering needs to be based upon formal foundations, but on the other hand, pure computer science concepts are generally not applicable in practice, e.g., due to algorithmic complexity.

4.2 Software Engineering Principles

Let's take a look at a very general and powerful principle of computer science:

If we need to solve a hard problem, we may try to partition it into smaller subproblems for which we know how to solve them. The basic recursive algorithm for this general pattern of *divide and conquer* is shown in Fig. 3. (Problem partition and combination of partial solutions will of course depend on the particular problem at hand).

Divide and Conquer in Software Development Projects: We also use this *divide and conquer* principle when working in software development projects: If the software development process is large, software engineers typically try to partition it into subprocesses with well-defined milestones.

If the task is to create a software product for performing a variety of related business tasks, they attempt to partition the set of requirements into subsets with small mutual overlap.

The following insights have been extracted from a series of experiments:

– In large projects with comparably low risk, e.g., due to available domain knowledge and an experienced project team (see Fig. 1), it is best to use process-oriented development models, like e.g. the Waterfall or V-Model.
– Is the project small but characterized by a high risk, e.g., due to missing domain knowledge and thus the necessity to apply a general approach, software engineers should apply product-oriented models, e.g., agile development methods.
– Large projects with high risk resulting from tight deadlines and poor domain knowledge are best run using product-oriented models, e.g., incremental development techniques.

```
[!t] solve_problem(Problem $p$)
  if algorithm_available_for($p$) then
    $a \longleftarrow$ algorithm_for($p$)
    $s \longleftarrow$ apply_algorithm($a$, $p$)
  else
    $\{p_i \ : \ i \in I\} \longleftarrow$ partition_of($p$)
    for each $i \in I$
       $s_i \longleftarrow$ solve_problem($p_i$)
    $s \longleftarrow$ combine_solutions($\{s_i \quad i \in I\}$)
  end if
  return $s$
```

Fig. 3. Divide and Conquer - Solving a Difficult Problem by Partitioning

(The fourth remaining case - small projects with low risk - are the ones that normally pose few problems. Moreover, they are irrelevant for most real-world settings.)

Only when the above software engineering principles will be adhered to, software development teams will have a chance to manage software development projects according to pre-defined budget and schedule.

4.3 Empirical Evidence

The key to success in software development projects is to define measurements and consequently use them in order to be able to measure work progress and detect failure or accomplishment in a rational and transparent manner.

That means that milestones in a development project need to be clearly marked so that responsible and further involved people can determine at any time whether they have been reached or not.

Note that having definitions and measures at one's disposal does not automatically guarantee that they be used. Inforcing their usage must be part of the project and can often only be accomplished by organizational changes or even changes in the working culture. This, in turn, requires top management commitment.

4.4 Evidence Is Context-Dependent

The challenge in software engineering, as a practical standard method and tool box for the development of complex software, is that most tasks will typically have to deal with organizations and will have to address human requirements of some kind. This has two concequences:

1. The methods will vary from organization to organization. Thus, software engineering will depend on the environment in which it is being applied; that is, it is context-sensitive; see again Fig. 1.

2. State of the art software engineering will change over time, as technical progress takes place, and working culture evolves.

An important consequence of both items is that we need to clearly document in what context C a software engineering method can be applied with what result, that is, we need to document the nature of f in formula (1) of Sect. 2.2. Furthermore, a mechanism for periodically revising our knowledge is required, in order to have a valid set in place at any time.

Laying the foundations of software engineering thus means to

– state **working hypotheses** that specify software engineering methods and their outcome together with the **context** of their application,
– make experiments, i.e., studies to gain **empirical evidence**, given a concrete scenario,
– formulate **facts** resulting from these studies, together with their respective context,
– abstract facts to **laws** by combining numerous facts with similar, if not equal, contexts,
– verify working hypotheses, and thereby build up and continously modify a concise **theory** of software engineering as a theoretical building block of computer science.

Current problems of software engineering in pratice can be directly related to these goals:

– Clear working hypotheses are often missing.
– There is no time for, or immediate benefit from empirical studies for the team who undertakes it.
– Facts are often ignored, or applied in differing contexts. Moreover, facts are often replaced by myths, that is, by unproven assumptions.
– Laws are rarely abstracted from facts. The respective contexts are sometimes equated which will lead to false laws; cf.examples in Sect.5.1.
– Up to now, a concise, practicable theory of software engineering does not exist.

5 Empirical Software Engineering

Let us come back to empirical evidence as the most important means to fill gaps in the body of knowledge of software engineering, cf. Sect. 4.3. and [5]. Methods and tools for performing empirical studies exist. Nevertheless, we still do have a lot of myths which impact our discipline in a negative way. Section 5.2 elaborates the example of reading-based inspections vs. testing, to demonstrate how proper use of empirical methods can turn the myth that *"testing is more effective than reading"* into a law that *"in general, reading is more effective"*.

Figure 4 illustrates that there is no software engineering other than empirical software engineering: Its technical building blocks - *formalism, systems*, and

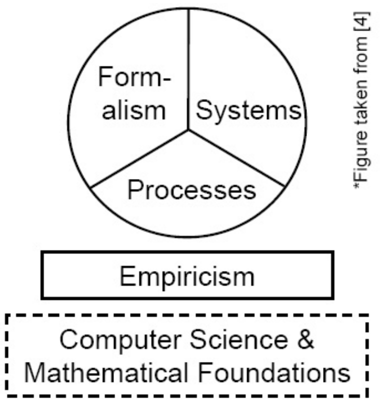


Fig. 4. Categories of Software Engineering

processes - need to be based on empiricism which rests, in turn, on computer science and mathematical foundations.

Prominent representatives of the existing empirical tool box are:

– GQM: *Goal Question Metrics* support decision making in order to guide the software development team towards the relevant measurements.
– QIP: *Quality Improvement Paradigm* have a strong empirical focus.
– EF: *Experience Management* deals with experience and hence expertise in project teams, preferably across different domains.

The Software Engineering Lab (SEL) of NASA's Goddard Space Flight Center has been the first organization establishing a sound experience management along-side a practical software development unit. They have achieved significant and sustained improvements over the years; cf. [6]. For these accomlishments, the SEL had been the first recipient of Carnegie Mellon's Software Engineering Institute's (SEI) Software Process Achievement Award.

According to [7], today, the top institutes on empirical software engineering research are:

– Fraunhofer IESE + CESE, Germany and USA, respectively,
– Simula Research Lab, Norway, and
– NICTA Empirical Group, Australia.

5.1 Facts Versus Myths

Today, empirical software engineering is best characterized by the following three statements:

1. If facts are missing then this gives rise to myths.
2. Concerning software development projects, there are more facts available than being used. Often, software engineers ignore available facts. (This, in turn, often happens under the pressure of unrealistic project schedules.)

3. Besides ignored empirical facts, there exist indeed many gaps of empirical knowledge.

Here are some examples for unproven hypotheses which give rise to myths in empirical software engineering:

– *"Changes are easier when made earlier in the software development process."*
– *"Pair reviews are effective and efficient."*
– *"Re-factoring replaces design for modifiability."*
– *"As the cost of defect reduction increases with lag time, does this mean that we need to focus more on inspections and reviews?"*

Likewise, the following trade-offs are still unresolved and support the insistence on related myths:

– *global distribution of software development versus local concentration*
– *subcontracting versus in-house projects*
– *construction for reuse versus one-time construction*
– *large teams versus small teams (resulting in a longer development time).*

We add some open research questions resulting from established laws; in the sense of the defintion given in Sect. 4.4. We will only be able to answer those based on new empirical evidence. (The cited laws result from the work of Boehm, Endres, Basili, Selby, and Rombach, et al.)

– Law: The cost of defect reduction increases with lag time. Question: *Does this mean that we need to focus more on inspections and reviews, or rather on the design of easily modifiable systems?*
– Law: Formal reviews reduce cost of rework and thus total development effort. Questions: *Under what conditions do object-oriented techniques reduce development effort? Under what conditions does commercial-off-the-shelf (COTS) software reduce development effort?*
– Question: *What is the relationship between good designs and domain knowledge?*
– Question: *Under what conditions can changes be implemented at less cost by means of agile methods?*
– Law: For software components, inspections and reviews are more effective and efficient than testing. (As the law states, this has been proven for software components; initially by Basili and Selby, see [8], and sustained by many other studies.) Question: *Is this also true for entire systems?*

We close this paragraph by giving a concise example of an empirical study. This example is to serve as a guideline for setting up a typical software engineering experiment. It dealt with the special inspection technique *reading*.

5.2 Example: Reading

Reading has become a key engineering technique in the toolbox of inspection procedures. It supports the individual analysis of any textual software document that may be dealing with requirements, design, code, test plans, etc. Generally speaking, reading enables local improvements in the software development process, implying global effects.

The experiment [9] provided insight into the effect of different variables, such as experience of readers and type of defects, on the reading technique. It included

- the investigation of early code reading versus testing experiments,
- the introduction of reading into NASA's cleanroom process,
- the replication of experiments and results in other groups, and
- the transfer of the results in other industries.

The results showed that reading

- can reduce failure rates by 25%,
- finds 90% of faults before testing,
- increases productivity by 30%,
- helps to better structure code in future projects, based on learning from reading,
- increases the predictability of project performance parameters such as cost, and compliance with schedules.

6 Conclusions and Outlook

In the previous chapter, we argued that empirical studies are the key to filling gaps in our knowledge of the field of software engineering. Only empirical evidence can give rise to facts and new laws. Consequently, there can be no software engineering other than empirical software engineering.

In order to address the most relevant research questions, a revision of agendas is necessary; especially

- the research agenda,
- the practical empirical agenda, i.e., we need to plan what experiments need to be performed in what contexts; ideally coordinated by a research network such as the International Software Engineering Research Network (ISERN), see [10], and
- the educational agenda.

The last item addresses the need to educate researchers for whom software engineering is naturally build upon an empirical foundation, and for whom experiments are the standard means to do research in software engineering.

Industrial organizations need to adopt long-established, well-founded engineering methods for the development of safe, secure, and trustworthy software. Concerning the project management side, they need to accept planning and working schemes, including schedules, that have been defined by experienced computer scientists and software engineers. Experience guarantees a bounded variance of software development time.

Last but not least, IT sub-organizations inside non-IT companies need to be granted a better standing, in order to function as a natural anchor for software development projects.

References

1. Stalk, G., Evans, P., Shulman, L.: Competing on Capabilities: the New Rules of Corporate Strategy. Harvard Business Review, 57–69 (1992)
2. Boehm, B., Basili, V.: Software Defect Reduction Top 10 List. Computer 34(1), 135–137 (2001)
3. The Standish Group: The Chaos Report 1994. World Wide Web (1994), `http://www.standishgroup.com/sample_research/PDFpages/chaos1994.pdf`
4. Broy, M., Rombach, D.: Software Engineering. Wurzeln, Stand und Perspectiven. Informatik Spektrum 25(6), 438–451 (2002)
5. Endres, A., Rombach, D.: A Handbook of Software and Systems Engineering. Addison-Wesley Longman, Amsterdam (2003)
6. Basili, V., Zelkowitz, M., McGarry, F., Page, J., Waligora, S., Pajerski, R.: Special Report: SEL's Software Process-Improvement Program. IEEE Software 12(6), 83–87 (1995)
7. Ren, J., Taylor, R.: Automatic and Versatile Publications Ranking for Research Institutions and Scholars. Communications of the ACM 50(6), 81–85 (2007)
8. Basili, V., Selby, R.: Comparing the Effectiveness of Software Testing Strategies. IEEE Transactions on Software Engineering 13(12), 1278–1296 (1987)
9. Basili, V., Green, S.: Software Process Evolution at the SEL. IEEE Software 11(4), 58–66 (1994)
10. ISERN: International Software Engineering Research Network. (World Wide Web), `http://isern.iese.de/network/ISERN/pub/`

Extending GQM by Argument Structures

Łukasz Cyra and Janusz Górski

Gdańsk University of Technology, Department of Software Engineering,
Narutowicza 11/12, 80-952 Gdańsk, Poland
`lukasz.cyra@eti.pg.gda.p, jango@pg.gda.pl`

Abstract. Effective methods for metrics definition are of particular importance, as measurement mechanisms are indispensable in virtually any engineering discipline. The paper describes how the well known Goal-Question-Metric (GQM) method of systematic metrics derivation from measurement goals can be extended by applying argument structures. The proposed approach is called Goal-Argument-Metric (GAM). The general ideas of GQM and GAM are briefly introduced and they are followed by the comparison of the two approaches. Next, the Trust-IT framework is described – it is used to develop argument structures in GAM. Then a case study of application of GAM is presented. The case study concerns derivation of metrics and direct measurements with the objective to assess effectiveness of Standards Conformity Framework (SCF), which is currently under development. In conclusion, early experience with GAM is presented and more information about on-going research on argument structures is given.

Keywords: GAM, GQM, Trust-IT, Trust case, Standards Conformity Framework, Measurement plan, metrics.

1 Introduction

Measurement mechanisms provide feedback that helps in evaluation of the actions undertaken and their results. However, identifying the scope of raw data to be collected and the metrics to be calculated from these data in order to achieve a particular measurement goal is a difficult and error-prone task. Selection of appropriate metrics which fit for the purpose and which do not generate unnecessary costs is a challenge. Implementation of the raw data collection process (which often needs non-trivial involvement of human effort) may be highly resource consuming. Collecting insufficient or excessive data and metrics can be frustrating and can undermine the whole measurements initiative. Therefore, it is of primary importance that measurement plans make evident the objectives and the scope of collected data and the resulting metrics.

The above problems provided strong motivation for the research towards development of effective and efficient methodologies supporting systematic metrics derivation. An example is Goal-Question-Metric (GQM) [1, 9], which is a well known methodology targeted at defining measurement plans.

In this paper we propose a modification of GQM which we call Goal-Argument-Metric (GAM). The purpose is to provide solutions to some problems arising while using GQM and in particular to provide better support for the identification and maintenance of the relationship between the measurement goals and the related metrics. The basic innovation concerns applying argument structures for stepwise refinement of the measurement goals into metrics and direct measurements and by maintaining the arguments in the easily readable and accessible form with the help of our TCT tool.

The paper first introduces GQM and GAM and compares the two approaches. Then it introduces the Trust-IT framework [3-7], which we use for expressing and maintaining argument structures in GAM. The applicability of GAM is then illustrated by a case study. In this case study we refer to the problem of assessment of the effectiveness of Standards Conformity Framework (SCF) [2, 7]. SCF, which is presently under development, is a part of Trust-IT and its objective is to support processes of achieving and assessing compliance with standards.

In conclusion we summarise our contribution and present plans for further research.

2 GQM

The Goal-Question-Metric (GQM) methodology was originally developed by V. Basili and D. Weiss and then significantly extended by D. Rombach. It is a practical methodology which helps in systematic derivation of measurement plans. GQM is well documented, for a thorough description see e.g. [1, 9]. Many other sources are also available on the Internet. The idea of GQM is graphically presented in Fig. 1.

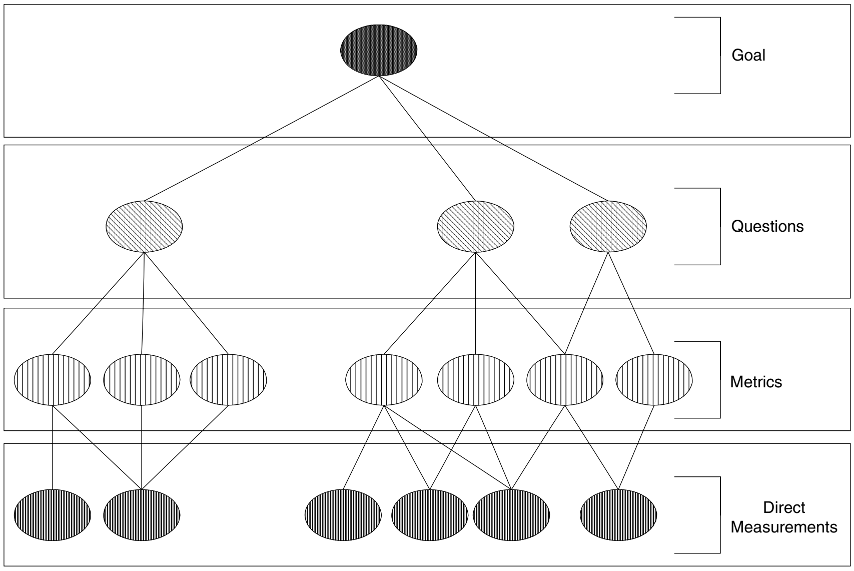


Fig. 1. GQM paradigm. Defining measurement goals (abstract level) refining them into questions (operational level), deriving metrics (quantitative information) and direct measurements.

GQM proceeds top-down, starting with the definition of an abstract measurement goal, which explicitly represents the measurement intent.

Then, referring to this goal, several questions are defined which brake the problem into more manageable chunks. The questions are defined in such a way that obtaining the answers to the questions leads to the achievement of the measurement goal. This step is the most difficult one, as deciding about the level of abstraction of the questions is by no means a trivial task. It is easy to make the questions too abstract or too detailed. In both cases, difficult problems related to identifying the relationship between the questions and the collected data and metrics, or the problems related to interpreting answers to the questions in the context of the measurement goal, may arise [9]. Therefore, a substantial experience is usually necessary before one can effectively apply GQM. This causes that the implementation of GQM requires a significant initial effort [9].

In the next step, based on the questions metrics are defined, which provide quantitative information then treated as answers to the questions. Finally, at the lowest decomposition layer, direct measurements are defined which provide the data necessary to calculate the metrics (see Fig. 1).

As the problem of defining ‘good’ questions is not easy and had no obvious solution, some additional steps have been proposed with the intention to bring more precision to GQM. For instance, templates for defining the measurement goal have been introduced and supported by different types of models providing additional explanatory information. The template requires that the goal is defined in a structured way, including: the object of study, the purpose, the quality focus, the viewpoint and the context. The structure of such a definition is as follows:

Analyze	<the object of study>
for the purpose of	< the purpose>
with respect to	<the quality focus>
from the viewpoint	<the viewpoint>
in the context of	<the context>

It has been confirmed by practical experience that the increased precision and clarity in the goal definition positively influences suitability and usefulness of the measures derived with the help of GQM [1].

Additionally, GQM can be supported by models of different types with the intention to better represent the domain knowledge. It has been suggested that descriptive, evaluation, and predictive models are applied to help in ‘grounding’ the abstract attributes, defining relationships between objects of different types, and making predictions.

GQM has evolved in time into its model-based variant, which explicitly considers models of processes and products. However, the basic idea is still the same: to derive metrics from goals using the three-step top-down procedure inspired by Fig. 1.

3 GAM

Following the main idea of GQM, GAM is a goal-oriented methodology for defining measurement plans. It differs, however, in the way the metrics and direct data measurements are derived from the goals. Instead of using partial solutions like templates

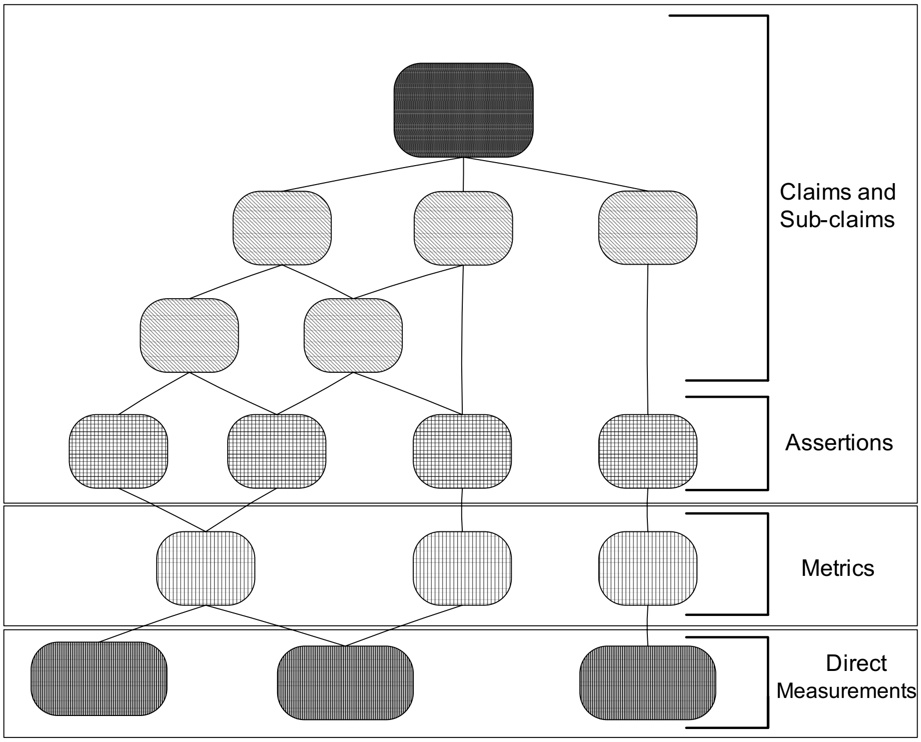


Fig. 2. GAM paradigm. Deriving direct measurements from goals by defining claims representing measurement goals, refining them into several levels of sub-claims, which can be finally argued using assertions referring to metrics.

and models, GAM provides seamless way of increasing precision in the whole process of metrics derivation.

In GAM, the goals and sub-goals are represented as claims and then the analysis focuses on identifying which data and which properties of the data (further sub-goals) are needed to demonstrate these claims.

The starting point is a claim postulating that the overall measurement goal has been achieved. Then the claim is justified by giving an argument which supports the claim. The argument can refer to other claims (about certain postulated properties) representing more manageable components of the problem. The inference rule used in the argument is stated explicitly showing the assumed argumentation strategy. If this rule is not self evident, another argument may be needed to demonstrate the validity of the inference rule. Such an argument can refer to the context of the goal (for instance, to argue the completeness of the evidence considered in the inference rule).

The procedure of decomposing claims into sub-claims is then repeated iteratively until it is possible to argue the leaf claims by directly referring to values of certain metrics. In such cases, the claims are supported by assertions on prospective values of such metrics (i.e. the assertions about metrics), as shown in Fig. 2.

In the next step, the assertions are used to build a list of metrics, which is usually a trivial task. Finally, direct measurements are derived from the metrics to define the scope of raw data to be collected.

From the argument structure which links the data and metrics with the measurement objective, it is straightforward to implement the bottom-up process that gathers the raw data (by means of direct measurements) and aggregates them into metrics. If the obtained values meet the criteria given by the assertions kept in the argument structure, the whole argument tree explicitly demonstrates that the initial goals have been met.

Fig. 2 illustrates the GAM approach in a graphical form.

4 Comparison of the Approaches

Considering the purpose and the general approach (top-down derivation and bottom-up interpretation) GQM and GAM look the same. The differences relate to the way of defining and maintaining the relationship between the measurement goals and the metrics.

The topmost claim in GAM is a direct counterpart of the measurement goal in GQM. Then, the sub-claims of GAM can be considered as answers to the questions in GQM. So it seems that both structures are still similar having counterparts of their elements: GQM is structured into layers of questions whereas GAM is structured into layers of claims with the strict correspondence between the two structures. However, this makes a significant difference, because it is easy (and natural) to link the adjacent claim levels by means of explicit arguments while it is not equally easy to identify and represent the relationship between the adjacent levels of questions. To demonstrate this difference let us consider the example represented in Fig. 3.

The example presents a refinement of the measurement goal which is to assess the support provided by a tool X. In case of GQM, a set of questions is defined with the intention to cover all the aspects related to analysis of the support provided by tool X. Identification of such questions is not an easy task and the analyst has to constantly control the scope of the analysis. By contrast, in case of GAM the focus is on finding an argumentation strategy which demonstrates the adequate support offered by tool X. In the example, the strategy is by considering different application scenarios for X. Once the strategy has been chosen, the refinement into sub-claims is a natural consequence.

In GQM, choosing an appropriate level of abstraction for the questions is (according to [9]) a difficult task and a substantial experience in application of the method is needed. It is possible to use more than one level of questions to make the definition of "proper" questions easier. The relationship between the adjacent levels of questions is not explicit, which increases the difficulty in using the method. Some approaches to deal with this difficulty have been proposed, for instance, the interpretation models of different types [1].

GAM admits multiple levels of claims and does not restrict the user in this respect. At each level, the problem is broken into more manageable sub-problems and the relationship between the adjacent levels is explicitly established by giving the

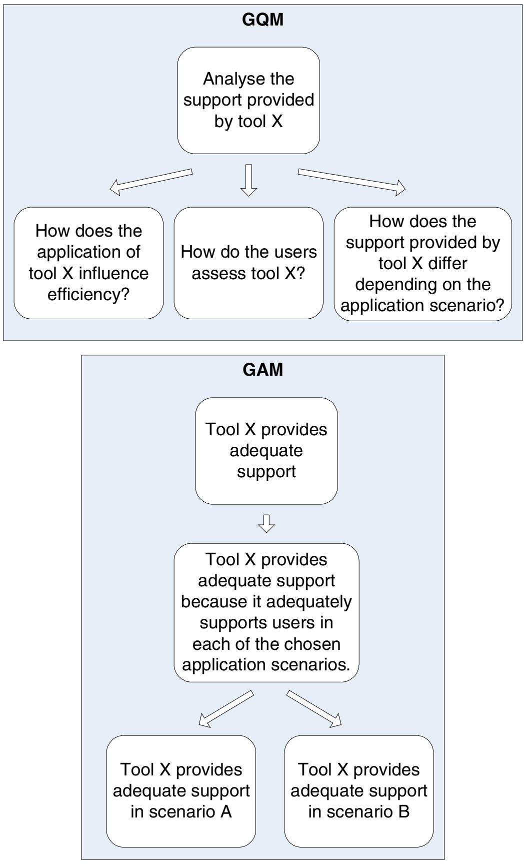

GQM
Analyse the support provided by tool X
How does the application of tool X influence efficiency?
How do the users assess tool X?
How does the support provided by tool X differ depending on the application scenario?
GAM
Tool X provides adequate support
Tool X provides adequate support because it adequately supports users in each of the chosen application scenarios.
Tool X provides adequate support in scenario A
Tool X provides adequate support in scenario B

Fig. 3. Comparison of GQM and GAM. Defining a set of questions on the basis of the measurement goal and a set of sub-claims demonstrating the root claim.

corresponding argument. The subsequent abstraction layers result naturally from the task of justifying the higher level claims by referring to the lower level ones. In order to create sound warrants for the arguments, the user of GAM is usually forced to refer to the information that is represented in the models foreseen in GQM, however, in this case it is simply a part of the argument development process and the scope of this information is easily controlled.

In our assessment, the most significant advantage of GAM is the introduction of argument strategies and warrants, which support the arguments. Considering what is necessary to justify a claim, finding an appropriate argumentation strategy and documenting those decisions provides for focusing the scope of the analysis and traceability of the results. Arguments make it evident whether the sub-components are necessary to support the claim and whether the decomposition is complete. Therefore, the questions like: ‘Is it a complete set of questions which must be taken into account?’ or ‘Do I really need this question to support the goal?’ do not appear in GAM.

Both methods are supported by advanced tools. For instance, in [8] a tool supporting GQM has been described. GAM is fully supported by the TCT tool [10] which is part of the Trust-IT framework. This framework is described in more detail in the subsequent sections.

5 Trust Cases

Development of arguments in GAM follows the approach defined in the Trust-IT framework. A part of the Trust-IT framework is the trust case language which provides means of expressing arguments [3, 4, 5, 7]. Similar languages have been used in the safety critical systems domain to express 'safety cases' – arguments justifying that a given system is adequately safe while considered in its target context. Trust cases differ from safety cases in several respects, for instance they can address broader (practically unlimited) scope of properties and do not have any particular restrictions on their structure and contents. We have already applied trust cases to analyse and justify different properties, including safety, security, privacy and others. Another interesting area of application of trust cases is assessing and demonstrating the compliance with standards, which we are presently investigating.

In GAM we represent argument structures as trust cases. Flexibility of the language and legibility of the arguments are two important factors which influenced this decision. Additional advantage is that trust cases are supported by an efficient Internet-enabled tool [10] which supports management and sharing of trust case structures.

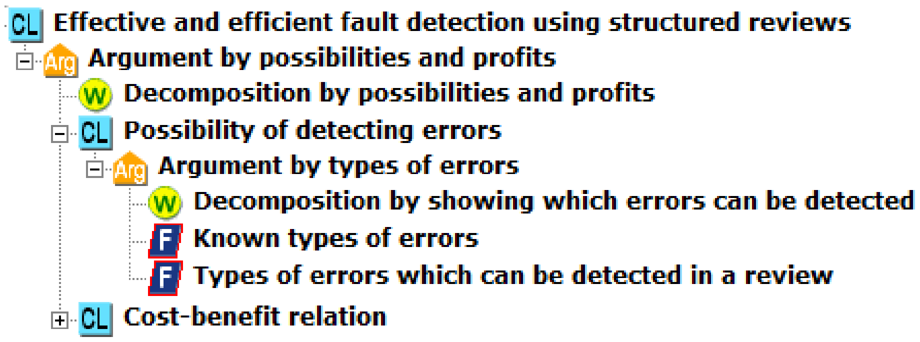
Trust cases are composed of nodes of different types. The type of a node represents its role in demonstrating a certain statement. The basic logical component of a trust case is an argument composed of a *claim* to be justified (denoted CL), *evidence* supporting the claim and an *inference rule* which shows how, on the basis of the evidence, the claimed property is achieved.

The evidence and the claim are connected using nodes of type *argument* (denoted Arg), which state the argumentation strategy. Apart from arguments also *counter-arguments* (denoted Arg) can be used. Instead of the argument which refers to the evidence supporting the stated claim, counter-arguments demonstrate that the claim is not true. They can be used to derive metrics from counter-claims in GAM.

The inference rule is represented as a node of type *warrant* (denoted W). The warrant demonstrates in detail the argumentation strategy and justify why the inference rule used is valid. *Assumption* nodes are also possible but they are omitted in the description as they are not used in GAM.

Finally, the evidence can be of type: claim or *fact* (denoted F). Facts contain information which does not need additional justification (because it is obvious) or information whose validity is demonstrated in external documents. In contrary, claims must be demonstrated by other arguments. This way (by justifying claims) a trust case develops into a tree structure composed of many levels of abstraction. An example is given in Fig. 4.

Facts which are based on information contained in external documents can be supported by a node of type reference (denoted Ref). Such nodes contain information about the location of documents (usually it is a URL).



- Effective and efficient fault detection using structured reviews
  - Argument by possibilities and profits
    - Decomposition by possibilities and profits
    - Possibility of detecting errors
      - Argument by types of errors
        - Decomposition by showing which errors can be detected
        - Known types of errors
        - Types of errors which can be detected in a review
    - Cost-benefit relation

Fig. 4. Trust case example. Demonstrating structured reviews effectiveness and efficiency by showing that it is possible to detect errors of different types and that the benefits outstrip the cost.

Additionally, anywhere in the argument tree an *information* node (denoted) can be placed. Such nodes contain explanatory information which does not constitute a part of the proper argument.

Each of the above-mentioned nodes can be represented as a *link* to specific part of the trust case (if this part is to be re-used). Depending on where the link points at, it is represented by , , , , , or .

6 Case Study: Overview of the Problem

In the case study we aimed at deriving metrics and direct measurements for the assessment of the effectiveness of *Standards Conformity Framework* (SCF) [2, 7].

SCF itself is part of the (broader) Trust-IT framework. SCF supports application of standards at the stages of achieving, assessing and maintaining compliance. The framework provides mechanisms which help to gather the evidence and present it in a legible way. The central component of SCF is a *Trust Case template* - a data structure derived from a given standard. Templates also include extra-standard data sources like guides, historical data, experts' knowledge, results of standards analyses and so on. All this information is kept in one electronic document. Such documents can be further assessed by auditors and if accepted, can be reused in many standards' compliance projects. SCF is supported by an on-line tool which enables teamwork while producing, gathering, and structuring the evidence which demonstrates the compliance with a standard.

SCF has already been used in some projects and the results are promising. To provide for a more objective assessment, a research program was initiated targeted at better understanding the benefits resulting from the framework application. As the overall goal, the analysis of the SCF's effectiveness was selected.

The objective was to derive metrics and direct measurements which could then be used in experiments to gather the necessary data needed to assess and demonstrate the effectiveness of SCF. The identified scope of direct measurements is going to be used while planning for a series of experiments targeting at the assessment of the effectiveness of the SCF framework.

Initially, we applied GQM to identify the scope of data to be gathered and the scope of metrics to be constructed from those data. The results were, however, not satisfactory although we had run two iterations of the GQM process to find the appropriate set of metrics. The major problems encountered were related to the derivation tree complexity and the scope management.

The complexity of the GQM tree resulted from the complexity of the problem itself (objective reason) and from the difficulties in defining the scope of data to be gathered (subjective reason). The scope of possible questions to be considered and possible metrics to provide answers to those questions was particularly broad also because we had to consider different variants. Therefore, deciding if a given question is beyond the scope or if the whole set of questions is complete was particularly difficult. In addition, while planning for the data gathering experiments it was often difficult to assess how a given data item influences the result of the measurement program.

The above difficulties led to the decision of applying argument structures to better control the relationship between the measurement objective, the metrics and data collection. Trust cases and the TCT supporting tool were chosen as the way to represent and maintain the argument structures.

7 Case Study: Application of GAM

To support derivation of metrics and measurements we created a trust case template of appropriate structure (see Fig. 5). It represents the whole measurement plan and is composed of four branches:

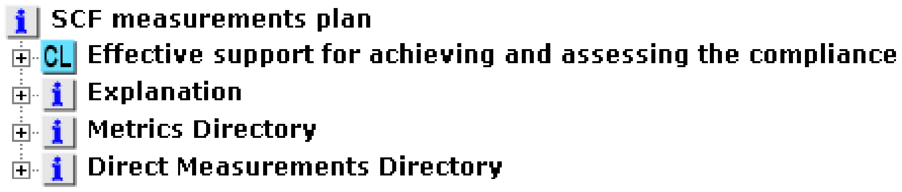
(1) '*Effective support for achieving and assessing the compliance*' is the top most claim (representing the measurement goal) which contains the whole argument structure. This claim is to be supported by the argument which justifies it (not shown in Fig. 5).

(2) *'Explanation'* contains additional information like the definitions of terms used to describe metrics and measurements.

(3) *'Metrics Directory'* is the list of all metrics derived from the measurement goal.

(4) *'Direct Measurements Directory'* contains the list of all direct measurements derived from the metrics.

In the next step, the argument structure was developed. The measurement goal was decomposed into three claims and a warrant which describes the inference rule used. This is presented in Fig. 6.



SCF measurements plan
- CL Effective support for achieving and assessing the compliance
- i Explanation
- i Metrics Directory
- i Direct Measurements Directory

Fig. 5. SCF measurement plan trust case - a tree composed of the argument structure, explanations, a list of metrics and a list of direct measurements

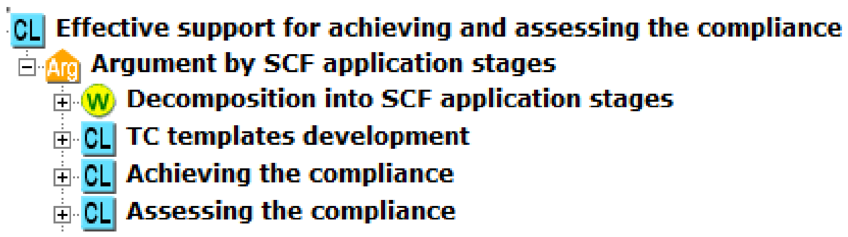

CL Effective support for achieving and assessing the compliance
- Arg Argument by SCF application stages
  - W Decomposition into SCF application stages
  - CL TC templates development
  - CL Achieving the compliance
  - CL Assessing the compliance

Fig. 6. First level of decomposition. Arguing SCF effectiveness by demonstrating possibility of developing sound TC templates which positively influence the process of achieving the compliance and increase efficiency of the assessment.

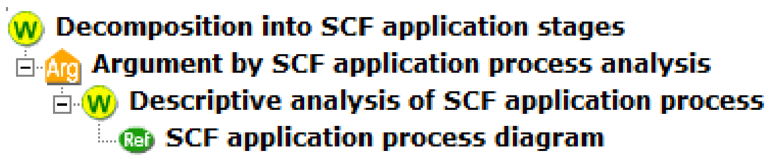

W Decomposition into SCF application stages
- Arg Argument by SCF application process analysis
  - W Descriptive analysis of SCF application process
    - Ref SCF application process diagram

Fig. 7. Argument supporting a warrant. Arguing that SCF effectiveness can be demonstrated by demonstrating three claims related to: development of templates, application of SCF at the stage of achieving the compliance, and application of SCF at the stage of assessing the compliance by the detailed analysis of the SCF application process.

In Fig. 6, it is argued that *'Effective support for achieving and assessing the compliance'* is provided because it is possible to create sound templates (represented by the claim *'TC templates development'*), application of the templates positively influences the resulting level of compliance (represented by the claim *'Achieving the compliance'*) and the performance of assessing the compliance is significantly improved (represented by the claim *'Assessing the compliance'*).

The decomposition of the argument is justified by the '*Decomposition into SCF application stages'* warrant which is further refined in Fig. 7.

The lower warrant (shown in Fig. 7) refers to the SCF application process structure and recalls the process diagram (through the link '*SCF application process diagram*'). The analysis of this process (included in the body of the warrant '*Descriptive analysis of SCF application process*') explains why the structure shown in Fig. 6 is sufficient to assess effectiveness of SCF.

In the same way the three claims represented in Fig. 6 were decomposed into more refined claims and justified by more refined arguments. Each time, appropriate warrants were provided constraining the scope and giving the reason for decisions.

Finally, at a certain level of abstraction, to justify the higher-level claim it was enough to directly refer to measurable properties. At that level, the decomposition process stops. This last step is illustrated in Fig. 8.

Each leaf of the argument structure refers to a metric. The metric represents a measurable value having a certain business meaning, which can be an aggregation of a few measurements. In this way the method supports definition of the most suitable metrics. Additionally, the leaves contain assertions which impose constraints on values

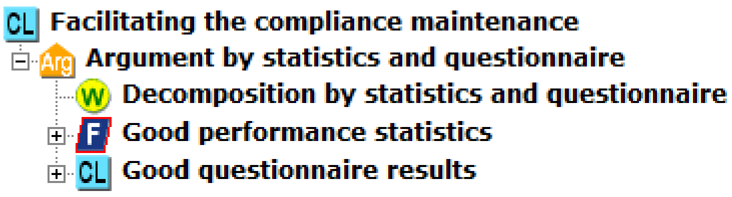

CL Facilitating the compliance maintenance
Arg Argument by statistics and questionnaire
W Decomposition by statistics and questionnaire
F Good performance statistics
CL Good questionnaire results

Fig. 8. Introduction of assertions in the argument structure. Demonstrating that compliance maintenance is facilitated by SCF because the statistics show improvement in the performance, and subjective opinions stated in questionnaires were positive.

of the metrics. An assertion states that a given metric *m* is in a certain subset of possible values *A* as shown in (1).

$$\begin{aligned} & M \in A; \textit{ where} \\ & \quad M \textit{ is a metric} \\ & \quad A \textit{ is a subset of possible values of } m \end{aligned} \tag{1}$$

For instance, in Fig. 8 the claim *'Facilitating the compliance maintenance'* postulating that SCF facilitates the compliance maintenance is argued using performance statistics and questionnaire results. The claim *'Good questionnaire results'* is decomposed further giving assessment criteria for answers to particular questions (explaining what 'good' means in this context). The fact *'Good performance statistics'* is directly connected with a metric. It states that dealing with a change takes less than 90% of time needed if SCF were not used.

Fig. 8. also shows that at the same abstraction level it is possible to have claims and facts simultaneously. This gives flexibility in structuring the argumentation tree according to the needs.

Finally, all the claims must be refined into assertions. The number of levels of abstraction is dictated by the problem itself. At the bottom of the argument structure we will find claims which are supported by assertions only. (See Fig. 9)

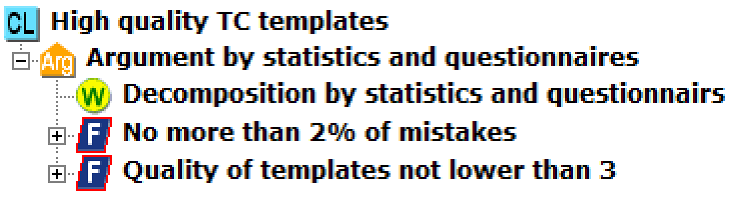
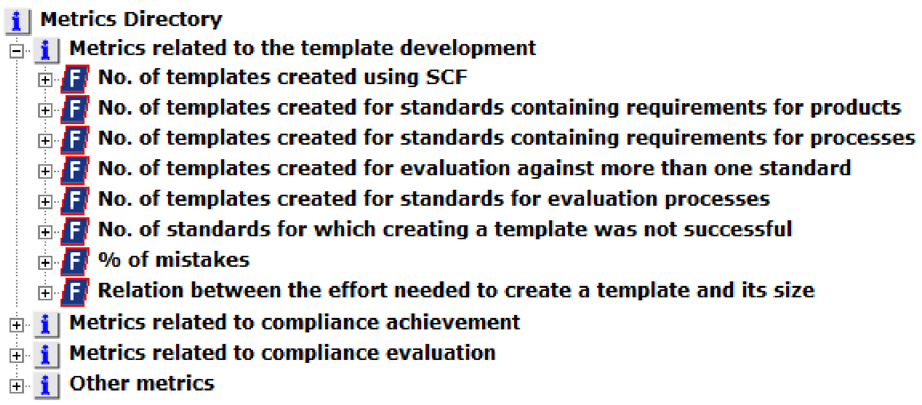

CL High quality TC templates
Arg Argument by statistics and questionnaires
W Decomposition by statistics and questionnairs
F No more than 2% of mistakes
F Quality of templates not lower than 3

Fig. 9. Claim demonstrated by assertions only. Arguing high quality of TC templates by showing the statistics about mistakes and presenting results of questionnaires.

In the example above, the claim *'High quality TC templates'* demonstrates that the templates developed according to the procedures defined by SCF are of high quality. It is justified by the requirement that the number of mistakes reported relates to less than 2% of requirements of the standard (represented as fact *'No more than 2% of mistakes'*) and the result of questionnaires used to assess the quality of templates generated



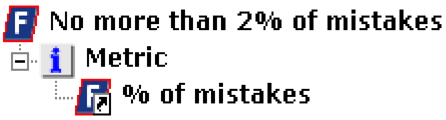
Metrics Directory
- Metrics related to the template development
  - No. of templates created using SCF
  - No. of templates created for standards containing requirements for products
  - No. of templates created for standards containing requirements for processes
  - No. of templates created for evaluation against more than one standard
  - No. of templates created for standards for evaluation processes
  - No. of standards for which creating a template was not successful
  - % of mistakes
  - Relation between the effort needed to create a template and its size
- Metrics related to compliance achievement
- Metrics related to compliance evaluation
- Other metrics

Fig. 10. Metrics directory

with the help of SCF is at least 3 in the (1,..,5) scale (this is represented by fact *'Quality of templates not lower than 3'*).

In the next step the assertions were used to derive metrics. All the identified metrics were collected as facts in the *'Metrics Directory'* branch shown in Fig. 10.

To provide for traceability in both directions (i.e. from assertions to metrics and form metrics to assertions) under every assertion an information node is added which contains a link to the metric used by the assertion (as in Fig. 11).



No more than 2% of mistakes
- Metric
  - % of mistakes

Fig. 11. Binding assertions and metrics. An assertion and a link to the metric derived from the assertion.

Let us consider the assertion shown in Fig. 11. It refers to a metric representing the per cent of mistakes in descriptions of requirements contained in templates. To construct such a metric we need raw data (a direct measurement). In general, a given metric *M* can be treated as a function *f( )* of several direct measurements dm_i as described in (2).

$$\begin{aligned} M = f(dm_i);\ & where \\ & dm_i \text{ is a direct measurement} \\ & f \text{ is a function} \end{aligned} \tag{2}$$

For instance, Fig. 12 gives an example metric and the related direct measurements.

The metric representing the number of mistakes in a template can be obtained using two direct measurements: one assessing the size of a template (represented as a link to fact *'Size of the template'*) and another one, assessing the number of mistakes

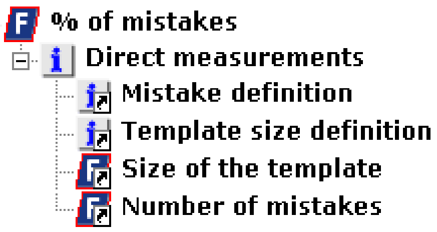


Fig. 12. Binding metrics and measurements. A metric, links to the direct measurements derived from the assertion and definitions of the notions needed to precisely express the metric.

in a template (represented as a link to fact *'Number of mistakes'*). Additionally, two links to information nodes, which contain the definitions of the notions used (the definitions of 'mistake' and 'template size') were added. The list of all the direct measurements is located in the branch *'Direct Measurements Directory'* of Fig. 5.

8 Summary

In the paper the GAM method of systematic derivation of metrics and measurements from measurement goals was presented. The method was compared with GQM, one of the most popular methods of this type. In addition, a case study of application of GAM was described in detail, showing its most crucial aspects.

Application of GAM led to satisfactory results and removed the difficulties we have faced while applying GQM. The method proved to be more effective while solving this particular problem. The initial investment in development of GQM tree took about 24 hours in each of the two iterations. By contrast, application of GAM required only 10 hours[1].

The authors are fully aware that a single case study is not enough to draw more general conclusions related to comparison of the two methods. However, the results obtained are very encouraging and GAM is ready to use together with its supporting tool. We are planning for more case studies to provide more evidence on effectiveness of the method.

The results presented in this paper have been achieved in the context of the broader research program related to application of argument structures in various contexts. Except measurement plans, trust cases have been already applied to argue safety, security and privacy of e-health services and to support application of security standards. The method is supported by a matured, ready to use tool, which has already been used in a few projects e.g. EU 6th Framework Integrated Project PIPS and EU 6th Framework STREP ANGEL. The tool provides effective means of editing the argumentation trees diminishing the difficulties related to maintenance, complexity and change management.

[1] It is worth mentioning however, that the GQM analysis was performed by a person without much prior experience with the method, and the GAM method was applied by a person having already some experience with the Trust-IT framework.

References

1. Briand, L.C., Differing, C., Rombach, H.D.: Practical Guidelines for Measurement-Based Process Improvement, Software Process. Improvement and Practice, No. 4 (1996)
2. Cyra, Ł., Górski, J.: Supporting compliance with safety standards by trust case templates. In: Proceedings of ESREL 2007 (2007)
3. Górski, J., et al.: Trust case: justifying trust in IT solution, Reliability Engineering and System Safety, vol. 89, pp. 33–47. Elsevier, Amsterdam (2005)
4. Górski, J.: Trust Case – a case for trustworthiness of IT infrastructures. In: Kowalik, J., Gorski, J., Sachenko, A. (eds.) Cyberspace Security and Defense: Research Issues. NATO ARW Series, pp. 125–142. Springer, Heidelberg (2005)
5. Górski, J.: Collaborative approach to trustworthiness of infrastructures. In: Proceedings of IEEE International Conference of Technologies for Homeland Security and Safety, TEHOSS 2005, pp. 137–142 (2005)
6. Górski, J.: Trust-IT – a framework for trust cases, Workshop on Assurance Cases for Security - The Metrics Challenge. In: DSN 2007 The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Edinburgh, UK, June 25 – June 28 (2007)
7. IAG, Information Assurance Group Homepage, `http://iag.pg.gda.pl/iag/`
8. Lavazza, L.: Providing Automated Support for the GQM Measurement Process. IEEE Software 17(3), 56–62 (2000)
9. van Solingen, R., Berghout, E.: The Goal/Question/Metric Method: a Practical Guide for Quality Improvement of Software Development. Cambridge University Press, Cambridge (1999)
10. TCT User Manual, Information Assurance Group, Gdansk University of Technology (2006)

On Metamodel-Based Design of Software Metrics

Erki Eessaar

Department of Informatics, Tallinn University of Technology,
Raja 15, 12618 Tallinn, Estonia
eessaar@staff.ttu.ee

Abstract. Metric values can be used in order to compare and evaluate software entities, find defects, and predict quality. For some programming languages there are much more known metrics than for others. It would be helpful, if one could use existing metrics in order to find candidates for new metrics. A solution is based on an observation that it is possible to specify abstract syntax of a language by using a metamodel. In the paper a metrics development method is proposed that uses metamodel-based translation. In addition, a metamodel of a language helps us to find the extent of a set of metrics in terms of that language. That allows us to evaluate the extent of the core of a language and to detect possible quality problems of a set of metrics. The paper contains examples of some candidate metrics for object-relational database design, which have been derived from existing metrics.

Keywords: Metric, Measure, Metamodel, UML, Object-relational database, Data model, Reusability.

1 Introduction

Metrics, the values of which characterize software designs, can be used in order to compare designs, find defects, and predict quality. For example, Choinzon and Ueda [1] refer to 22 *object-oriented design* metrics that are presented in the research literature. In addition, they define 18 new design metrics. There are fewer metrics that allow us to evaluate database designs. For example, Piattini et al. [2] present three table oriented metrics for *relational* databases. Piattini et al. [3] present twelve metrics that help us to evaluate the design of an *object-relational* database. Muller [4] proposes to evaluate structural cohesion of tables based on their normal forms.

Metamodeling is a well-known activity in software engineering that allows us to specify abstract syntax of a language. Seidewitz [5] writes that a metamodel "*makes statements about what can be expressed in the valid models of a certain modeling language.*" If we use UML as a metamodeling language, then language elements and their relationships are presented by using classes/attributes (properties) and attributes/relationships, respectively [6]. Is it possible to use metamodels in order to create and improve metrics? McQuillan and Power [7] write that definitions of metrics should be *reusable*. Researchers have used *metamodels* and *ontologies* in order to *present* object-oriented design metrics [8] and database design metrics [9], respectively, as precisely as possible. For example, SQL:2003 [10] is a large international standard that specifies the database programming language SQL. An SQL:2003 ontology [9] is presented by using UML. The ontology resembles a metamodel. A difference with a metamodel of SQL:2003 is that the ontology follows the *principle of minimal ontological commitment* [11] and therefore covers *only* the most important parts of SQL:2003 instead of specifying the entire language.

Baroni et al. [12] think that an SQL:2003 ontology, which is a step towards a complete SQL:2003 metamodel, helps us to prevent ambiguity in metrics specifications and automate the collection process of metrics values.

In this paper, we propose *additional* means for using metamodels in the development of metrics. We assume that metrics that belong to a set M help us to evaluate software entities that are created by using a language L. Models, patterns, and fragments of code are examples of software entities.

The *first goal* of the paper is to propose a *metamodel-based* method for creating *candidate* metrics. This novel method uses a metamodel-based translation and allows us to *reuse* existing metrics specifications. Such method could be used in case of any software development language if a metamodel of the language is available.

The *second goal* of the paper is to propose a *metamodel-based* method for calculating the *extent* of a set of metrics M in terms of a language L. This method allows us to find concrete numerical estimates of the size of the core of L as the designers of metrics see it. A small extent of M is a sign of *possible* quality problems of M because M may be incomplete. For example, McQuillan and Power [7] note that existing UML metrics deal only with a small part of all the possible UML diagram types. The existing metrics evaluation methods [3, 13, 14] do not take into account whether all the metrics, which belong to a set of related metrics, together help us to evaluate all (or at least most of the) parts of a software entity.

The data model, based on which a database system (DBMS) is implemented, is a kind of abstract language [15]. In this work, we investigate two *object-relational* data model approaches as the examples:

1. The underlying data model of SQL:2003 (OR_{SQL}) [10].
2. The underlying data model of The Third Manifesto (OR_{TTM}) [16].

We have found few metrics about OR_{SQL} database design and no metrics about OR_{TTM} database design.

The *third goal* of the paper is to use the proposed metamodel-based methods in order to evaluate the existing OR_{SQL} database design metrics and to show how to develop an OR_{TTM} database design metric based on an OR_{SQL} database design metric.

The rest of the paper is organized as follows. Section 2 analyzes how we can use metamodels of languages in order to create and evaluate metrics. In Section 3, we present examples. Firstly, we evaluate some OR_{SQL} database design metrics in terms of an OR_{SQL} metamodel. Secondly, we design some candidate OR_{TTM} database design metrics based on a set of OR_{SQL} database design metrics. Thirdly, we find the extent of some sets of metrics. Finally, Section 4 summarizes the paper.

2 On Using Metamodels in the Development of Metrics

Piattini et al. [3] and IEEE Standard for a Software Quality Metrics Methodology [17] describe frameworks of metrics development. They do not propose the *reuse* of existing metrics as one possible method how to find *candidate metrics*. A candidate metric is a metric that has not yet been approved or rejected by experts.

We think that it is not always necessary to start development of a metric from scratch. Instead, we could try to reuse existing metrics. The *motivation* of this approach is that it allows us to create quickly candidate metrics and experiment with them in order to improve our understanding of a domain and get new ideas. In addition, candidate metrics provide a communication basis for discussions among all groups that are involved in the development of a new set of metrics. It is possible that a candidate metric evolves and becomes accepted and validated metric or the candidate metric is rejected after evaluation. The proposed approach should *complement* existing methods of metrics development but not replace them. Figure 1 presents the concepts that are used in the proposed approach and their interconnections.

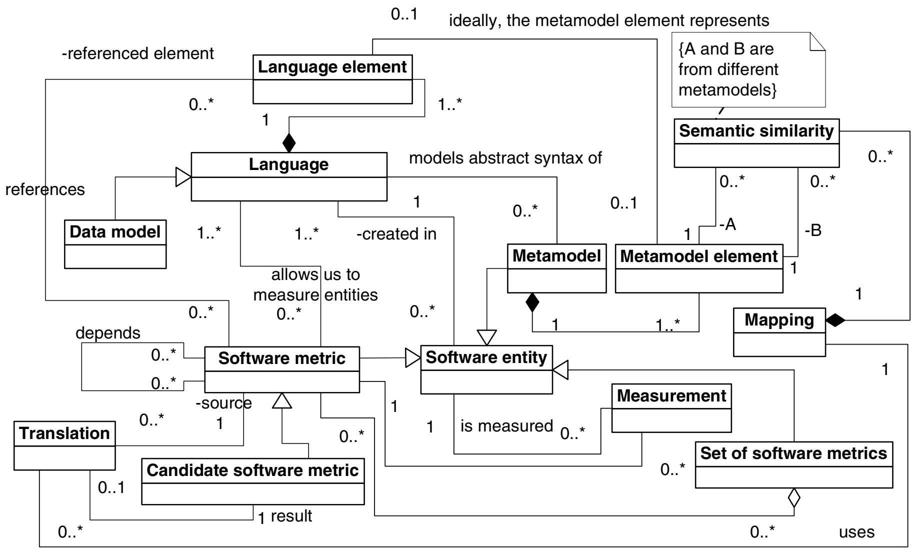


Fig. 1. A domain model of the proposed approach

Each language consists of one or more language elements. It is possible to represent abstract syntax of a language by using a metamodel. A language can have different metamodels, which are for instance created by different parties or are presented with the help of different languages. A metamodel, a software metric, and a set of software metrics are examples of software entities. Each software entity is created by using one or more languages. For example, a metamodel of UML [18] consists of UML diagrams, OCL expressions, and free-form English text. Another example is that OR$_{SQL}$ metrics [9] are presented by using OCL expressions and free-form English text. A language can have associated metrics that can be used in order to measure properties of the software entities that are created by using this language. Each metamodel consists of one or more metamodel elements. There could exist mappings between elements of different metamodels that allow us to create candidate metrics by using metamodel-based translation. A metric could be calculated based on values of other metrics.

Let us assume that the metrics that belong to a set M help us to evaluate software entities that are created by using a language L. Let us also assume that there is a language L', the corresponding metrics of which belong to a set M'. All the methods that are proposed in this paper require the existence of the *metamodels* of L and L' and also the existence of *mapping* of elements of L and L' metamodels. If UML is used in order to create these metamodels, then the elements that must participate in the mapping are *classes*. For example, a metamodel of UML [18] contains classes like “class”, “action”, and “actor” and a metamodel of OR_{SQL} [19] contains classes like “data type”, “constructed data type”, and “data type constructor”. We follow the example of Opdahl and Henderson-Sellers [20], who evaluate a language based on classes of a metamodel and do not use a mapping between relationships and a mapping between attributes.

A pair of elements (that are from the different metamodels) exists in the mapping if the constructs behind these elements have exactly the same semantics or they are semantically quite similar. Designers of L and L' and users of both these languages are the experts who are the best suited to decide whether the *semantic similarity* of the underlying constructs of two elements is big enough in order to place a pair of these elements into the mapping or not. Ideally, these mappings should be *standardized*.

The use of mapping of elements of metamodels in order to evaluate languages or translate models is not a new idea. However, we use this approach in a *new context*. For example, ontological evaluation of a language is a comparison of the concrete classes of a language metamodel (language constructs) with the concepts of an ontology in order to find ontological discrepancies: construct overload, construct redundancy, construct excess, and construct deficit [20]. Opdahl and Henderson-Sellers [20] use UML metamodel in order to perform an ontological evaluation of UML by comparing it with Bunge–Wand–Weber (BWW) model of information systems. Researchers have proposed metamodel-based comparison of ontologies [21]. It is also possible to compare two languages by using their metamodels. For example, researchers have proposed metamodel-based comparison of data models [19, 22]. The work of Levendovszky et al. [23] is an example of study about metamodel-based model transformations from one language to another.

2.1 New Means of Using Metamodels in Metrics Development

In this section, we present some new means of using metamodels of languages L and L' in order to create and improve metrics that belong to the sets M and M', respectively. We will present examples of the use of these means in Section 3.

1. A metamodel of L helps us to find shortcomings in the specification of individual metrics that belong to M. We have to make sure that all the language elements that are referenced in a specification of a metric (the set of these language elements is X) have a corresponding element in a metamodel of L (the set of all the metamodel elements is Y; there is a total injective function $f: X \rightarrow Y$) and X is the same in all the different specifications of the same metric. If these conditions are not fulfilled, then

it shows us that the wording of a metric may not be precise enough. The result of this investigation could be improved wording of the specifications of metrics or the creation of new candidate metrics.

2. It is possible to develop candidate metrics for L' (that belong to M') by translating metrics that belong to M. This translation is based on a mapping of elements of metamodels of languages L and L'.
3. A metamodel of L helps us to evaluate the extent of M and find the elements of L that are not covered by M. This *may* lead to the creation of new candidate metrics.

The quality of the results of the use of these means depends on the quality of a metamodel. For example, if a metric refers to a language element *l* (that belongs to L) but a metamodel of L has no element that represents *l*, then we will erroneously conclude that the metric is imprecise (see the first mean) because an element of X has no corresponding element in Y. This example stresses an importance of evaluation and standardization of metamodels.

2.1.1 Metamodel-Based Creation of a Candidate Metric

Let us assume that we want to translate a metric *m* from a set M in order to use it in case of software entities that are created by using L'. Next, we propose a method that allows us to develop metrics by using a *metamodel-based translation*:

1. Extract *nouns* from the text of a specification of *m*.
2. Find all the elements of L metamodel that correspond to the nouns that are found during step 1. It allows us to find language elements, based on which a value of *m* is calculated.
3. For each element of L metamodel that is found during step 2, find a corresponding element of L' metamodel. *Discrepancies* of the metamodels will cause some problems:
   - If an element of L metamodel has more than one corresponding element of L' metamodel, then it is not possible to perform *automatic* translation and a human expert has to choose *one* corresponding element of L' metamodel.
   - If at least one of the found elements (see step 2) of L metamodel does not have a corresponding metamodel element of L' (there is a *construct deficit* in L'), then it is not possible to perform *automatic* translation. A human expert has to investigate whether it is possible to use any metamodel element of L'. If it is not possible, then the process finishes. As you can see, the bigger are the discrepancies between two languages, the harder it is to translate a metric.
4. In case of each element of L' metamodel (each class in case of UML) that is found during step 3, check whether it is part of a specialization hierarchy.
   - If a metamodel element *e'* is part of a specialization hierarchy, then a human user has to evaluate whether it is *instead* possible to use some direct or indirect *supertype* of *e'* in order to construct a metric for L'. If the use of a supertype is reasonable, then a metrics designer has to use this supertype *instead* of *e'* in order to construct a new metric. It ensures that this new metric can be used in as many cases as possible.
5. Use the names of all the selected metamodel elements of L' (from step 3, 4) in order to construct a candidate metric *m'*.
   - "Initialism is an abbreviations formed from initial letters" [24]. If *m* has an initialism, then create an initialism of *m'* based on *m*. Firstly, we have to identify a

phrase or name based on which an initialism of *m* is created. Secondly, we have to find the corresponding name or phrase in *m'*. Finally, we have to use initial letters of words in this phrase or name in *m'* in order to construct an initialism to *m'*.

6. Validate the new candidate metric *m'* formally and empirically in order to accept or reject it. The validation procedure is not the subject of this paper. However, there are already a lot of studies about evaluation of metrics [3, 13, 14].

If a metric *m* is a derived metric, the value of which is calculated based on the values of a set of metrics, then we firstly have to translate metrics that belong to this set before we can translate *m*.

For instance, the proposed method could be used in order to translate metrics of UML models [25] or OR_{SQL} database designs to metrics that could be used in case of Object-Process models [26] or OR_{TTM} database design, respectively. This would allow us to quickly find some metrics and to start their evaluation.

A poblem is that if the quality of an initial metric is low, then the quality of a resulting metric will also be low. If an initial metric has associated tresholds of undesirable values [1], then we cannot use them in case of a new metric, without extensive testing. It is also possible that a new metric will become less important than the original, because languages L and L' could pay attention to different things and hence different parts of these languages are important to the designers. A new metric might be about relatively unimportant part.

The existence of this method makes it possible to at least partially automate translation of metrics. It is not possible to fully automate it because sometimes a human expert has to make decisions (see steps 3, 4, 6).

2.1.2 Metamodel-Based Calculation of the Extent of a Set of Metrics

It is possible that some elements of a language L are not taken into account by *any* metric in M. The percentage of the metamodel elements that are covered by at least one metric in M shows us the *extent* of M in terms of L. The extent of M (we denote it E(M)) is a candidate *metric* that helps us to evaluate M in terms of *completeness*. E(M) value is a percentage. The bigger the value is, the more complete is M.

More precisely, let us assume that we use UML in order to create metamodels. If we calculate the value of E(M), then we have to take into account a mapping MA between metrics that belong to M and classes in a metamodel of L. MA contains a pair of a metric *m* and a class *c*, if the calculation formula of *m* takes into account a language element that is presented by *c*.

We can calculate E(M) based on the formula (1) where:

- a is the *total number* of different classes of a metamodel of L, which participate in at least one pair in MA, and their direct or indirect subclasses. We should not count any class more than once. For example, if two classes in the mapping have the same subclass, then we have to count this subclass only once.
- b is the *total number* of all classes in a metamodel of L.

$$E(M) = a*100/b \,. \tag{1}$$

All the elements of a metamodel of L that do not participate in any pair in MA represent the parts of L that are not covered by the metrics in M.

We try to measure *completness* of a set of metrics by using this metric. We have to use *matching* of metrics and metamodel elements and *counting* of matches and metamodel elements in order to calculate this metric. This metric can be used within and across projects and workgroups that deal with the development of metrics or decide the use of particular metrics in a particular project.

Firstly, if we assume that metrics should pay attention only to the most important elements of L, then E(M) shows us the extent of the *core* of L as the designers of metrics see it. If this core is small, then it raises a question whether L containts unnecessary elements. If a set of metrics M has small E(M) value, then it does not *necessarily* mean that this set has quality problems. Different parts of a language could contribute differently to the overall quality of a software entity, that is created by using L. However, a small E(M) value points to the *possible* quality problems of M, because M might be incomplete and therefore additional investigation is needed.

A language could have more than one metamodel. For example, they could be created by different parties or by using different languages. It is possible, that:

1. Different metamodels specify different sets of language elements. For instance, CIM (Common Information Model) is a conceptual information model that specifies different areas of information technology management. Part of CIM Database Model [27] is a model of SQL Schema. It presents only eight classes that correspond to the constructs that are specified in the SQL standard [10]. On the other hand, the OR_{SQL} metamodel [19] contains 110 classes.
2. In one metamodel a relationship between language elements is presented with the help of an association class but in another metamodel by using an association. For instance, Baroni et al. [12] use associations in order to model relationships between classes *Referential constraint* and *Column*. On the other hand, the OR_{SQL} metamodel [22] contains association classes *Referencing column* and *Referenced column* in order to specify these relationships.
3. In one metamodel a language element is presented with the help of an attribute but in another metamodel by using a class. For instance, CIM Database Model [27] contains class *SqlDomain* that has attribute *DataType*. There is no separate class *DataType* in CIM Database Model. On the other hand, *Data type* is a separate class in the OR_{SQL} metamodel [22].

There could also be similar differences between different versions of the same metamodel. Therefore, we can find different E(M) value for the same set of metrics if we use different metamodels. It means that each E(M) value should always be accompanied with the information about the metamodel (including its version) based on which it is calculated. If a language has more than one set of metrics and we want to compare these sets in terms of E(M), then we have to use the same metamodel version in order to calculate E(M) values.

A possible negative side efect of the use of this metric is the creation of simplistic and unuseful metrics in order to increase the value of E(M).

Empirical validation of a metric should involve case studies [3]. The next section contains a case study about the use of E(M).

3 Case Study: Object-Relational Database Design Metrics

In this section, we demonstrate and analyze the use of metamodel-based methods that allow us to develop and analyze metrics. We introduced them in Section 2.

The concept "data model" has different meanings in different contexts. In this paper a *data model* is an abstract, self-contained, implementation-independent definition of elements of a set of sets {T, S, O, C} that together make up the abstract machine with which database users interact. In this case: T is a set of data types and types of data types; S is a set of data structures and types of data structures; O is a set of operators and types of operators; C is a set of constraints and types of constraints. This is a revised version of the definition that is presented by Date [15] and our previous definition [19]. Relational and object-relational data model are examples of this kind of data models. These data models are abstract languages [15] and we can use the methods that were presented in Section 2 in order to create and improve their corresponding metrics.

In this section, we investigate the object-relational (OR) data model. This model should combine the best properties of the relational data model and object-oriented programming languages. Currently there is no common OR data model yet. The work of Seshadri [28], 3rd- generation DBMS manifesto [29], The Third Manifesto (OR_{TTM}) [16], the work of Stonebraker et al. [30, 4], and SQL: 2003 (OR_{SQL}) [10] are all examples of different OR data model approaches. However, they have significant differences. For example, all the approaches from the set of previously mentioned approaches support the idea of an abstract data type system that allows designers to construct new types. However, there are different opinions about the exact nature of this system. For example, only 3rd- generation DBMS manifesto [29] and SQL: 2003 [10] propose the use of array type constructor. On the other hand, only Stonebraker et al. [30, 4] and SQL: 2003 [10] propose the use of reference type constructors. Eessaar [22] presents metamodels of OR_{SQL} and OR_{TTM} and their metamodel-based comparison.

More precisely, in this section we investigate OR_{SQL} and OR_{TTM} database design metrics. Piattini et al. [3] propose twelve metrics in order to evaluate OR_{SQL} database designs. We denote the set of these metrics as M_{ORSQL}. We are not aware of database design metrics, the specification of which uses OR_{TTM} terminology and which are created specifically for OR_{TTM}. Therefore, a task of this section is to investigate, how to create candidate OR_{TTM} database design metrics.

3.1 On Evaluating the Wording of Existing OR_{SQL} Database Design Metrics

A metamodel of a language (a data model in this case) allows us to find shortcomings in the specifications of metrics. A metamodel, is in this case a kind of aiding tool.

A metrics designer has to check, whether all the language elements that are referred in various specifications of a metric have exactly one corresponding element in a metamodel of the language or whether there are inconsistencies. For example, some specifications of the metrics that belong to M_{ORSQL} refer to "complex columns". The OR_{SQL} metamodel [22] does not have a class "complex column" and OR_{SQL} specification [10] does not refer to this concept. In addition, Piattini et al. [3] do not give exact definition of "complex column". Baroni et al. [9] write that a complex column has a structured type. However, a user-defined type is a structured type or a distinct type in

OR_{SQL}. In addition, OR_{SQL} allows us to use *constructed types* (multiset type, array type, row type) as declared types of columns. Both base and viewed tables can have columns, the declared type of which is not a predefined data type.

A metrics designer has also to check, whether all specifications of the same metric refer to exactly the same set of metamodel elements. For example, informally, a value of metric PCC(T) is “percentage of complex columns of a table T” [3]. Based on a metamodel of OR_{SQL} [22], we can see that a table is a base table, a transient table or a derived table (these classes form a specialization hierarchy). A viewed table (view) is a derived table. However, Piattini et al. [3] do not indicate, whether PCC(T) considers only base tables or also viewed tables. They are both schema objects. Baroni et al. [9] presents PCC(T) more formally by using OCL and shows that a PCC(T) value is calculated only based on *base tables*.

These examples illustrate that (1) informal specifications metrics should be more precise and (2) we need additional metrics that would take into account viewed tables, distinct types and constructed types.

3.2 On Designing OR_{TTM} Database Design Metrics Based on Existing Metrics

Table 1 presents mapping of some *classes* of the metamodels of OR_{SQL} and OR_{TTM}.

Table 1. Mapping of some classes of the metamodels of OR_{SQL} and OR_{TTM}

Class in the metamodel of OR_{SQL} [22]	*Class in the metamodel of OR_{TTM} [22]*
Base table	Real relvar, Relation
Typed base table	-
Structured type	User-defined scalar type
Base table column	Relvar attribute
Predefined data type	Built-in scalar type
Attribute	Attribute
SQL-invoked method	Read-only operator, Update operator
SQL-schema	-
Referential constraint	Referential constraint
Referencing column, Referenced column	-

Column “*Class in the metamodel of OR_{SQL}*” contains names of classes from the OR_{SQL} metamodel [22]. Name of a class exists in this column, if specification of at least one metric from the set M_{ORSQL} refers to a language element that has this corresponding class in a metamodel of OR_{SQL}. Column “*Class in the metamodel of OR_{TTM}*” contains names of the corresponding classes in the OR_{TTM} metamodel [22]. A pair of classes from the metamodels of OR_{SQL} and OR_{TTM} exists in the mapping, if these classes represent language elements that are semantically equivalent or significantly similar.

Next, we present examples of manual resolution of *construct deficit* problem that was described in step 3 of the algorithm in Section 2.1.1. The Third Manifesto argues explicitly against pointers at the *logical* database level and typed tables (including *typed base tables*) in the section “OO Prescriptions” [16]. Therefore, we cannot completely translate metric *Table size of a table T* that belongs to M_{ORSQL}.

Schema Size is a metric from M_{ORSQL}. A database is a named container of database relational variables (relvars) in OR_{TTM} [16]. OR_{SQL}, on the other hand, does not use the concept "Database". Instead it uses concepts "SQL-schema", "Catalog" and "Cluster", which are all collections of objects. An object is a cluster, a catalog, a SQL-schema, or a schema object. The OR_{SQL} metamodel class "SQL-schema" has no corresponding class in the OR_{TTM} metamodel. We think that in case of OR_{TTM} we could instead calculate *Database Size* (DS) instead of Schema Size. DS is sum of the size of every relvar in a database (a metric for estimating the size of a relvar must also be translated from OR_{SQL}).

Depth of relational tree of a table T DRT(T) is a metric from M_{ORSQL} that shows us "the longest path between a table and the remaining tables in the schema database" [9]. We have created classes *Referencing Column* and *Referenced Column* in the OR_{SQL} metamodel in order to model associations between *Base table column* and *Referential constraint.* Classes *Referencing Column* and *Referenced Column* are necessary in the OR_{SQL} metamodel because OR_{SQL} pays attention to the order of column names in a referential constraint specification and we need a place for the attribute *ordinal_position.* It is possible (but not necessary) to create corresponding classes in the OR_{TTM} metamodel. However, these classes would not have any attributes (including *ordinal_position*, because OR_{TTM} does not pay attention to the order of attribute names in a referential constraint specification). In addition, metrics in M_{ORSQL} do not take into account the ordinal position and therefore we conclude that it is possible to find corresponding metrics for DRT(T) in OR_{TTM} despite the construct deficit.

3.2.1 An Example

Next, we demonstrate how to create candidate OR_{TTM} database design metrics based on the OR_{SQL} metrics by using the algorithm that was introduced in Section 2.1.1. We investigate metrics NFK(T) and RD(T) that belong to the set M_{ORSQL} [3]. Baroni et al. [9] present specifications of NFK(T) and RD(T) in the following way:

- "NFK (Number of Foreign Keys): Number of foreign keys defined in a table.

  ```
  BaseTable:: NFK(): Integer= self.foreignKeyNumber()
  ```

- RD (Referential Degree): Number of foreign keys in a table divided by the number of attributes of the same table.

  ```
  BaseTable::RD(): Real= self.NFK() / (self.allColumns()
  -> size())"
  ```

These specifications consist of a natural language part and are also presented by using OCL, which arguably makes them more formal and understandable. Unfortunately, Baroni et al. [9] do not specify functions that are used in the OCL specification. We note that tables have *columns* and structured types have *attributes* according to the OR_{SQL} metamodel [22]. As you can see, analysis with the help of a metamodel may help us to improve the existing wording of metrics.

RD is an example of a metric that depends on another metric (NFK) and therefore we have to firstly translate NFK. We also note that metric *Referential Degree of a table T* (RD(T)) has different semantics in the studies of Piattini et al. [3] and Baroni et al. [9] and it causes confusion. Piattini et al. [3] defines RD(T) metric as "as the number of foreign keys in the table T". The corresponding metric in the work of Baroni et al. [9] is named *Number of Foreign Keys*.

Steps 1, 2: Relevant classes of the OR_{SQL} metamodel [22] are: *Base table*, *Base table column*, *Referential constrain* (see Table 1). We can find them by investigating nouns in the existing specifications of metrics.

Step 3: Table 1 presents classes of the OR_{TTM} metamodel that correspond to some classes of the OR_{SQL} metamodel. Firstly, some elements of the OR_{SQL} metamodel have more than one corresponding element in the OR_{TTM} metamodel. *Base table* has two corresponding classes in the OR_{TTM} metamodel – *Real relational variable (Real relvar)* and *Relational value (Relation).* OR_{TTM} clearly distinguishes the concepts "value" and "variable". A variable has at any moment one value, but it is possible to change this value. In OR_{SQL}, the concept "table" means "table value" as well as "table variable". The next definition is an example of that: "*A table is a collection of rows having one or more columns*" [10]. It is an example of *construct overload* [20] in OR_{SQL} because a construct in OR_{SQL} corresponds to several not-overlapping constructs in OR_{TTM}. Date and Darwen [16] write that referential constraints apply to *relvars*. Therefore, we decide that the corresponding class to *Base table* is in this case *Real relvar*.

Step 4: We identified the concept "real relvar" during the step 2. Date and Darwen [16] write: "*Referential constraints are usually thought of as applying to real relvars only. In the Manifesto, by contrast, we regard them as applying to virtual relvars as well.*" *Real relvar* and *Virtual relvar* are subclasses of *Relvar* in the OR_{TTM} metamodel. Therefore, in this case we can use class *Relvar* instead of class *Real relvar*.

Step 5: Now we can create specifications of two candidate metrics for OR_{TTM} by replacing OR_{SQL} concepts in the specifications with OR_{TTM} concepts. The specification consists of an informal natural language specification and a specification that is written in OCL. The level of precision of the specifications is analogous to [9].

- NRC (Number of Referential Constraints): Number of referential constraints where a relvar is the referencing relvar.

```
Relvar:: NRC(): Integer=
self.referentialConstraintNumber()
```

- RD (Referential Degree): Number of referential constraints where a relvar is the referencing relvar divided by the number of attributes of the same relvar.

```
Relvar:: RD(): Real= self.NRC() /(self.allAttributes()
-> size())
```

We created initialisms NRC and RD based on the names "Number of Referential Constraints" and "Referential Degree", respectively.

We also note that we can translate some metrics that are not intended to database design, in order to find candidate database design metrics. For example, Habela [31] presents a *metamodel* of an object-oriented database system. Date [15] explains that classes in object-oriented systems correspond to scalar data types in OR_{TTM} databases. An attribute in a class corresponds to a component of a possible representation of a scalar type. A method of a class corresponds to an operator that has been defined in an OR_{TTM} database. Therefore, it is possible to translate some OO design metrics [1] to candidate

OR_{TTM} database design metrics. For example, *Number of Attributes (NOA)* [1] becomes to *Number of components in a possible representation of a given type* and *Number of Methods in a Class (NOM)* becomes to *Number of read-only operators, the return value of which has a given type.*

3.3 On Evaluating the Extent of Sets of Database Design Metrics

We could create a set of metrics for OR_{TTM} by translating all the metrics in M_{ORSQL}. We denote this set as M_{ORTTM}. In this section, we evaluate the extent of the metrics in M_{ORSQL} and M_{ORTTM} based on the formula (1) (see Section 2.1.2).

The OR_{SQL} metamodel contains 110 classes [19]. Table 1 refers directly to 11 classes of the metamodel. These classes have additional 20 *different* subclasses. Therefore, the extent of M_{ORSQL} is: $E(M_{ORSQL})=((11+20)*100)/110=28.2\%$. As you can see, more than two thirds of OR_{SQL} constructs are not covered by these metrics.

UML [18] allows us to use packages in order to group model elements and manage complexity. According to *definition* (see Section 3), a data model has *four* components. Eessaar [19, 22] proposes to create four corresponding packages in order to manage the complexity of a metamodel of a data model that is presented by using UML: *Data types*, *Data structures*, *Data operators,* and *Data integrity*.

Ideally, each metamodel element should belong to exactly one of these packages. However, Eessaar [19, 22] has found 3 classes of the OR_{SQL} metamodel that cannot be classified to any of these packages. Table 2 presents the extent of M_{ORSQL} in terms of each of these packages. It shows us, how much metrics in M_{ORSQL} pay attention to the *different aspects* of OR_{SQL} data model.

Table 2. The extent of M_{ORSQL} in terms of the different data model components

Data model component	*Amt. of classes and their subclasses in the mapping (a)*	*Total amt. of classes in a package (b) [19]*	*$E(M_{ORSQL})$ (a*100)/b*
Data types	11	38	28.9%
Data structures	14	26	53.8%
Data integrity	3	16	18.8%
Data operators	3	27	11.1%

Table 2 shows us that metrics in M_{ORSQL} pay attention mostly to the *structural* part of OR_{SQL}. This is in line with the claims of the authors of metrics in M_{ORSQL}, who see these metrics as *structural* metrics. The biggest advantage of OR data models is possibility to create new types and operators [15]. However, existing OR_{SQL} metrics should pay more attention to types and operators. We can say this because Table 1 does not refer to classes of the OR_{SQL} metamodel that specify language elements like constructed data types, distinct types, and regular SQL-invoked functions. Table CHECK constraints, viewed tables, and user-defined functions / stored procedures with no overloading are examples of *mandatory* SQL features [10] that are not covered by the existing metrics according to Table 1. Type constructors, domains, triggers, and sequence generators are examples of *optional* SQL features [10] that are not covered by the existing metrics according to Table 1. On the other hand, a metric in

M_{ORSQL} takes into account typed tables and structured types that are *optional* SQL features [10]. As you can see, there is not one-to-one correspondence between the core of SQL and the existing metrics that belong to M_{ORSQL}.

Next, we calculate the extent of M_{ORTTM} based on the OR_{TTM} metamodel in order to evaluate M_{ORTTM}. We assume that M_{ORTTM} covers the following classes (and their subclasses): *Relvar*, *User-defined scalar type*, *Relvar attribute*, *Built-in scalar type*, *Attribute*, *Read-only operator*, *Update operator*, *Referential constraint*, *Database*. These 9 classes have 29 subclasses. The OR_{TTM} metamodel contains 95 classes [19].

Therefore, the extent of M_{ORTTM} is: $E(M_{ORTTM})=((9+29)*100)/95=40\%$. This extent is bigger compared to the extent of M_{ORSQL}.

A possible reason could be that OR_{SQL} violates the *orthogonality* principle more than OR_{TTM} [16, 19, 22]. Date and Darwen [16] write that the orthogonality principle means that a deliberate attempt has been made to *avoid arbitrary restrictions* in combinations of different language constructs. For example, OR_{SQL} permits foreign key constraints *only* in base tables but OR_{TTM} in *all* relvars (including virtual). Therefore, M_{ORTTM} metrics are calculated based on bigger amount of *different* types of database objects compared to M_{ORSQL}.

It could be argued that some constructs of a data model cannot be used or misused in a way that affects the overall quality of database design and therefore corresponding metrics are not needed. However, why when to develop standards and systems that specify and allow us to create entities that are unnecessary and not very useful? Most of the current database design metrics that are proposed by researchers are simple counts that are not very precisely described. It rather seems that small E(M) values point to the need to continue development of OR_{SQL} and OR_{TTM} metrics.

4 Conclusions

In the paper, we investigated how to use metamodels of languages in order to evaluate and improve specifications of existing software metrics and to design candidate metrics.

We proposed a metamodel-based derivation method of candidate metrics and new candidate metric E(M) that allows us to evaluate completeness of sets of metrics. The metamodel-based derivation method allows us to reuse existing metrics by translating them so that they are possibly usable in a new context. Actual usefulness of these new candidate metrics must be found out based on careful evaluation. The evaluation procedure was not in the scope of the paper. The proposed method is not intended to replace existing methods of metrics development but should complement them. Currently it is too early to say whether its use will become common practice.

We demonstrated the usefulness of the proposed method based on database design metrics. The paper considered two object-relational data model approaches – SQL:2003 (OR_{SQL}) and The Third Manifesto (OR_{TTM}) as the examples. The analysis of some existing OR_{SQL} design metrics revealed problems in the wording of them. We demonstrated how to translate some existing OR_{SQL} metrics in order to create candidate metrics for evaluating OR_{TTM} database design. In the proposed case study the languages (data models) are relatively similar to each other. There would be more discrepancies between metamodels if the languages are more different. It will allow us to translate fewer metrics and will reduce possibility of automatic metric translation.

We also found that the completeness of an existing set of OR_{SQL} metrics is small ($E(M) \approx 28\%$). These metrics together cover only small part of all possible OR_{SQL} constructs. Closer investigation showed that these metrics do not pay enough attention to different kinds of data types and routines and therefore design of new metrics must continue.

Future work will include development of more OR_{TTM} database design metrics and further evaluation of E(M).

References

1. Choinzon, M., Ueda, Y.: Design Defects in Object Oriented Designs Using Design Metrics. In: 7th Joint Conference on Knowledge-Based Software Engineering, pp. 61–72. IOS Press, Amsterdam (2006)
2. Piattini, M., Calero, C., Genero, M.: Table Oriented Metrics for Relational Databases. Software Quality Journal 9(2) (June 2001)
3. Piattini, M., Calero, C., Sahraoui, H., Lounis, H.: Object-Relational Database Metrics. L'Object (March 2001)
4. Muller, R.J.: Database Design for Smarties. Morgan Kaufmann, San Francisco (1999)
5. Seidewitz, E.: What models mean. IEEE Software 20(5), 26–31 (2003)
6. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons, USA (2004)
7. McQuillan, J.A., Power, J.F.: On the application of software metrics to UML models. In: Model Size Metrics Workshop of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (2006)
8. Reißing, R.: Towards a Model for Object-Oriented Design Measurement. In: 5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, pp. 71–84 (2001)
9. Baroni, A.L., Calero, C., Piattini, M., Abreu, F.B.: A Formal Definition for Object-Relational Database Metrics. In: 7th International Conference on Enterprise Information Systems (2005)
10. Melton, J.: ISO/IEC 9075-2:2003 (E) Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation) (August 2003)
11. Gruber, T.R.: Towards principles for the design of ontologies used for knowledge sharing. International Journal of Human Computer Studies 43(5/6), 907–928 (1995)
12. Baroni, A.L., Abreu, F.B., Calero, C.: Finding Where to Apply Object-Relational Database Schema Refactorings: An Ontology-Guided Approach. In: X Jornadas Sobre Inginiería del Software y Bases de Datos (2005)
13. Schneidewind, N.F.: Methodology for Validating Software Metrics. IEEE Trans. Softw. Eng. 18(5), 410–422 (1992)
14. Kaner, C., Bond, P.: Software Engineering Metrics: What Do They Measure and How Do We Know? In: 10th International Software Metrics Symposium (2004)
15. Date, C.J.: An Introduction to Database Systems, 8th edn. Pearson/Addison-Wesley, Boston (2003)
16. Date, C.J., Darwen, H.: Types and the Relational Model. The Third Manifesto, 3rd edn. Addison-Wesley, Reading (2006)

17. IEEE Std. 1061-1998, Standard for a Software Quality Metrics Methodology. IEEE Standards Dept. (1998)
18. OMG UML 2.0 Superstructure Specification, formal/05-07-04
19. Eessaar, E.: On Specification and Evaluation of Object-Relational Data Models. WSEAS Transactions on Computer Research 2(2), 163–170 (2007)
20. Opdahl, A.L., Henderson-Sellers, B.: Ontological Evaluation of the UML Using the Bunge–Wand–Weber Model. Software and Systems Modeling 1(1), 43–67 (2002)
21. Davies, I., Green, P., Milton, S., Rosemann, M.: Using Meta Models for the Comparison of Ontologies. In: Evaluation of Modeling Methods in Systems Analysis and Design Workshop (2003)
22. Eessaar, E.: Relational and Object-Relational Database Management Systems as Platforms for Managing Software Engineering Artifacts. Ph.D. Thesis. Tallinn University of Technology, Estonia (2006)
23. Levendovszky, T., Karsai, G., Maroti, M., Ledeczi, A., Charaf, H.: Model Reuse with Metamodel-Based Transformations. In: Gacek, C. (ed.) ICSR 2002. LNCS, vol. 2319, pp. 166–178. Springer, Heidelberg (2002)
24. Merriam-Webster, Inc. Merriam-webster's online dictionary, `http://www.m-w.com/`
25. Kim, H., Boldyreff, C.: Developing software metrics applicable to UML Models. In: 6th International Workshop on Quantitative Approaches in Object–Oriented Software Engineering, pp. 147–153 (2002)
26. Dori, D., Reinhartz-Berger, I.: An OPM-Based Metamodel of System Development Process. In: Song, I.-Y., Liddle, S.W., Ling, T.-W., Scheuermann, P. (eds.) ER 2003. LNCS, vol. 2813, pp. 105–117. Springer, Heidelberg (2003)
27. DMTF Common Information Model (CIM) Standards. CIM Schema Ver. 2.15. Database specification
28. Seshadri, P.: Enhanced abstract data types in object-relational databases. The VLDB Journal 7(3), 130–140 (1998)
29. Stonebraker, M., Rowe, L.A., Lindsay, B., Gray, J., Carey, M., Brodie, M., Bernstein, P., Beech, D.: Third-generation database system manifesto. Computer Standards & Interfaces 13(1–3), 41–54 (1991)
30. Stonebraker, M., Brown, P., Moore, D.: Object-Relational DBMSs: Tracking the Next Great Wave, 2nd edn. Morgan Kaufmann, San Francisco (1999)
31. Habela, P.: Metamodel for Object-Oriented Database Management Systems. PhD Thesis. Polish Academy of Sciences, Warsaw, Poland (2002)

Automatic Transactions Identification in Use Cases[*]

Mirosław Ochodek and Jerzy Nawrocki

Poznań University of Technology, Institute of Computing Science,
ul. Piotrowo 3A, 60-965 Poznań, Poland
{Miroslaw.Ochodek,Jerzy.Nawrocki}@cs.put.poznan.pl

Abstract. Since the early 90's of the previous century, use cases have became informal industry standard for presenting functional requirements. The rapid popularity growth stimulated many different approaches for their presentation and writing styles. Unfortunately, this variability makes automatic processing of use cases very difficult. This problem might be mitigated by the use of transaction concept, which is defined as an atomic part of the use case scenario. In this paper we present approach to the automatic transaction discovery in the textual use cases, through the NLP analysis. The proposed solution was implemented as a prototype tool UCTD and preliminarily verified in a case study.

Keywords: Use cases, Use-cases transactions, Use Case Points, Requirements engineering, Effort estimation, Natural language processing.

1 Introduction

In the field of requirements specification there are two extremes. One is a formal approach based on sound mathematical background. People interested in this approach can use such notations as VDM, Z [17], Statecharts [16], Petri Nets [24] and many others. The problem is that those notations lead to quite long specifications (sometimes their size is comparable to code size) and proving correctness of commercial-like programs can be quite time consuming (an interesting case study has been described by Wolfgang Reif [23,13]). But what is more important, mathematically sound specifications usually are unreadable for a typical end-user.

Another extreme is completely unformal approach based on the oral communication. The best example of that approach are user stories, which are one of the main practices of Extreme Programming [6]. Here the main weakness is dependence on human memory. In case of difficult requirements with many competing approaches, each one having indirect impact, relying on human memory can be dangerous.

[*] This research has been financially supported by the Ministry of Science and Higher Education grant N516 001 31/0269.

Somewhere in between there are use cases. They have been introduced by Ivar Jacobson [19] and further developed by Cockburn [9] and others [1]. A typical use case describes a user-valued transaction in a sequence of steps, and each step is expressed in a natural language (if needed, one can extend a given step with an alternative behaviour). That makes use cases readable for end-users.

Use cases can be used not only for the system's behaviour specification, but also for the effort estimation. Kerner proposed so-called Use Case Points method (UCP) [20] resembling Function Points [2]. In the UCP size of each use case depends on the number of transactions embedded in it. Unfortunately, the notion of use-case transaction is vague. Due to this, development of sizing tools for use-cases is almost impossible.

In the paper the term of use-case transaction is formalised and a method of automatic identification of transactions in use cases is presented. Transaction identification can be used as a bases for effort estimation and for automatic use-case review.

The paper is organised as follows. In the next section, the rationale for transaction counting is presented as well as existing work concerning their automatic discovery. In Section 4, a formal definition of the use case transaction is introduced, together with an approach for their automatic extraction from the textual use case. The case study for preliminary verification of the model and method is presented in Section 5. Some remarks and lessons learned concerning use case writing style for transaction identification, are presented in Section 6.

2 Use-Case Transactions Counting Problem

The lack of formal standards for the use cases presentation causes many problems with the developing generic and format independent tools and methods. Thus, it is very important to find a common denominator, which might be derived from the use cases despite the differences in the presentation format.

The use case transaction is a decent example of such a concept. It is strictly connected with the use case logic, while dependancy on its notation is limited. It has been introduced by Ivar Jacobson [18]. According to the Jacobson each step should be a transaction, with the four types of phrases included:

- the main actor sends request and data to the system,
- the system validates given data,
- the system performs internal state change operations,
- the system responds to the actor with the operation result.

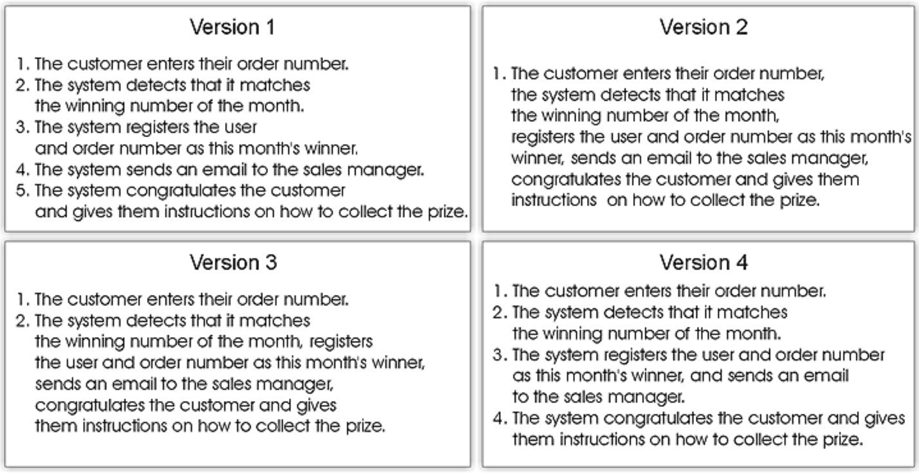
Transactions are mainly used as a complexity measurement within the Use Case Points (UCP) method [20]. The UCP is an accepted software size metric [4], which might be also used for the effort estimation. Each use case is classified into one complexity class depending on the number of transactions. Use case with three or less transactions is considered as a simple, from four to seven as an average and finally if it has more then seven transactions, it is classified as a complex one. The next step of the UCP method, is a computation of the UUCW (*Unadjusted Use*

Case Weight), which is the sum of simple use cases set cardinality multiplied by 5, cardinality of average set multiplied by 10 and cardinality of complex set by 15.

A transition between size and effort is done by multiplying the UCP method output by the Productivity Factor (PF), which defines how many man hours are required to cover one Use Case Point. The easiest way to obtain the PF within the certain organisation is to compute it from the historical data or use values presented in the literature (according to authors writing about the UCP, PF varies from 20 to 36 h/UCP [20,27]). The Productivity Factor makes UCP method easy to calibrate, as long as historical data comes from the similar projects and rules used for the counting transactions number and assessing other method components does not differ significantly between the considered projects (*experts should be consistent in their approach to the transactions counting*).

According to some authors [5,8,25] use case transactions might be counted as the number of steps in the use case scenarios (main and alternative). This approach is acceptable as long as the use cases are written according to the Jacobson's one step - one transaction rule. In other case relying on counting steps as transactions, might trigger overestimation problems.

Alistair Cockburn in [9] presented relevant example that the same scenario might be easily written with the different number of steps. Each of the four presented use cases consists of the one transaction but the number of steps varies from the one to five (Cockburn's example, is presented in the figure 1). If we follow the UCP classification rules for the use cases complexity classification, two presented examples would be classified as a simple and the other two as an average.


Version 1

1. The customer enters their order number.
2. The system detects that it matches the winning number of the month.
3. The system registers the user and order number as this month's winner.
4. The system sends an email to the sales manager.
5. The system congratulates the customer and gives them instructions on how to collect the prize.

Version 2

1. The customer enters their order number, the system detects that it matches the winning number of the month, registers the user and order number as this month's winner, sends an email to the sales manager, congratulates the customer and gives them instructions on how to collect the prize.

Version 3

1. The customer enters their order number.
2. The system detects that it matches the winning number of the month, registers the user and order number as this month's winner, sends an email to the sales manager, congratulates the customer and gives them instructions on how to collect the prize.

Version 4

1. The customer enters their order number.
2. The system detects that it matches the winning number of the month.
3. The system registers the user and order number as this month's winner, and sends an email to the sales manager.
4. The system congratulates the customer and gives them instructions on how to collect the prize.

Fig. 1. The example [9] of four versions of the one-transaction use case with scenario varying from 1 to 5 steps. Relying on scenarios length (measured in steps) while counting transactions number may lead to corrupted results.

In the case study presented in the section 5, the UUCW counted for thirty use cases based on the number of steps is 2 times higher then mean UUCW value counted for all experts based on their transactions number assessment.

The important problem concerning transactions identification appears as a difference in formats and styles of writing use cases, which involves other views on the actors actions. The list of some use case structure problems, in the transaction discovery context, is presented in the section 6.

3 Definition of Use-Case Transaction

We have already stated that measuring use case complexity based on the transactions number, seems to be better idea then counting steps in scenarios. However, the notion of transaction should be formally defined in order to provide clear rules for their automatic identification.

The approach to transaction identification for use cases written in Japanese was described by Kusumoto et al. [22]. Authors proposed a system (U-EST), which processed requirements stored in the XMI format and estimating effort based on the UCP method. It is stated that morphological analysis for all statement was conducted in order to find transactions. Each transaction was concerned as a set of subject and predicate that relates to actor's operation and system response. Unfortunately, it is not clear for us, whether authors perceived the pair of actor and system phrases as a transaction or each of them as separate one. The U-EST system was evaluated using data provided by experienced engineers from the Hitachi company. In the case study classification given by the system was compared only with one specialist (authors did not provide exact number of identified transactions, but the final use case classification with the UCP method). The assumption was made that use case grammar is "absurdly simple". Based on our experience, it seems that use cases language, especially in the commerce requirements specifications, is not always as simple as it is suggested in the literature.

Another redefinition of the use case transaction was presented by Sergey Diev [11]. The transaction idea remains similar to the Jacobson's, but it is more general. Author defines transaction as the smallest unit of activity that is meaningful from the actor's point of view. What is more important, use case transaction should be self-contained and leaves the business of the application in a consistent state.

3.1 Proposed Transaction Model

We would like to propose a transaction model which is based on the Ivar Jacobson's transaction definition [18].

We enumerates four types of actions, which are relevant from the use case transaction point of view:

– actor's request action (U),
– system data validation action (SV),

– system expletive actions (e.g. system internal state change action), which are neither validation nor confirmation actions (SE),
– system response action (SR).

We perceive transaction as an atomic sequence of activities (actions) performed by actor and system, which is performed entirely or not at all. Each action belongs to one of the four sets U, SR, SV, SE.

Definition 1. The certain transaction is a shortest sequence of actor's and system actions, which starts from the actor's request (U) and finishes with the system response (SR). The system validation (SV) and system expletive (SE) actions may optionally occur within the starting and ending action. The pattern for the certain transaction written as a regular expression:

$$T_{certain} = \text{U+ [SV SE]}* \text{ SR+}$$

Where

– *U - actor's request action,*
– *SV - system data validation action,*
– *SE - system expletive action,*
– *SR - system response action.*

Definition 2. We mark transaction as an uncertain (but still as transaction) if it starts from the actor's action (U), but it lacks corresponding system response phrase (SR). In this case system expletive action (SE) might be treated as a transaction closure. It is possible only if the next action is the actor's action (U) or the end of scenario. Thus, uncertain transaction might be defined as:

$$T_{uncertain} = \text{(U+ SV}* \text{ SE+) (?![SV SR])}$$

Where

– *U - actor's request action,*
– *SV - system data validation action,*
– *SE - system expletive action,*
– *SR - system response action.*

Alternative scenarios (extensions) are also included in the transaction discovery process. The role of extensions scenario depends on the type of corresponding action in the main scenario. Extensions which are regarding system actions, are part of the transaction started in the main scenario. If the extension is triggered by the U-type action, the user decision point is reached. This means that main scenario is being forked into two or more alternative paths. To remark decision point, expression in assertion form might be added before action, in order to express actor's intention (*e.g. User wants to add article*). If the corresponding extension also starts with the assertion (*e.g. User wants to add news*), the alternative execution paths are clearly defined. If the actions in such alternative scenario matches patterns presented in definitions 1 or 2, a new transaction is marked.

Presented certain transaction definition should satisfy both conditions defined by Sergey Diev [11]. The actor makes a request (U), which implies that action is meaningful from his point of view. System responds to the actor's request (SR), which means that the business of the application is left in a consistent state.

Based on the transaction definition, we would like to propose a graphical representation of transaction, which is the Transaction Tree. The root element represents a transaction itself. The next two layers states for the types of actions, which are named here as parts. Each part groups corresponding types of phrases. The action is represented in the tree as:

– <Actor> is the actor's name with distinction to actor(real) and system,
– <PredicateSyn> represents set of action's predicate synonyms (for example show, present etc. would be grouped together),
– <Object> is the object phrase.

The Transaction Tree might also include prolog (*eg. Use case starts when actor enters the page; System displays welcome page etc.*) or epilog phrases (*Use case ends etc*).

4 Transaction Identification with NLP Tools

We use NLP (*Natural Language Processing*) techniques to extract the Transaction Tree and mark transactions occurrence in the use case text. The most important stage of this process is detection of actor's and system actions.

Transactions Discovery Process. The transactions identification rules was developed based on thirty one different use cases coming from the literature [1,9]

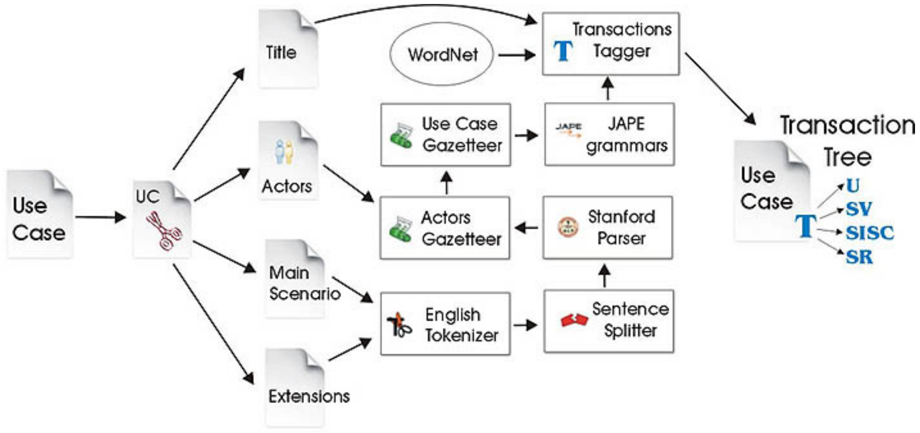


Fig. 2. The UCTD application processing pipeline. Textual use case is split into title, main scenario and extensions documents. Actors names are extracted into the GATE Gazetteer form. The main scenario and extensions part are processed in order to find actions. Finally, the Transaction Tagger performs construction of the Transaction Tree.

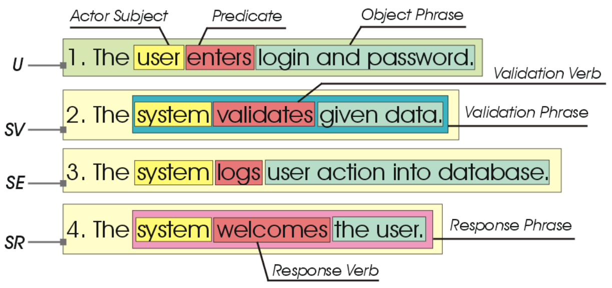
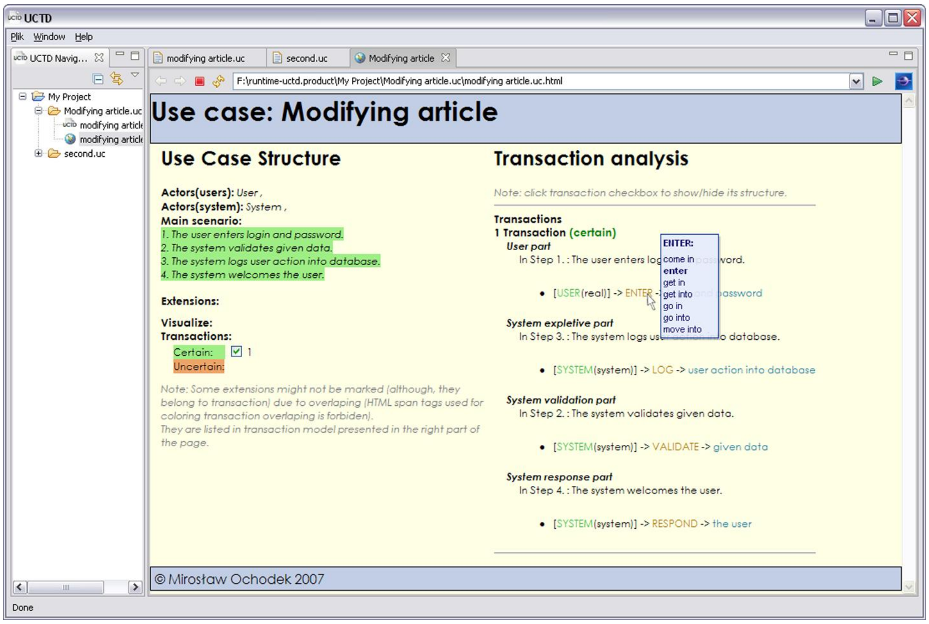

Actor Subject
Predicate
Object Phrase
U
1. The user enters login and password.
Validation Verb
SV
2. The system validates given data.
Validation Phrase
SE
3. The system logs user action into database.
SR
4. The system welcomes the user.
Response Phrase
Response Verb

Fig. 3. Example of the use case text with annotations. The most important annotations, like actor subject, predicate, object phrase, validation verb and response verb are marked. According to the defined rules, validations and response phrases are extracted from the text. Finally, each phrase is classified into the part U, SV, SE or SR.



Use case: Modifying article

Use Case Structure

Actors(users): *User ,*
Actors(system): *System ,*
Main scenario:
1. The user enters login and password.
2. The system validates given data.
3. The system logs user action into database.
4. The system welcomes the user.

Extensions:

Visualize:
Transactions:
Certain: ✓ 1
Uncertain:

Note: Some extensions might not be marked (although, they belong to transaction) due to overlaping (HTML span tags used for coloring transaction overlaping is forbiden).
They are listed in transaction model presented in the right part of the page.

Transaction analysis

Note: click transaction checkbox to show/hide its structure.

Transactions
1 Transaction (certain)
User part
In Step 1. : The user enters log[illegible]word.

- [USER(real)] -> ENTER -[illegible]assword

ENTER:
come in
enter
get in
get into
go in
go into
move into

System expletive part
In Step 3. : The system logs us[illegible]o database.

- [SYSTEM(system)] -> LOG -> user action into database

System validation part
In Step 2. : The system validates given data.

- [SYSTEM(system)] -> VALIDATE -> given data

System response part
In Step 4. : The system welcomes the user.

- [SYSTEM(system)] -> RESPOND -> the user

© Mirosław Ochodek 2007

Fig. 4. The UCTD application screen (HTML report). On the left side use case is presented (chosen transaction is highlighted). Transaction Tree is presented on the right side.

and various Internet sources. The variability of styles helped us to develop solution, which is flexible enough to handle different approaches for writing use cases.

The proposed idea was implemented as a prototype tool UCTD. It is capable of analysing use cases written in English. The application was developed based on the GATE Framework [10]. The grammatical structure analysis was performed with the use of Stanford Parser [21]. The use case processing chain is presented in the figure 2 and consists of the following steps:

1. Preprocessing phase, which involves use case structure verification (see 4), actors extraction into the GATE Gazetteer form (lists of words which are looked up in the text). Use case is split into title, main scenario and extensions documents. Each part is further tokenized and sentences are annotated.
2. POS tagger is used to annotate parts of speech.
3. English Grammar parser is used to annotate parts of sentence (subjects, predicates, objects etc.).
4. Important words and phrases are being looked up in the scenarios text (actors, validations, response phrases etc.).
5. Actor and system actions are marked with the use of JAPE grammars. Additional postprocessing is done to increase accuracy. The example of use case text with annotations is presented in the figure 3.
6. Transaction Tree is built based on the actor's and system actions sequences. Actions predicates are converted into the sets of predicates synonyms using the WordNet dictionary [15] or with the use of explicitly defined lists of synonyms. The example of the Transaction Tree, extracted from the annotated text, is presented in the figure 4 (the UCTD application screen).

5 Transaction Identification – A Case Study

In order to verify proposed approach for transaction identification we conducted case study, based on thirty use cases coming from the industry project, developed for the e-government sector.

The specification was tagged individually by six experts and UCTD tool. We wished to measure system accuracy as well as investigate how the experts deal with the transaction detection problem in the specification coming from the software industry (*The case study was preceded by the warmup phase with the assessment of four use cases taken from the literature examples*). The detailed results of the case study are presented in the table 1.

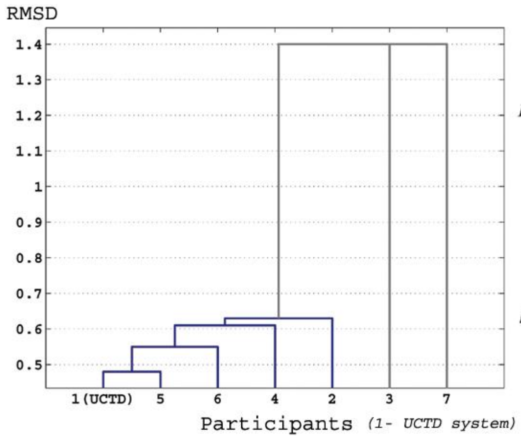
5.1 Comparison of System and Experts Results

Although, use cases structure were correct, we perceived few scenarios as ambiguous from the transactions identification point of view. Thus, we assumed that each expert presents his own approach to transaction counting and neither of their assessments could be accepted as the "standard". In order to measure discrepancy between system and experts opinions, error analysis was performed (concerning differences in number of transactions counted by the participants for each use case). The root mean square deviation (RMSD) matrix, dendrogram (RMSD) and Spearman's rank correlation coefficients matrix are presented in the figure 5. According to the RMSD analysis, experts clearly differs in their decisions. However, three clusters can be distinguished. The group with the greatest cardinality, couples four experts (2, 4, 5, 6) and the UCTD system. The rest of experts (3, 7) differs from themselves as much as from the rest of participants. In this case discrepancy between the UCTD system was not greater then difference between the experts. The average time required for transaction counting

Table 1. Case study results summary. All use cases are presented with the number of steps (in the main scenario and extensions) and transactions counted by the system and experts.

Use Case ID	No. Steps (All)	UCTD(1)	2	3	4	5	6	7
WF1101	14	3	3	3	4	2	3	4
WF1102	14	2	5	4	3	2	3	4
WF1104	5	1	1	1	1	1	1	3
WF1105	7	2	2	2	2	2	1	3
WF1106	6	1	1	3	1	2	1	1
WF1107	4	1	1	1	1	1	1	2
WF1108	4	1	1	1	1	1	1	2
WF1109	6	1	1	2	1	1	1	2
WF1110	14	3	3	7	4	2	2	5
WF1201	9	1	1	3	1	1	1	3
WF1202	6	2	2	3	1	2	1	3
WF1203	3	1	1	2	1	1	1	1
WF1204	6	1	1	2	1	1	1	2
WF2501	9	2	2	3	1	2	2	4
WF2502	5	1	1	2	2	1	1	2
WF2503	17	3	2	3	2	2	2	3
WF2505	7	2	1	2	1	1	2	3
WF2506	11	2	2	3	3	2	1	5
WF3301	3	1	1	2	1	1	1	1
WF3302	5	1	1	3	1	1	1	3
WF3303	2	1	1	1	1	1	1	1
WF4201	4	1	1	1	1	1	1	2
WF4202	3	1	1	1	1	1	1	1
WF4203	9	2	3	5	2	2	1	1
WF4301	18	2	2	5	2	2	2	8
WF4311	7	2	1	1	1	1	1	4
WF4312	6	2	2	3	1	2	1	3
WF4402	5	1	1	3	1	2	1	3
WF4403	4	1	1	2	1	1	1	2
WF4404	3	1	1	2	1	1	1	2
Time	-	41s	39min	35min	17min	28min	34min	46min
$\sum Transactions$	-	46	47	76	45	43	39	83
UCP	320	150	155	170	160	150	150	190

per expert was 33 minutes, while the system processed whole corpus in 41 seconds. The correlation analysis was conducted to identify whether differences in experts opinions are of the systematic nature. Generally, correlation coefficient is higher for experts pairs with the lower RMSD. Lower correlation is observed for experts who differed more in their assessments (higher RMSD). Although, correlation is significantly different from zero in most cases, we did not observed variability caused by experts systematic over or underestimation.



a)

$$RMSD = \begin{vmatrix} 0.000 & 0.658 & 1.483 & 0.606 & 0.483 & 0.548 & 1.742 \\ & 0.000 & 1.402 & 0.632 & 0.683 & 0.683 & 1.826 \\ & & 0.000 & 1.472 & 1.538 & 1.722 & 1.402 \\ & & & 0.000 & 0.730 & 0.683 & 1.807 \\ & & & & 0.000 & 0.577 & 1.880 \\ & & & & & 0.000 & 1.949 \\ & & & & & & 0.000 \end{vmatrix}$$

b)

$$\rho = \begin{vmatrix} 1.000 & 0.899 & 0.642 & 0.770 & 0.798 & 0.796 & 0.731 \\ & 1.000 & 0.771 & 0.845 & 0.889 & 0.755 & 0.636 \\ & & 1.000 & 0.654 & 0.813 & 0.626 & 0.523 \\ & & & 1.000 & 0.729 & 0.741 & 0.586 \\ & & & & 1.000 & 0.662 & 0.586 \\ & & & & & 1.000 & 0.688 \\ & & & & & & 1.000 \end{vmatrix}$$

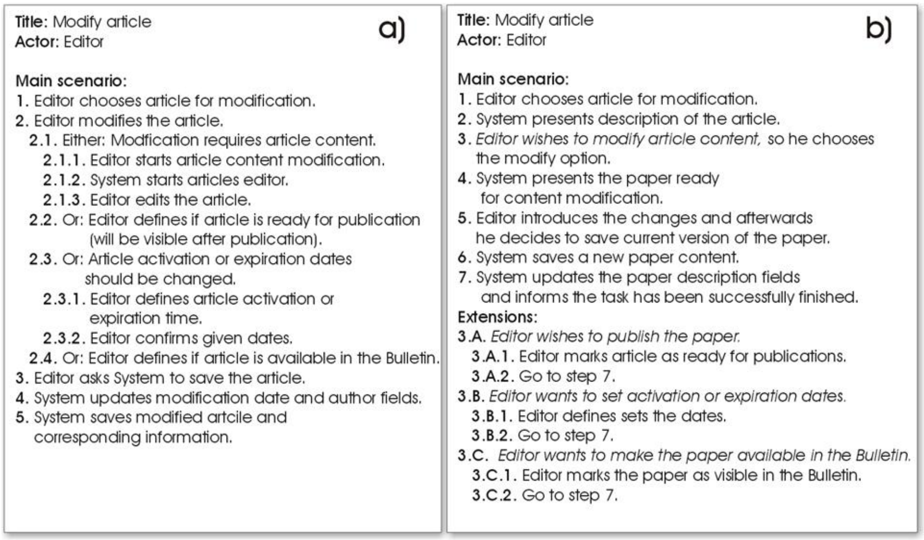
Fig. 5. Experts transaction counting errors comparison. a) root mean square deviation (RMSD) matrix and corresponding dendrogram (nearest distance) are presented for the system and experts. RMSD coupling reveals that experts differs from themselves (and system) in transactions counting. However, the group of four coherent experts and system, could be observed: the UCTD tool (1) and experts (2, 4, 5, 6). b) Spearman's rank correlation coefficients matrix.

If we consider case study set in the UCP method context, the range of use cases complexity (UUCW), based on transactions marked by the experts was from 150 to 190, with the mean value of 158. Majority of use cases were rather simple (3 from 7 experts marked all use cases as simple, which gave UUCW value 150). The variability could be even grater if analysed use cases were more complex (*if more use cases were closer to simple, average and complex sets membership border values*). For comparison the UUCW counted as a number of steps would be 320.

6 Transaction-Driven Use-Case Writing Style

We have noticed that reason for decisions variance might come from the use cases structural problems (e.g. responds steps omissions, sequences of actor's requests etc.). Furthermore, we observed that literature examples used for the warmup phase caused less difficulties for the experts. We have analysed use cases, which involved greatest experts differences to point the most important problems.

In the figure 6, two versions of the same use case are presented. The first one is coming from the requirements specification assessed in the case study. What is interesting transactions number estimated by experts, varied from 2 to 5 (assessments were 2, 3, 4, 5 transactions). There are at least two main problems in the structure of this use case. The first one is omitting system response phrases. This practice was very often observed. This makes use case shorter but as a side effect scenario becomes more ambiguous. The sequence of actor requests might be grouped within the one transaction. This is acceptable when more general



a)

Title: Modify article
Actor: Editor

Main scenario:
1. Editor chooses article for modification.
2. Editor modifies the article.
  2.1. Either: Modfication requires article content.
    2.1.1. Editor starts article content modification.
    2.1.2. System starts articles editor.
    2.1.3. Editor edits the article.
  2.2. Or: Editor defines if article is ready for publication (will be visible after publication).
  2.3. Or: Article activation or expiration dates should be changed.
    2.3.1. Editor defines article activation or expiration time.
    2.3.2. Editor confirms given dates.
  2.4. Or: Editor defines if article is available in the Bulletin.
3. Editor asks System to save the article.
4. System updates modification date and author fields.
5. System saves modified artcile and corresponding information.

b)

Title: Modify article
Actor: Editor

Main scenario:
1. Editor chooses article for modification.
2. System presents description of the article.
3. *Editor wishes to modify article content,* so he chooses the modify option.
4. System presents the paper ready for content modification.
5. Editor introduces the changes and afterwards he decides to save current version of the paper.
6. System saves a new paper content.
7. System updates the paper description fields and informs the task has been successfully finished.
Extensions:
3.A. *Editor wishes to publish the paper.*
  3.A.1. Editor marks article as ready for publications.
  3.A.2. Go to step 7.
3.B. *Editor wants to set activation or expiration dates.*
  3.B.1. Editor defines sets the dates.
  3.B.2. Go to step 7.
3.C. *Editor wants to make the paper available in the Bulletin.*
  3.C.1. Editor marks the paper as visible in the Bulletin.
  3.C.2. Go to step 7.

Fig. 6. Use case from the case study corpus. a) original use case, with omitted actions and condition statement, b) modified use case, conforming transaction driven writing style.

action (*e.g. User fills the login form*) is split into many subactions (*e.g. User enters login. User enters password.*). However, such sequence of actor's requests might also emerge as a result of omitting system response. We have noticed that, in some cases, experts where treating single actor's action as transaction, based on their experience. Another important problem is conditional statement in the main scenario. According to the guidelines concerning use cases, main scenario should describe the most common sequence of actions. Introducing many alternative paths in the main scenario makes the use case difficult to read, because the main goal is being blurred.

Second version of the use case, presented in the figure 6, has been rewritten by authors to present more transactional approach to writing use case scenarios. We leave assessment of the two presented versions to the reader.

Based on our experienced, it seems that use cases which cannot be easily split into transactions are potentially ambiguous. Problems in transactions identifications are not an evidence that specification is incorrect, however they should be perceived rather as a "bad smell" (an indicator of potential problem). Providing use cases with a structural problems may lead to a similar variance in experts opinions as in case of presented case study.

We have gathered few guidelines for writing use cases which are suitable for the transactions identification:

– use case should be defined at system level, scenarios should contain actor's and system phrases (*e.g. User does ..., System does ... etc.*). The transaction concept is hardly applicable to the business level use cases.
– actors names (real and system) should be defined explicitly,

– use cases should be provided in a structured textual form (or in other format which might be transformed into text). This means that use case contains sequences of steps, which forms scenarios. Each step begins with a prefix. In order to process extensions, their prefixes should correspond to the step prefix.
– system responses should not be omitted. This helps in finding transaction beginning and closure meaningful from the actor point of view,
– a main scenario should describe the most common sequence of actions. It should not contain conditional statements. If many alternative paths are possible, it seems to be better idea to use extensions triggered on the actor action with assertion statement, which presents actor's intention.

In order to decrease ambiguity of use cases scenarios another action could be taken as incorporating guidelines regarding writing properly structured use cases [1,8,9,26], conducting requirements specification inspections and reviews [3,12]. Many other, potential problems concerning use case structure might be found automatically through the NLP analysis [7,14].

7 Conclusions

In the paper we have presented an approach to automatic transaction counting in use cases. We have proposed a formal model of transaction, which can be used for processing requirements specification. Transaction Tree derived from a textual use case, provides detailed information about its structure (which might be further used not only for effort estimation). Even though the presented model of transaction is formal, it is still easy to applicate. It can be automatically extracted from use-case text in reasonably short time.

The proposed solution for automatic transaction discovery has been implemented as a prototype tool UCTD. Its accuracy was preliminarily verified in the described case study. According to the results, the system did not differ more from the experts than the experts between themselves. Moreover, the tool provided most uniform judgements when compared with the human experts. In addition, conducted case study revealed, that transaction identification is difficult even for people. When experts vary in their opinions, it can significantly impact on the decisions, which are made based on the number of transactions (e.g. size and effort estimation).

The reason for variability in expert opinions might be caused by quality of specification or use case writing style. We believe that knowing number of certain and uncertain transactions might help in detecting structural defects in use cases.

In the near future we would like to elaborate a method and a tool, which would be capable of handling Polish, which is mother tongue for the authors.

Acknowledgments. We would like to thank Alicja Ciemniewska, Piotr Godek, Jakub Jurkiewicz, Wojciech Kopras, Marek Kubiak, Bartosz Michalik and ï£¡ukasz Olek for valuable discussions and involvement in the proposed solution assessment.

This research has been financially supported by the Ministry of Science and Higher Education under grant N516 001 31/0269.

References

1. Adolph, S., Bramble, P., Cockburn, A., Pols, A.: Patterns for Effective Use Cases. Addison-Wesley, Reading (2002)
2. Albrecht, A.J., Gaffney Jr., J.E.: Software function, source lines of code and envelopment effort prediction: a software science validation. Mcgraw-Hill International Series In Software Engineering, pp. 137–154 (1993)
3. Anda, B., Sjøberg, D.I.K.: Towards an inspection technique for use case models. In: Proceedings of the 14th international conference on Software engineering and knowledge engineering, pp. 127–134 (2002)
4. Arnold, M., Pedross, P.: Software Size Measurement and Productivity Rating in a Large-Scale Software Development Department. In: Proc. of the 20th ICSE, pp. 490–493 (1998)
5. Banerjee, G., Production, A.: Use Case Estimation Framework.
6. Beck, K., Fowler, M.: Planning Extreme Programming. Addison-Wesley, Reading (2000)
7. Ciemniewska, A., Jurkiewicz, J., Nawrocki, J., Olek, Ł.: Supporting use-case reviews. In: Abramowicz, W. (ed.) BIS 2007. LNCS, vol. 4439, pp. 424–437. Springer, Heidelberg (2007)
8. Clemmons, R.K.: Project Estimation With Use Case Points. CrossTalk–The Journal of Defense Software Engineering (February 2006)
9. Cockburn, A.: Writing effective use cases. Addison-Wesley, Boston (2001)
10. Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V.: GATE: A framework and graphical development environment for robust NLP tools and applications. In: Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (2002)
11. Diev, S.: Software estimation in the maintenance context. ACM SIGSOFT Software Engineering Notes 31(2), 1–8 (2006)
12. Dutoit, A.H., Paech, B.: Rationale-Based Use Case Specification. Requirements Engineering 7(1), 3–19 (2002)
13. Endres, A., Rombach, H.D.: A Handbook of Software and Systems Engineering: Empirical Observations, Laws, and Theories. Addison-Wesley, Reading (2003)
14. Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: Applications of linguistic techniques for use case analysis. Requirements Engineering 8(3), 161–170 (2003)
15. Fellbaum, C.: Wordnet: an electronic lexical database. Mit Pr (1998)
16. Harel, D.: Statecharts: A visual formalism for complex systems. Technical report, Weizmann Institute of Science, Dept. of Computer Science (1986)
17. Harry, A.: Formal Methods Fact File: VDM and Z. Wiley, Chichester (1996)
18. Jacobson, I.: Object-oriented development in an industrial environment. ACM SIGPLAN Notices 22(12), 183–191 (1987)
19. Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.: Object-oriented software engineering: A use case driven approach (1992)
20. Karner, G.: Resource Estimation for Objectory Projects. Objective Systems SF AB (copyright owned by Rational Software) (1993)
21. Klein, D., Manning, C.D.: Accurate unlexicalized parsing. In: Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics, pp. 423–430 (2003)

22. Kusumoto, S., Matukawa, F., Inoue, K., Hanabusa, S., Maegawa, Y.: Estimating effort by use case points: method, tool and case study. In: Proceedings. 10th International Symposium on Software Metrics, pp. 292–299 (2004)
23. Reif, W.: Formale methoden fur sicherheitskritische software - der kiv-ansatz. Informatik - Forschung und Entwicklung 14, 193–202 (1999)
24. Reisig, W.: Petri nets, an introduction. In: Salomaa, A., Brauer, W., Rozenberg, G. (eds.) EATCS, Monographs on Theoretical Computer Science, Berlin, Springer, Heidelberg (1985)
25. Ribu, K.: Estimating object-oriented software projects with use cases. Master's thesis, University of Oslo, Department of Informatics (2001)
26. Rolland, C., Achour, C.B.: Guiding the construction of textual use case specifications. Data Know Eng. 25(1), 125–160 (1998)
27. Schneider, G., Winters, J.P.: Applying use cases: a practical guide. Addison-Wesley Longman Publishing Co., Inc., Boston (1998)

A Collaborative Method for Reuse Potential Assessment in Reengineering-Based Product Line Adoption

Muhammad Asim Noor[1], Paul Grünbacher[2], and Christopher Hoyer[1]

[1] Institute for Systems Engineering & Automation,
Johannes Kepler University, 4040 Linz, Austria
{man,hoc}@sea.uni-linz.ac.at
[2] Christian Doppler Laboratory for Automated Software Engineering,
Johannes Kepler University, 4040 Linz, Austria
paul.gruenbacher@jku.at

Abstract. Software product lines are rarely developed from scratch. Instead the development of a product line by reengineering existing systems is a more common scenario, which relies on the collaboration of diverse stakeholders to lay its foundations. The paper describes a collaborative scoping approach for organizations migrating existing products to a product line. The approach uses established practices from the field of reengineering and architectural recovery and synthesizes them in a collaborative process. The proposed approach employs best practices and tools from the area of collaboration engineering to achieve effective collaboration. The paper presents a case study as initial validation of the proposed approach.

Keywords: Reuse potential assessment, collaboration, product line adoption, product line planning.

1 Introduction

Organizations usually do not start software product lines from scratch. It is more common that organizations with successful products in a particular domain find the need to adopt a product line (PL) approach to capitalize on systematic reuse of the common functionality among existing products [1, 2]. Existing systems are the result of large investment and can not be easily discarded as they embody substantial domain knowledge and expertise. The reuse of existing assets is critical in PL adoption as developing existing systems anew for a PL is typically expensive and risky [3]. Careful planning is thus needed for the success of product line adoption. It is essential to assess the suitability of existing assets for reuse in a product line and to estimate the effort required to tailor those assets. Software products are planned, designed, and developed collaboratively by diverse people. The knowledge essential to assess the reuse potential of existing products is distributed among architects, product managers, developers, or maintainers and spread in documents and application source code [4]. Numerous formal approaches for reuse potential assessment exist in areas such as software maintenance [5, 6] or software reengineering [7-10]. These methods focus on formally captured information in documentation, models, and source code. The *collaborative aspects of reuse potential assessment* have so far received only little attention. Although existing approaches related to reuse potential assessment [8, 11-13] are collaborative in nature they are rather vague with respect to how effective collaboration can be achieved.

In the paper we present a collaborative and stakeholder-centric approach to reuse potential assessment. It uses stakeholders' knowledge and experience of existing systems as primary sources of information to identify the existing components and to prioritize them according to their potential for reuse. The approach also enables the team to produce working estimates of the effort needed to modify those components. Such an approach is invaluable at PL scoping and planning stage. It is conducted at a high level of granularity (i.e., logical components, subsystem or packages) to avoid getting lost in technical details.

Boehm has argued that collaborative methods are key elements of future software engineering methods [14]. We thus believe that the collaborative approach nicely complements more formal approaches. It uses proven techniques and guideline from the discipline of collaboration engineering (CE) to achieve effective collaboration. CE is an approach to designing work practices for high-value collaborative tasks [15, 16]. In CE proven patterns of group collaboration, called thinkLets, are used to describe collaborative processes [15] and to foster the interaction of individuals and teams. ThinkLets describe collaborative techniques in a compact form and can be flexibly combined to achieve the desired results. For instance, there are thinkLets that concisely describe different brainstorming and prioritization techniques. It has been shown that different software engineering tasks can be supported by composing collaborative activities from thinkLets [17, 18]. While thinkLets might appear process-centric and tool-centric at first sight they should rather be seen as facilitation techniques that are optimized to structure high-value group tasks.

The remainder of the paper is structured as follow: In Section 2 we discuss related work in the field of product line adoption, software assessment for reengineering, and collaboration engineering. Section 3 presents layers 1 and 2 of our approach and explains how the approach is supported by thinkLets. In Section 4 we present a case study and discuss its results. A conclusion and an outlook to further research round out the paper.

2 Related Work

Many methods and techniques are reported in literature [5-7, 9, 10] for assessing existing software for maintenance and evolution. These methods aim at evaluating existing software with respect to business value and technical value to identify promising candidates for reengineering (e.g. [5]). The technical value is determined by variables such as maintainability, decomposability, deterioration, or obsolescence. The Product Line Practice Framework by Clements *et al.* [1] represents a comprehensive framework dealing with all aspects of product lines from development to evolution. It provides high level guidance for mining existing assets. The approach makes use of the options analysis for reengineering (OAR) method [8] and the mining architecture for product line (MAP) method [12] for identifying existing assets to be reused in product line development. However, the framework does not shed light on the collaborative aspects of this process [8, 12].

Bergey *et al.* [8] define software reengineering as "transforming an existing design of a software system (or element of that systems) to a new design while preserving the system's intrinsic functionality". Traditional reengineering approaches start with the analysis of legacy assets, the extraction of design and architectural information followed by an exploration of the options and possibilities, and the implementation of the best option (e.g. [13]). The *Horse shoe* model presented as part of *OAR* in [8] is an example of such an approach. SRAH [7] is a process for assessing legacy software to select the best options for legacy software evolution and to ease maintenance. The output of the process is a succinct report on which senior management can make informed decisions. Kolb *et al.* report on a case study [19] about the use of refactoring techniques to evolve and adapt existing components for reuse in a product line.

Several authors have addressed product line scoping and planning. For instance, Schmid [20] proposes a three staged approach comprised of product mapping, domain potential analysis and reuse infrastructure scoping. We presented a collaborative product mapping approach in the context of product line adoption in earlier work [4, 21, 22] based on Schmid's framework. In [23] Schmid explores the economic impact of product line adoption and evolution and identifies the four adoption strategies: big bang, project integration, reengineering-based, and leveraged (deriving a product line from another product line). In reengineering-based product line adoption the scoping activities gain a different focus as the product map guides the extraction of features from legacy systems as suggested in [4, 21, 22]. Other work on product line adoption can be found in a case study by Bayer *et al.* [24] who report on a migration process guided by the RE-PLACE approach. In [25] Ebert *et al.* identify a clear business focus, strong release planning, and requirements management as success criteria for product line adoption. Kircher *et al.* in [26] discuss challenges in product line adoption and report a set of best practices.

The discipline of collaboration engineering provides a wide range of practices, patterns and tools to achieve effective collaboration. Collaboration engineering aims at designing work practices for practitioners to support high-value recurring collaborative tasks [15]. There are six general patterns of collaboration: generate, reduce, clarify, organize, evaluate, and build consensus [27]. The approach tries to provide the efficiency and effectiveness of professional facilitators to the practitioners who are not experts in team interaction. ThinkLets [16] describe patterns for collaborative activities and have become widely accepted building blocks for designing collaborative processes. A thinkLet is a named, scripted, and well-tested activity that produces a known pattern of collaboration among people working together on a common goal [28]. There are currently about 70 well-documented thinkLets [29] some of which are used in our approach to reuse potential assessment. Many collaborative processes have been successfully designed using thinkLets. An example is the requirements negotiation method EasyWinWin which incorporates a number of agile principles [30]. Our earlier work [4, 21, 22] also suggest that collaborative techniques are valuable and useful in product line planning.

3 A Collaborative Process for Reuse Potential Assessment

Fig. 1 shows the involved participants, inputs and outputs of the Reuse Potential Assessment (RPA) process. The process relies on the knowledge and experience of the participants, available documentation and analyses of the systems to be reused, and the proposed product map of the future product line.

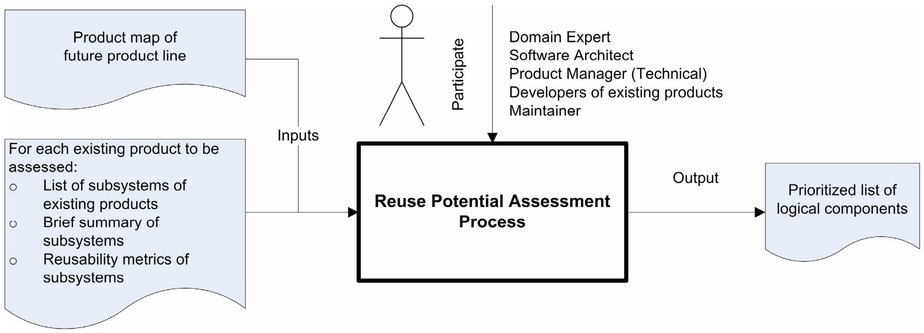

Product map of future product line
For each existing product to be assessed:
List of subsystems of existing products
Brief summary of subsystems
Reusability metrics of subsystems
Inputs
Participate
Domain Expert
Software Architect
Product Manager (Technical)
Developers of existing products
Maintainer
Reuse Potential Assessment Process
Output
Prioritized list of logical components

Fig. 1. Participants, inputs and output of the reuse potential assessment process

The selection of the right participants is a key factor for the success of the collaborative RPA process [31]. The selection must be based upon the knowledge, experience and expertise of people with the products to be assessed. The inclusion of domain experts and software architects in the team is essential. The number of technical experts needed for the RPA process for a particular product depends on the size and complexity of the products to be assessed.

A product map as defined in [4, 21, 22] is an important input to the RPA process. It is used to ensure a shared understanding about the common functionality among the products of the PL and helps the team to identify logical components from the existing systems. A further input to the RPA process is a list of subsystem for each of the products to be assessed. A brief summary explaining the functionality of the subsystems is also provided. Furthermore, reusability metrics for the subsystems are extracted beforehand and are provided as part of the subsystem summary. Similar to existing models for reuse potential assessment [5, 8, 10] our RPA process makes use of static analyses of the existing systems. The metrics used to evaluate the reusability are size (e.g., file size method size), complexity (e.g., cyclomatic complexity, boolean expression complexity, nesting levels), decomposability (e.g., n tier architecture), dependencies (e.g., data abstraction coupling, fan-out), or understandability (e.g., ratio of non commented line of code, naming) [5]. The static analysis can be facilitated by tools, e.g., Checkstyle[1]. Commercial IDEs (e.g. IntelliJIDEA) support static analysis on the desired levels of abstraction (e.g., method, class, or package).

We describe the collaborative RPA process on three layers of abstraction: Fig. 2 shows the highest layer 1, i.e., the tasks of the process, input and output of the tasks, the collaboration patterns used in the execution of the task, and the sequence of the

[1] http://eclipse-cs.sourceforge.net/index.shtml

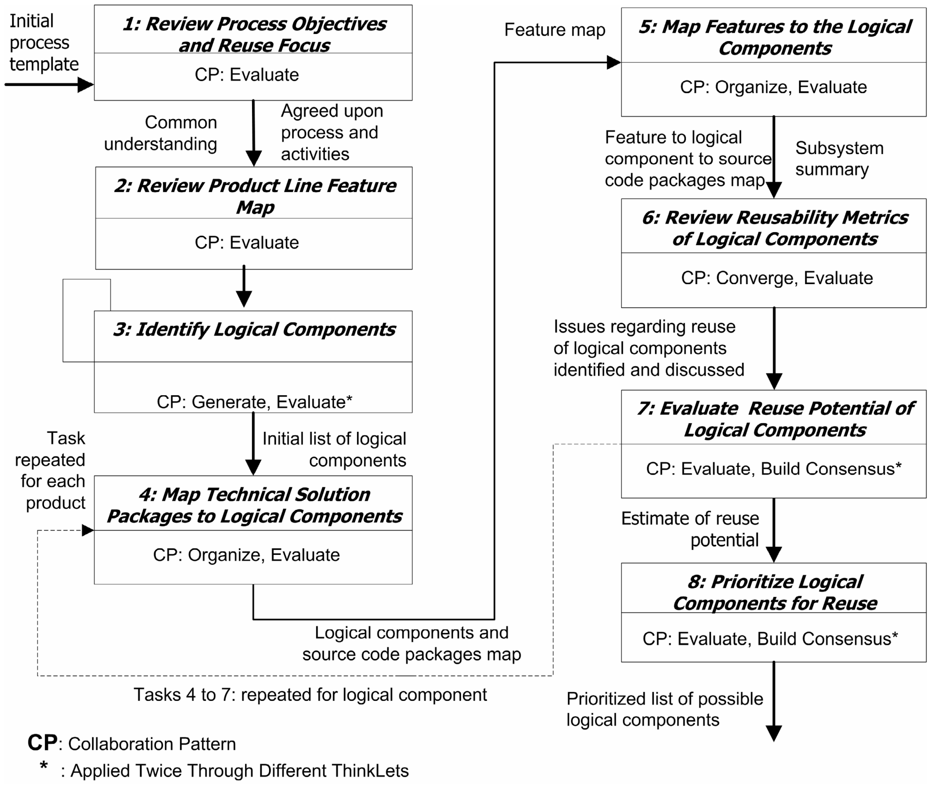

Initial process template
1: Review Process Objectives and Reuse Focus
CP: Evaluate
Common understanding
Agreed upon process and activities
2: Review Product Line Feature Map
CP: Evaluate
3: Identify Logical Components
CP: Generate, Evaluate*
Initial list of logical components
Task repeated for each product
4: Map Technical Solution Packages to Logical Components
CP: Organize, Evaluate
Logical components and source code packages map
Tasks 4 to 7: repeated for logical component
Feature map
5: Map Features to the Logical Components
CP: Organize, Evaluate
Feature to logical component to source code packages map
Subsystem summary
6: Review Reusability Metrics of Logical Components
CP: Converge, Evaluate
Issues regarding reuse of logical components identified and discussed
7: Evaluate Reuse Potential of Logical Components
CP: Evaluate, Build Consensus*
Estimate of reuse potential
8: Prioritize Logical Components for Reuse
CP: Evaluate, Build Consensus*
Prioritized list of possible logical components
CP: Collaboration Pattern
* : Applied Twice Through Different ThinkLets

Fig. 2. Task view of the RPA process (Layer 1)

execution of the tasks. At layer 2, we show how the tasks and collaboration patterns are supported by thinkLets (cf. Fig. 3). Layer 3 (cf. case study section 4) describes a concrete enactment of the process during a pilot case study demonstrating the actual use of collaborative tools.

Fig. 2 shows the process tasks and their associated thinkLets. There are seven thinkLets used in the process: The thinkLet *ReviewReflect* facilitates a group to review an outline or a document. Team members collaboratively go through the outline and record their thoughts and suggestion by adding comments. Right after, these ideas are discussed in a moderated fashion. Consolidated recommendations are prepared or changes to the documents are made. The thinkLet *BucketWalk* aims at achieving a shared vision amongst groups of people by a collaborative walkthrough of all the items in different categories while encouraging the discussion for issues and demanding explanation. The group does not move forward before open issues are resolved. The thinkLet *LeafHopper* aims at eliciting ideas from participants regarding a set of topics. The thinkLet *PopcornSort* helps structuring collected raw ideas into appropriate categories.

The thinkLet *StrawPoll* enables decision-making through measuring opinions of the participants in quantitative terms. A wide variety of voting methods can be used

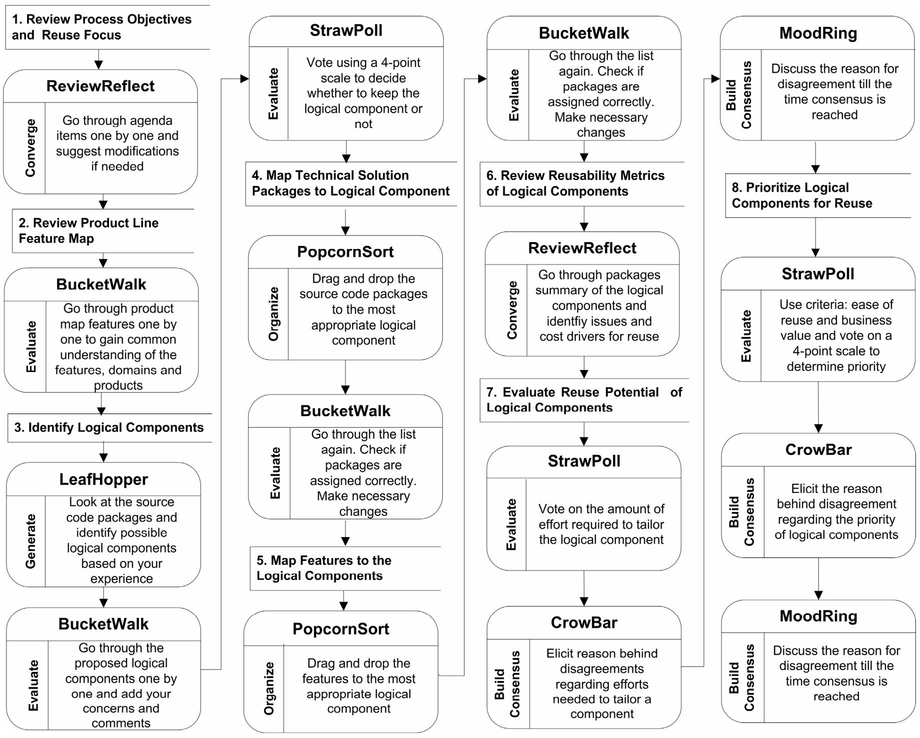

1. Review Process Objectives and Reuse Focus
ReviewReflect
Converge
Go through agenda items one by one and suggest modifications if needed
2. Review Product Line Feature Map
BucketWalk
Evaluate
Go through product map features one by one to gain common understanding of the features, domains and products
3. Identify Logical Components
LeafHopper
Generate
Look at the source code packages and identify possible logical components based on your experience
BucketWalk
Evaluate
Go through the proposed logical components one by one and add your concerns and comments
StrawPoll
Evaluate
Vote using a 4-point scale to decide whether to keep the logical component or not
4. Map Technical Solution Packages to Logical Component
PopcornSort
Organize
Drag and drop the source code packages to the most appropriate logical component
BucketWalk
Evaluate
Go through the list again. Check if packages are assigned correctly. Make necessary changes
5. Map Features to the Logical Components
PopcornSort
Organize
Drag and drop the features to the most appropriate logical component
BucketWalk
Evaluate
Go through the list again. Check if packages are assigned correctly. Make necessary changes
6. Review Reusability Metrics of Logical Components
ReviewReflect
Converge
Go through packages summary of the logical components and identfiy issues and cost drivers for reuse
7. Evaluate Reuse Potential of Logical Components
StrawPoll
Evaluate
Vote on the amount of effort required to tailor the logical component
CrowBar
Build Consensus
Elicit reason behind disagreements regarding efforts needed to tailor a component
MoodRing
Build Consensus
Discuss the reason for disagreement till the time consensus is reached
8. Prioritize Logical Components for Reuse
StrawPoll
Evaluate
Use criteria: ease of reuse and business value and vote on a 4-point scale to determine priority
CrowBar
Build Consensus
Elicit the reason behind disagreement regarding the priority of logical components
MoodRing
Build Consensus
Discuss the reason for disagreement till the time consensus is reached

Fig. 3. ThinkLets view of the process (Layer II)

with this thinkLet. The thinkLet *CrowBar* helps to elicit reasons for discord. It is usually used after the thinkLet *StrawPoll*, which highlights the agreements and disagreements resulting from certain issues. The thinkLet *MoodRing* aims at building consensus. It is usually used together with the thinkLet *CrowBar*, which highlights reasons for disagreement. These reasons are discussed in a moderated fashion. During the discussion persons originally disagreeing can change their mind and change their vote anonymously.

More specifically the purpose of each task in the collaborative process is as follows:

Task 1: Review Process Objectives and Reuse Focus. The facilitator (i) communicates to the participants the objectives of the overall process and the agenda and (ii) fine-tunes the process in light of the participants' input. The involved stakeholders include product managers, architects, developers, maintainers and domain experts. The participants collaboratively review each task of the agenda. The team agrees on the focus of reuse at this stage, i.e., the principal elements of interest for a team meeting (e.g., particular areas of the source code, algorithms, GUIcomponents, documentation, test cases or test data, etc.). This task is supported by the thinkLet *ReviewReflect.*

Task 2: Review Product Line Feature Map. The RPA process relies on a product map as input, which can be defined collaboratively as outlined in [4, 21, 22]. It is important for the team to develop a shared vision regarding the scope and vision of the product line before mining for potential assets. The participants familiarize themselves with the product map of the product line under development, which is described in terms of features, domains and products. They use the thinkLet *BucketWalk* to collaboratively navigate through the product map and to suggest changes. This helps to create a shared vision among participants regarding the scope and structure of the product line. This knowledge is essential to identify correct logical components, which may be suitable to be reused as PL core assets. Schmid [20] suggests to use domains to ease the task of identifying core assets for a PL. Domains are defined as relatively independent coherent clusters of functionality that contain one or more features. Features represent externally visible characteristics of systems (e.g., software project tracking in a product line for project management).

Task 3: Identify Logical Components. In this task participants identify logical components from existing products, which are then further investigated for their suitability to be included in the product line. A logical component can be seen as an abstract core asset of the envisaged PL. A logical component can be realized by either adapting existing implementation or by developing them anew. However, the focus of our approach is on reuse of existing assets. That is why we discuss only the case of adaptation. The challenge of this task is to identify solution elements (e.g., classes, modules, subsystems, libraries) from existing systems as candidate core assets. It is essential to balance the desire for high quality core assets and the effort required to adapt the existing technical implementation. The output of the task is a list of logical components for every existing product. The task is based on a collaborative walk-through of the modules of a product. For instance, in case of a Java-based system, modules are packages with a brief summary and the list of constituent classes. Assuming that the participants are well familiar with the products, they identify candidate logical components based on the information presented to them. The collection of the logical components is accomplished by executing the thinkLet *LeafHopper* which uses directed brainstorming. The participants brainstorm candidate components to a shared list. People see what other logical components have been suggested by other participants and they can add comments. The thinkLet *BucketWalk* facilitates common understanding and refinement of the list of logical components through moderated discussion. The thinkLet *StrawPoll* (electronic voting) may be conducted to reach consensus within the team whether a proposed logical component should be kept for further investigation or not. This task is repeated for every existing product and results into a list of logical components for every investigated system.

Task 4: Map Technical Solution Packages to Logical Components. Participants identify links between the logical components and the technical solution packages of the existing product. For instance, dependencies are established between existing modules or subsystems which implement the functionality of a logical component. The scope and definition of logical components are refined where necessary. The thinkLet *PopcornSort* facilitates the assignment of implementation units (e.g., modules or classes or packages) to the appropriate logical components and helps bounding the scope of the logical component. The thinkLet *BucketWalk* ensures consensus among participants about the appropriateness of the scope of the logical components. This task is repeated for each prospective product and results in improved definitions of the logical components. The results can be refined in task 6 when performing a more detailed analysis of the existing system. This task can be supported by feature location approaches (e.g., [32]) or scenario-based traced analysis techniques [33] depending on the complexity of the system and the knowledge of the stakeholders.

Task 5: Map Features to the Logical Components. Participants identify links between logical components and the features from the product map. This task is performed in a similar manner as the previous task using the thinkLets *PopcornSort* and *BucketWalk.* Each feature is assessed and assigned to the appropriate logical component. The task is repeated for every product. The goal of tasks 4 and 5 is to establish initial coarse-grain traceability between features, logical components and source code. This traceability allows the visualization of the logical components and eases later design activities. Even if traceability links exist (e.g., between requirements, design artefacts and source code) the above two tasks may be performed to take into account the new abstraction layer of logical components.

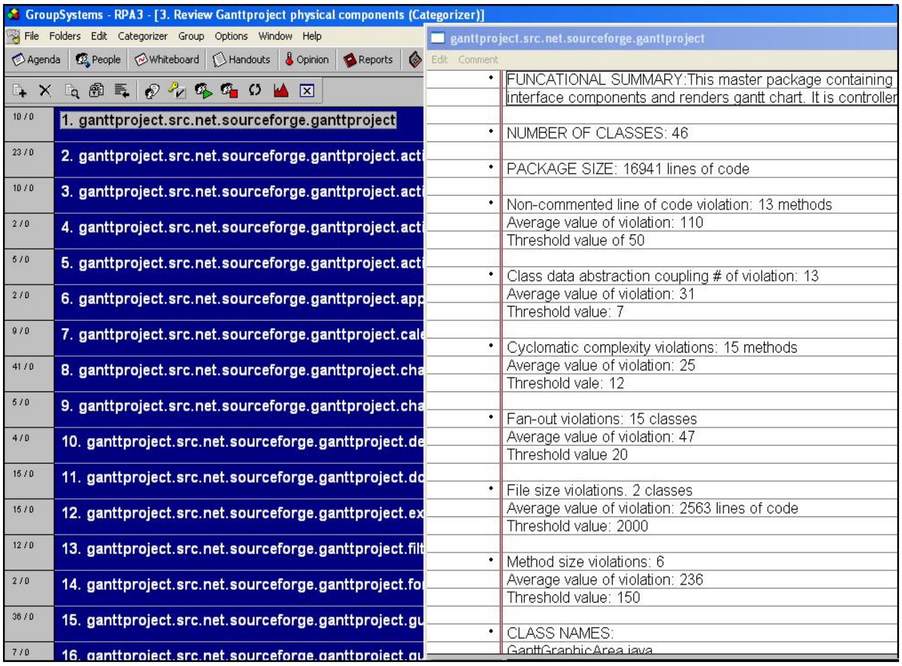
Task 6: Review Reusability Metrics of Logical Components. Participants review the logical components using the thinkLet *ReviewReflect.* For each package thought to be reusable in the logical component, they go through the information provided in the subsystem summary. Mainly, the different implications for the efforts required for reusing the package are reviewed. First, participants collaboratively go through the information (functional summary and metrics of each package) and add their opinion. Questions they try to answer include 'What are the challenges in reusing this package?' or 'What reengineering techniques are suitable for this package?'. Later, the collected comments are discussed in a moderated fashion and a consolidated list of issues and possible solution is created for each logical component.

Task 7: Evaluate the Reuse Potential of Logical Components. Participants estimate the costs and effort required to adapt the logical component as a core asset of the product line. The reuse effort estimation of the participants can be elicited through the thinkLet *StrawPoll,* where participant have to assess the level of effort required to tailor a particular logical component. In case stakeholders cannot agree about the efforts required to tailor a certain logical components the thinkLet *CrowBar* is used followed by the thinkLet *MoodRing* to reveal the reasons for disagreement. This process is repeated for each logical component. The result of this task is an initial estimate of cost/effort for adapting the logical components. These estimates are intended for selecting the most promising components for adaptation during planning while more formal cost estimation approaches can be used at design time. The task is repeated for each product.

Task 8: Prioritize Logical Components for Reuse. In this task the logical components are prioritized for further investigation at design time and later adaptation. It is accomplished through the thinkLet *StrawPoll,* which is conducted by electronicvoting. Participants assign values on a scale of 4 to each logical component. Two parameters are used: (i) the value of reuse and (ii) the effort required to reuse the logical component. Logical components which require less tailoring effort and have the highest business value will be assigned a top priority. In case of significant differences of opinion the thinkLets *CrowBar* and *MoodRing* are executed as in the previous task. This task concludes the RPA process and produces a list of the mostpromising candidate logical components for further adaptation and refinement as core assets.

4 Initial Evaluation

We conducted a pilot case study to assess the usefulness of the proposed collaborative process and the usability of the supporting tools. The study was a fictitious organisation developing a product line for project management tools based on open source code assets. In order to define the desired product line, a group of three domain experts developed a product map containing 120 features, 14 domains and three products for the project management domain. The feasibility study was based on the open source systems Gantt Project[2] and Project Factory[3]: The size of Gantt Project is 51 packages, 492 classes and is 63 KLOC. The size of Project Factory is 16 packages, 140 classes and 29 KLOC.


GroupSystems - RPA3 - [3. Review Ganttproject physical components (Categorizer)]
File Folders Edit Categorizer Group Options Window Help
Agenda People Whiteboard Handouts Opinion Reports
ganttproject.src.net.sourceforge.ganttproject
Edit Comment

```
10 / 0   1. ganttproject.src.net.sourceforge.ganttproject
23 / 0   2. ganttproject.src.net.sourceforge.ganttproject.acti
10 / 0   3. ganttproject.src.net.sourceforge.ganttproject.acti
2 / 0    4. ganttproject.src.net.sourceforge.ganttproject.acti
5 / 0    5. ganttproject.src.net.sourceforge.ganttproject.acti
2 / 0    6. ganttproject.src.net.sourceforge.ganttproject.app
9 / 0    7. ganttproject.src.net.sourceforge.ganttproject.cal
41 / 0   8. ganttproject.src.net.sourceforge.ganttproject.cha
5 / 0    9. ganttproject.src.net.sourceforge.ganttproject.cha
4 / 0    10. ganttproject.src.net.sourceforge.ganttproject.de
15 / 0   11. ganttproject.src.net.sourceforge.ganttproject.dc
15 / 0   12. ganttproject.src.net.sourceforge.ganttproject.ex
12 / 0   13. ganttproject.src.net.sourceforge.ganttproject.filt
2 / 0    14. ganttproject.src.net.sourceforge.ganttproject.fo
36 / 0   15. ganttproject.src.net.sourceforge.ganttproject.gu
7 / 0    16. ganttproject.src.net.sourceforge.ganttproject.gu
```

```
• FUNCATIONAL SUMMARY:This master package containing
  interface components and renders gantt chart. It is controller
• NUMBER OF CLASSES: 46
• PACKAGE SIZE: 16941 lines of code
• Non-commented line of code violation: 13 methods
  Average value of violation: 110
  Threshold value of 50
• Class data abstraction coupling # of violation: 13
  Average value of violation: 31
  Threshold value: 7
• Cyclomatic complexity violations: 15 methods
  Average value of violation: 25
  Threshold vale: 12
• Fan-out violations: 15 classes
  Average value of violation: 47
  Threshold value 20
• File size violations. 2 classes
  Average value of violation: 2563 lines of code
  Threshold value: 2000
• Method size violations: 6
  Average value of violation: 236
  Threshold value: 150
• CLASS NAMES:
  GanttGraphicArea.java
```

Fig. 4. Package summary containing reusability metrics

Three engineers participated in this case study. The process was conducted in three workshops with duration of approximately 4 hours each. As preparation for the workshop the moderator (i) defined the agenda according to the tasks identified in this paper, (ii) uploaded the feature map of the product line to the collaboration tool GroupSystems, and (iii) uploaded the package list and package summaries containing the reusability metrics to the collaborative tool (see Figure 4). The feature map was developed prior to the workshop following the method described in [4, 21, 22].

[2] http://sourceforge.net/projects/ganttproject
[3] http://sourceforge.net/projects/projectfactory/

Table 1. Metrics applied in the static analysis

Metric	Level	Description	Value
Cyclomatic Complexity	Method	A measure for the minimum number of possible paths through the source and therefore the number of required tests.	10
Coupling	Class	A measure for the number of instantiations of other classes within the given class.	7
Fan-out	Class	A measure for the number of other classes a given class relies on.	20
NCSS	Method	A measure for the number of non-commented source statement within a method.	50
BEC	Line	A measure for the number of &&, || and ^ in an expression. (BEC = Boolean expression complexity).	3
File Size	Class	A measure of the file size in lines of code.	2000
Method Size	Method	A measure for the method size in lines of code.	150

Table 2. Gantt project metrics

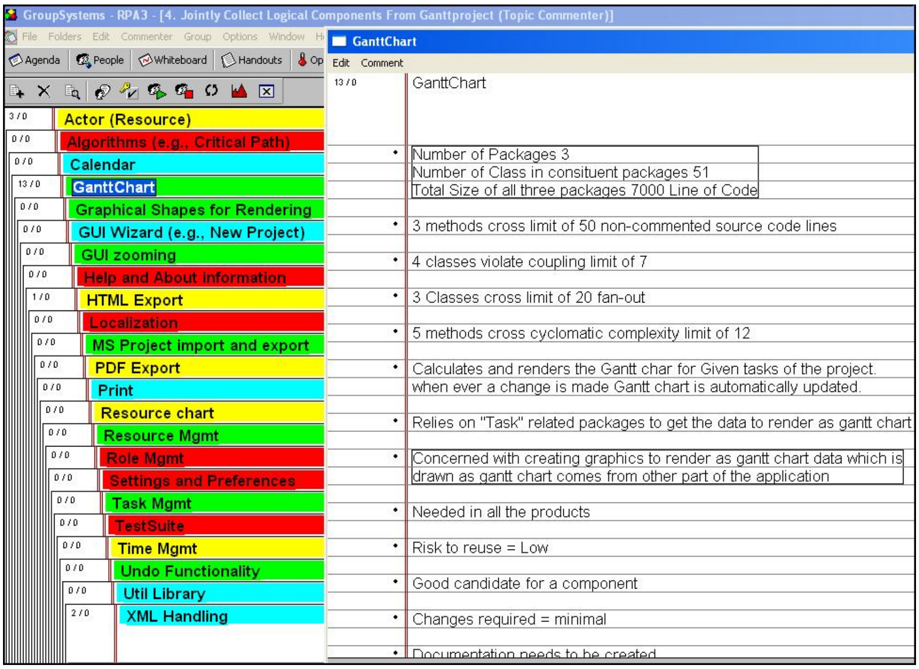
Metric	# of violations	Average violations value
Cyclomatic complexity	45	15
Coupling:	51	18
Fan-out	36	38
NCSS (Non-commented lines of code)	35	97
BEC (Boolean expression complexity)	3	9
File Size	2	2539
Method Size	12	230

In order to extract the reusability metrics a static analysis was conducted by one software engineer for both products. Source code metrics such as cyclomatic complexity, data abstraction coupling, fan-out, non commented line of code, size of the class and size of the methods, are reported in literature to be useful indicators of the reusability of the code [19, 34]. Generally, it is assumed that the lower the value of above mentioned metrics the more reusable is that source code [34]. These combined metrics were used to complement the overall picture and help to identify the reusable software elements.

Cyclomatic complexity, non-commented line of code, and size of method were measured at method level. The remainder at class level. We used the open source tool CheckStyle (available as an Eclipse plug-in) to calculate these metrics. The default threshold values of CheckStyle (for above mentioned metrics) were used. These values concur with existing literature in the field of software maintenance and evolution [19, 35]. Table 1 shows a brief description of the metrics, along with the default threshold values of CheckStyle.

Table 2 shows the consolidated summary of these metrics for one of the products assessed, i.e., Gantt project. For example, among 492 classes of Gantt Project 51 classes have a class data abstraction coupling of more than 7. Among these 51 classes the average class data abstraction coupling is 18. Further analysis shows that these are mainly those classes which primarily deal with the GUI (in the case of Gantt Project they use Java Swing components). This indicates coupling is not a big hindrance when reusing the Gantt Project source code.

Overall, these metrics indicate that the Gantt Project implementation is not overly complex as only 45 methods exceed the threshold complexity value. Mostly, these


GroupSystems - RPA3 - [4. Jointly Collect Logical Components From Ganttproject (Topic Commenter)]
File Folders Edit Commenter Group Options Window H
Agenda People Whiteboard Handouts Op
GanttChart
Edit Comment

3 / 0	Actor (Resource)
0 / 0	Algorithms (e.g., Critical Path)
0 / 0	Calendar
13 / 0	GanttChart
0 / 0	Graphical Shapes for Rendering
0 / 0	GUI Wizard (e.g., New Project)
0 / 0	GUI zooming
0 / 0	Help and About information
1 / 0	HTML Export
0 / 0	Localization
0 / 0	MS Project import and export
0 / 0	PDF Export
0 / 0	Print
0 / 0	Resource chart
0 / 0	Resource Mgmt
0 / 0	Role Mgmt
0 / 0	Settings and Preferences
0 / 0	Task Mgmt
0 / 0	TestSuite
0 / 0	Time Mgmt
0 / 0	Undo Functionality
0 / 0	Util Library
2 / 0	XML Handling

13 / 0 GanttChart

- Number of Packages 3
  Number of Class in consituent packages 51
  Total Size of all three packages 7000 Line of Code
- 3 methods cross limit of 50 non-commented source code lines
- 4 classes violate coupling limit of 7
- 3 Classes cross limit of 20 fan-out
- 5 methods cross cyclomatic complexity limit of 12
- Calculates and renders the Gantt char for Given tasks of the project. when ever a change is made Gantt chart is automatically updated.
- Relies on "Task" related packages to get the data to render as gantt chart
- Concerned with creating graphics to render as gantt chart data which is drawn as gantt chart comes from other part of the application
- Needed in all the products
- Risk to reuse = Low
- Good candidate for a component
- Changes required = minimal
- Documentation needs to be created

Fig. 5. Consolidate list of issues of a logical component (output of task 6)

methods handle XML tags as data is stored in XML files. Most classes are within an acceptable range of coupling and fan-out. There are only two classes and 12 methods which violate the modest limit (2000, 150 LOC). The code is mostly well commented. The extracted metrics indicate that components can be extracted from Gantt Project implementation without excessive difficulty. Similar metrics were extracted from Project Factory but are not reported in this paper due to space limitations. These metrics were added in the package summary for easy perusal of the participants. The package level summary serves as a quick reference of the package implementation and is used to define the logical components (task 3) and to review the reusability metrics of the logical component (task 6).

In the following, we describe the enactment of each task in the pilot case study: The first two tasks were performed once in the beginning whereas task 3 was repeated for the two analyzed products. Tasks 4 to 7 were repeated for each logical component. The first tasks (*Review process objective and reuse focus* and *Review product line feature map*) were accomplished by conducting the thinkLets *ReviewReflect* and *BucketWalk* respectively. The small number of participants who had already jointly developed the process and product map earlier simplified these tasks. However, in more realistic settings a presentation about the agenda would be needed to explain the purpose and objectives of the exercise. The focus of the reuse was on GUI elements, algorithms and cohesive functionality (e.g., forecasting, Gantt chart) in the source code.

The third task was to identify logical components from the source code. In Fig. 5 coloured entries represent the identified logical components. Participants used the thinkLet *LeafHopper* to identify the logical components. This task was performed for both products simultaneously. In total the team identified 23 logical components from Gantt Project. Many logical components were later found to be of too limited size or use but no new components were added. Different team members had identified logical components at different level of granularity, which did not raise problems as logical components were refined in subsequent steps. Logical components were also filtered out if considered as inappropriate based on selected criteria (e.g. the minimum size of the implementation encompassed by the logical component).

Due to the tight schedule of the team members at the time of the case study task 4 (*Map Technical Solution Packages to Logical Components)* and task 5 (*Map Features to the Logical Components*) were performed asynchronously. This allowed creating a better visualization of the logical components, which was also necessary to support the detailed analysis in task 6 as detailed traceability reports where unavailable for the selected open source applications.

Task 6 (*Review reusability metrics of logical component*) was accomplished by conducting the thinkLet *ReviewReflect*. The participants created a consolidated list of issues and opinions for each logical component as shown in Fig. 5. This task aimed at collecting information about the reusability of packages (in order to realize the logical component) from the calculated metrics and the technical knowledge and experience of the people with these packages.

Task 7 (*Evaluate the Reuse Potential of Logical Components*) is supposed to be performed directly after task 6 so that participants still have the findings of the previous step in their minds. However, in our case study the team did not perform this task. Estimates of effort and cost can only be made on the basis of above mentioned information if the team has in depth knowledge of the system. Such knowledge is available only for people that have been involved in the design, development or maintenance of the product. The teams in our case did not have such an intimate knowledge. Instead, the prioritization of the logical components was performed directly after task 7.

Table 3. Prioritized high level logical components

Logical Component	# of Packages	# of Classes	Total Size
GUI Components	11	103	13 KLOC
Task Management	6	67	7 KLOC
Gantt Chart	3	51	7 KLOC
IO Handling	4	51	6 KLOC
Calendar and Time Mgmt	3	31	3 KLOC
Actor (resource) Mgmt	2	19	2 KLOC
Test Suite	1	22	2 KLOC
Project Forecast (Factory)	NA	2	1 KLOC
Project Management	1	46	17 KLOC

Lastly, the prioritization (task 8) was done using a collaborative voting tool. First, the team members assessed the business value and the ease of reuse of each component on a scale of 4. The average of these two values determined the priority of the component. Table 3 shows the list of prioritized high level logical components beginning with the components having highest priority.

These components have a similar business value as all are important in a project management application. Their priority is mainly determined on the basis of ease of reuse.

We started the case study with two products offering quite similar functionality. We aimed at identifying reusable components from these two products which can be modified and used as core assets of a product line in the project management domain. Out of 9 components identified for reuse, 8 come from Gantt Project and only the component "Project Forecast" comes from Project Factory.

It is essential that people with intimate technical knowledge of the products participate in the reuse potential assessment. Without such knowledge identifying relevant logical components and creating traceability links between features, logical component and physical component of the technical solution is difficult. The case study team also suffered to some extent from these problems due to a lack of in-depth knowledge of the products. It identified 9 components that can be reused as core assets in the product line (as shown in table 3) based on an initial list of 24 and 13 candidate components from Gantt Project and Factory respectively.

5 Conclusions and Future Work

We presented a collaborative approach for reuse potential analysis that is intended to complement more formal approaches for reengineering legacy assets in the area of product line planning. The process aims at supporting a team to collaboratively identify components with a high reuse potential from different legacy products. The process also increases the understanding of traceability and the dependencies between features and technical solution components and provides initial estimates for the effort of reuse. The presented process relies on the careful selection of stakeholders to ensure the knowledge and experience required. Absence of such knowledge and experience will undermine the collaborative aspects of the process and force the team to rely on more formal approaches, i.e., reverse engineering.

Due to our experience with collaboration engineering methods and thinkLets in other areas of software engineering such as requirements negotiation, risk management, or software inspection we expect this collaborative process to scale well in a real-world product line setting. The experience gained in the feasibility study confirms these findings. We will use the process in near future with an industrial partner specialized in ERP solutions who is currently shifting to a product line approach. The experiences also confirm that thinkLets can be effectively supported by collaborative tools, in our case a Group Support System (GSS[4]) tool was used to support the stakehoder collaboration.

References

1. Clements, P., Northrop, L.: Software product lines: practices and patterns. Addison-Wesley, Reading (2002)
2. Boeckle, G., Clements, P., McGregor, J.D., Muthig, D., Schmid, K., Siemens, A.G., Munich, G.: Calculating ROI for software product lines. IEEESoftware 21(3), 23–31 (2004)

[4] http://www.groupsystems.com

3. Aversano, L., Tortorella, M.: An assessment strategy for identifying legacy system evolution requirements in eBusiness context. J. Softw. Maint. Evol.: Res. Pract. 16, 255–276 (2004)
4. Noor, M.A., Grünbacher, P., Briggs, R.O.: Defining a Collaborative Approach for Product Line Scoping: A Case Study in Collaboration Engineering. In: IASTED Conference on Software Engineering (SE 2007) Innsbruck, Austria, February 13 (2007)
5. De Lucia, A., Fasolino, A.R., Pompelle, E.: A decisional framework for legacy system management. In: Proceedings of IEEE International Conference on Software Maintenance, pp. 642–651 (2001)
6. Ransom, J., Sommerville, I., Warren, I.: A Method for Assessing Legacy Systems for Evolution. In: Proceedings of Reengineering Forum, p. 98 (1998)
7. Software Engineering Assessment HandBook Version 3, DoD US (1997) last checked: 9-08-07, http://www.swen.uwaterloo.ca/~kostas/ECE750-3/srah.pdf
8. Bergey, J.K., O'Brien, L., Smith, D.: Options Analysis for Reengineering (OAR): A Method for Mining Legacy Assets 2001. Carnegie Mellon University, Software Engineering Institute
9. Caldiera, G., Basili, V.R.: Identifying and qualifying reusable software components. IEEE Computer 24(2), 61–70 (1991)
10. Sneed, H.M.: Planning the reengineering of legacy systems. IEEE Software 12(1), 24–34 (1995)
11. De Baud, J.M., Flege, O., Knauber, P.: PuLSE-DSSA—a method for the development of software reference architectures. In: Proceedings of the third international workshop on Software architecture, pp. 25–28 (1998)
12. O'Brien, L., Smith, D.: MAP and OAR Methods: Techniques for Developing Core Assets for Software Product Lines from Existing Assets. Carnegie Mellon University, Software Engineering Institute (2002)
13. Stoermer, C., O'Brien, L.: MAP-Mining Architectures for Product Line Evaluations. In: Proceedings of the IEEE/IFIP Working Conference on Software Architectures, Amsterdam, The Netherlands, August 2001, pp. 35–44 (2001)
14. Boehm, B.: A view of 20th and 21st century software engineering. In: Proceeding of the 28th international conference on Software engineering (ICSE 2006), pp. 12–29 (2006)
15. Briggs, R.O., d.V.G.J., Nunamaker, J.J.F.: Collaboration Engineering with ThinkLets to Pursue Sustained Success with Group Support Systems. Journal of Management Information Systems 19(4), 31–64 (2003)
16. Briggs, R.O., De Vreede, G.J., Nunamaker Jr, J.F., Tobey, D.: ThinkLets: achieving predictable, repeatable patterns of group interaction with group support systems (GSS). In: Proceedings of the 34th Annual Hawaii International Conference on System Sciences, p. 9 (2001)
17. Grünbacher, P., Halling, M., Biffl, S.: An empirical study on groupware support for software inspection meetings. In: 18th IEEE International Conference on Automated Software Engineering, pp. 4–11 (2003)
18. Grünbacher, P., Seyff, N., Briggs, R.O., In, H.P., Kitapci, H., Port, D.: Making every student a winner: The WinWin approach in software engineering education. Journal of Systems and Software 80(8), 1191–1200 (2007)
19. Kolb, R., Muthig, D., Patzke, T., Yamauchi, K.: Refactoring a legacy component for reuse in a software product line: a case study. Journal of Software Maintenance and Evolution: Research and Practice 18, 109–132 (2006)
20. Schmid, K.: A comprehensive product line scoping approach and its validation. In: Proceedings of the 24th International Conference on Software Engineering, pp. 593–603 (2002)
21. Noor, M.A., Rabiser, R., Grünbacher, P.: A Collaborative Approach for Reengineering-based Product Line Scoping in APLE - 1st International Workshop on Agile Product Line Engineering 2006, Baltimore, Maryland (2006)

22. Noor, M.A., Rabiser, R., Grünbacher, P.: Agile Product Line Planning: A Collaborative Approach and a Case Study. Journal of Systems and Software (to appear), doi:10.1016/j.jss.2007.10.028
23. Schmid, K., Verlage, M.: The Economic Impact of Product Line Adoption and Evolution. IEEE Software 19(4), 50–57 (2002)
24. Bayer, J., Girard, J.F., Wuerthner, M., De Baud, J.M., Apel, M.: Transitioning legacy assets to a product line architecture. ACM SIGSOFT Software Engineering Notes 24(6), 446–463 (1999)
25. Ebert, C., Smouts, M.: Tricks and Traps of Initiating a Product Line Concept in Existing Products. In: Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), pp. 520–525 (2003)
26. Kircher, M., Schwanninger, C., Groher, I.: Transitioning to a Software Product Family Approach - Challenges and Best Practices. In: 10th International Software Product Line Conference, 2006, pp. 163–171 (2006)
27. Briggs, R.O., Kolfschoten, G.L., Vreede, G.J.d., Dean, D.L.: Defining Key Concepts for Collaboration Engineering. In: Americas Conference on Information Systems, AIS, Acapulco (2006)
28. De Vreede, G.J., Kolfschoten, G.L., Briggs, R.O.: ThinkLets: a collaboration engineering pattern language. International Journal of Computer Applications in Technology 25(2), 140–154 (2006)
29. Kolfschoten, G.L., Appelman, J.H., Briggs, R.O., de Vreede, G.J.: Recurring patterns of facilitation interventions in GSS sessions. In: Proceedings of the 37th Annual Hawaii International Conference on System Sciences, pp. 19–28 (2004)
30. Boehm, B.W., Ross, R.: Theory-W software project management principles and examples. IEEE Transactions on Software Engineering 1989 15(7), 902–916 (1989)
31. Harder, R.J., Keeter, J.M., Woodcock, B.W., Ferguson, J.W., Wills, F.W.: Insights in Implementing Collaboration Engineering. In: Proceedings of the 38th Annual Hawaii International Conference on System Sciences, HICSS 2005, p. 15b (2005)
32. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. IEEE Transactions on Software Engineering 29(3), 210–224 (2003)
33. Egyed, A.: A Scenario-Driven Approach to Traceability. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE), Toronto, Canada, pp. 123–132 (2001)
34. Barnard, J.: A new reusability metric for object-oriented software. Software Quality Journal 7, 35–50 (1998)
35. McCabe, T.J., Butler, C.W.: Design complexity measurement and testing. Communications of the ACM 32(12), 1415–1425 (1989)

Corporate-, Agile- and Open Source Software Development: A Witch's Brew or An Elixir of Life?

Morkel Theunissen, Derrick Kourie, and Andrew Boake

Espresso Research Group, Department of Computer Science, University of Pretoria
{mtheunis,dkourie}@cs.up.ac.za, Andrew.Boake@absa.co.za

Abstract. The observation that the Open Source Software development style is becoming part of corporate software development, raises questions about its compatibility with traditional development processes. Particular compatibility questions arise where the existing corporate development style is in the agile tradition. These questions are identified and discussed. Measures that can be taken to avoid clashes are suggested. An example illustrates how Corporate-, Agile- and Open Source Software could intersect, and SPEM modelling is employed to show how corporate processes would need to adapt to accommodate the new scenario.

Keywords: Open source software development, Agile software development, Corporate software development, Compatibility.

1 Introduction

"Double, double, toile and trouble; Fire burne, and Cauldron bubble."
—Macbeth (Act IV, Scene 1)

A dispassionate consideration of the cauldron of forces at play in corporate[1] software development, may well raise the question of whether two contemporary (and apparently orthogonal) approaches belong in the brew: Agile Software Development (ASD) and Open Source Software Development (OSSD). Over the past decade, the impact that each of these paradigms has had on the software development industry has grown, and there are signs that this trend will continue. It therefore seems relevant to consider the extent to which these paradigms are mutually exclusive, and, conversely, whether synergies between them can be found. Could blending them into corporate software development processes produce an elixir of life[2] or will they combine into a poisonous witches' brew?

[1] The term *corporate* is used to reference a medium to large enterprise that has its own in-house software developers.

[2] Using this metaphor does not imply that we believe that a silver bullet might be at hand—to reach for another well-used metaphor in software development contexts. However, sobriety does not negate the validity of aspiring to an optimal approach in software development endeavours.

Although not universally practiced, ASD is already widely represented in many industries and, to this extent, is already in the cauldron. Open Source Software (OSS), too, plays a significant role in many corporate contexts. However, this role varies widely: from simple usage of OSS (as in the provision of internet infrastructure components and development tools); to attempts by corporates to leverage the energy inherent in OSSD by managing such projects (the Eclipse project of IBM being a prominent example). There are many variants between these extremes, such as making casual contributions to OSS as a byproduct of using it. An example of just such a scenario will be presented below. Some corporates have even adopted an in-house OSS development style (HP's POS approach being a case in point [1]).

The present discussion excludes scenarios such as the latter, namely where the corporate fully controls the development style. It also excludes simple, uninvolved OSS usage. The corporate OSSD activity that remains and that is the subject of the present discussion (i.e. development which is only partially controlled by corporates), though fairly limited, nevertheless has been growing quite significantly, and could become a disruptive force in the development processes into which it is supposed to blend.

For the purpose of this discussion the extent of corporate participation and the OSS project's scope and size will not be considered. Instead a more generic view of the forces at work is taken. Scope and size influences are therefore a matter for further study.

It should be noted at the outset that OSSD is not a formally defined development methodology. Although every OSS community uses its own process, there is nevertheless a common overall philosophy. For the purposes of this paper OSSD is taken to refer to development that is more or less in line with common principles that have emerged from prominent OSS projects.

Section 2 briefly surveys the nature of corporates, difficulties they are likely to encounter when attempting to engage in OSSD, and managerial adjustments that will be needed. This discussion does not specifically refer to ASD. Instead, the connection between ASD and OSSD is left to Section 3. Here it is pointed out that the OSSD and ASD paradigms apparently embody contradictory attributes, and that consequently, any attempt by a corporate to simultaneously engage in both would appear to be particularly fraught with difficulties. The section also points to ways in which the identified tensions could be managed. An example to illustrate further the kinds of difficulties that might be anticipated is given in Section 4 and SPEM is used as a medium for illustrating ways in which processes might be adapted to alleviate difficulties. Section 5 proposes general ways of ameliorating difficulties that have been identidifed, before concluding in Section 6.

2 OSSD and Corporate Culture

OSS refers to software developed by a movement that values a distributed, open, collaborative development model, as well as the free distribution and modification of its software. As mentioned before OSS already plays a significant role in many corporate software development contexts and as such have been scrutinised by many. Our aim is only to highlight the elements relevant to the discussion at hand.

The corporate environment conventionally places certain requirements on the software development process to enforce employee accountability. This imposes a number of stresses on a development team in such an environment—stresses that will inevitably be accentuated when attempting to engage in OSSD. The following items illustrate some of the clashes that could occur between common corporate culture and that of people typically engaged in OSSD:

– *Monitoring of developers*
  In an environment where remuneration for work is the norm, there is a need to manage and monitor employees, and this is generally taken for granted by regular corporate employees. However, participants in traditional OSS projects are not subjected to such regulation, due to the voluntary nature of the development. In the corporate paradigm, once a manager has assigned a task to a subordinate, it is normally assumed that the manager will track the subordinate's progress and activities, and respond appropriately. This scenario can become complicated when an OSSD style is used internally. It is difficult to monitor what developers are contributing to different and disjoint OSS efforts. Furthermore, it may be difficult for management to assess the importance or relevance of an OSS contribution that is not directly used by the organisation.
– *Fixed time schedules*
  Traditional OSS projects live by the principle of "release often", but these releases are largely *ad hoc*, occurring whenever the core maintainers feel that it is time to do a release. Within the corporate environment there is a need to link different software development projects to fixed time frames so as to support business-driven goals such as taking advantage of market windows and managing financial aspects such as Return-On-Investment (ROI), and IT-driven goals such as coordinated roll-out of related projects.
– *Quality Assurance Processes*
  OSSD, by its very nature, encourages extensive peer review. One of the underlying notions in OSS is expressed in the aphorism known as Linus' Law: "Given enough eyeballs, all bugs are shallow." [2]. However, although some OSS projects may apply certain rules prior to accepting contributions (patches), there are generally no formal OSS code review processes (in particular between the core members). In contrast, in both the agile paradigm, and in many other traditional software engineering approaches, code review procedures are adhered to more diligently.

It is incumbent on corporate management to take cognisance of the contradictions between these styles of producing software, and to manage them as and when needed. These difficulties notwithstanding, creative management solutions should be sought where these conflicts arise most prominently. Sections 4 and 5 illustrate these matters in a specific example. In general, it remains the responsibility of a manager has to ensure that a developer completes essential tasks in due time and in compliance with the requisite quality. Work on random OSS-associated tasks that might be regarded as interesting or fun should be relegated to secondary status, if tolerated at all within the work hours.

There are other aspects to the OSS paradigm that corporate software developers need to understand. One of these aspects is the legal standing of software development, specifically in relation to open-source licenses. Corporate developers generally know about *proprietary* licensing, but they might not be prepared for the variety of OSS licenses and their inter-relationships. OSS developers need to be able to distinguish between the different licenses and their compatibilities. For example, consider the implications if one wanted to link in modules from a library that was issued under the GNU General Public License version 2 (GPLv2), while one's own code was distributed under the Berkeley Software Distribution (BSD) license? An OSS developer would need to know that the BSD and GPLv2 licenses are compatible, *but only if* the BSD code is distributed under the revised[3] BSD version [3,4].

Furthermore, managers of software developers would need to realise that if they have both closed-source and open-source projects in their portfolio, then they should isolate the developers from one another to ensure that no 'contamination' of code takes place.

3 OSSD and ASD

ASD values individuals and their interactions, working software, customer collaboration and responding to change. The primary drivers for ASD are speed and flexibility. Born out of a desire to reduce the overhead caused by over ceremony of traditional software development, the principles of ASD have gained increasing acceptance by corporate developers, under pressure to produce quality software at a rapid pace. As in the case of OSSD, ASD too is an overall philosophy with many variants, each of which finds its application in different development teams.

3.1 OSSD Is Not ASD

Superficially, ASD and OSSD have many aspects in common, including the early delivery of useful software, the valuing of feedback, basing scope and design primarily on utility, and an informed and productive developer community. Indeed, It has been alleged that OSS as a style is just another instance of ASD [5]. To assess the validity of this allegation, we have investigated the extent to which the generic development model of OSS as set out in literature (for example, [2,6]) complies with the principles set out in the *Agile Manifesto* [7]. Furthermore we have compared the stereotypical OSS style and Extreme Programming (XP). Both of these investigations are reported in [8]. Another study by Warsta and

[3] Without the advertisement clause.

Abrahamsson [9] further highlights the differences and similarities between OSS and ASD. Our findings are summarised below.

ASD was born within the corporate world and consequently has a strong focus on certain elements that are not associated with traditional OSSD. These include: *co-located teams*; *assigned* team membership; and *remunerated* employment which brings along a concomitant obligations and hierarchical relationships. Furthermore the *client* role tends to be played by a *non-programmer* who may be the business-user. In contrast, OSSD traditionally starts off with a single developer who is simultaneously the 'client', in that the software to be developed is intended to address a personal need (which could be work related). As the project grows other developers around the world may contribute or even eventually take over the project.

In further assessing the two paradigms, we noted that "the discussion was mainly (but not exclusively) in reference to the stereotypical traditional OSS development style. In reality, the culture surrounding OSS development is neither monolithic nor static". The same viewpoint was taken in regard to ASD. The compliance investigation showed that—at least to some extent—OSSD is indeed compatible with the following list of principles taken verbatim from the Agile Manifesto:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Working software is the primary measure of progress.
- Continuous attention to technical excellence and good design enhances agility.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Build projects around motivated individuals ... and trust them to get the job done.

However, the remaining Agile Manifesto principles are not at the core of the stereotypical OSSD approach. These are:

- Business people and developers must work together daily throughout the project.
- ...Give them the environment and support they need...
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.
- Simplicity—the art of maximising the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organising teams.

The investigation in [8] therefore concluded that there are distinct differences between the two approaches. As a result, the scope for synergy between them is constrained, due to tensions that are likely to arise between teams who follow these opposing principles.

3.2 Adapting ASD

Consider a typical scenario where individual co-located agile development teams collaborate either with other such teams scattered around the world in an OSSD style, or with external OSS projects. Figure 1 provides an example of two Agile teams contributing to an OSS project. An alternative scenario would be where one of the agile teams –as a entity– form part of the core team of the OSS project. This subsection highlights some of the potential adjustments that might be needed.

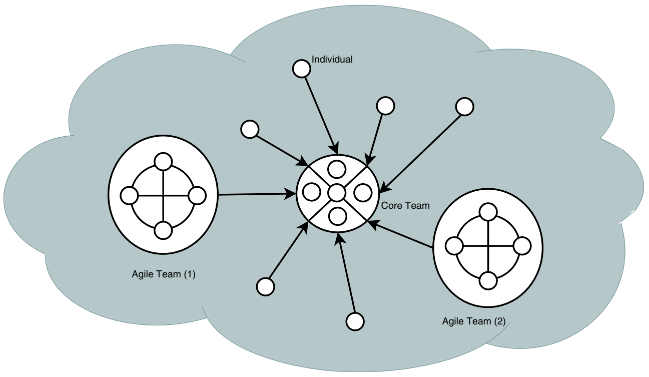


Fig. 1. Agile Teams participating in an OSS project

– *Adapting to remote communication*
From the agile perspective, accommodating a different way of communicating between developers might perhaps be the most challenging. As stated before, the stereotypical agile approach depends extensively on co-located teams who rely on face-to-face verbal communication between members and the availability of on-site customers. This is typically not the case within an OSS development team. Furthermore, the daily routine of an agile team is usually rigorously controlled. Typically, an agile team will start the day off with a short stand-up meeting, followed by a three to four hour focussed session of uninterrupted development. They may then break for lunch, followed by yet another focussed session. During these focussed sessions the developers are typically prohibited from using telephones, e-mail, IRC or any form of external communication, both inbound and outbound. In contrast the OSSD style requires almost *constant* access to communication media such as e-mail,

IRC and the Web. These media, which facilitate 24/7 flow of information will seem extremely 'noisy'[4] to developers accustomed to the agile style.

An added problem is the need to translate and transmit the verbal communication between co-located developers to other distributed external developers. This need to continually document and electronically broadcast the typically informal verbal communication between agile developers may prove to be a severe obstacle in the quest for synergy between ASD and OSSD.

– *Relinquishing of control*
  Agile developers are accustomed to having a large say in the decision making processes that control the direction of a project and the development style and culture within the project. However, when the team is simply yet another contributor in a larger community of developers, some of this control (possibly over *many* aspects of the project) may be lost. This could be a disturbing prospect for ASD developers and should be taken into account when the team interacts with the larger OSS community.

  The view on time-schedules is related to this control issue. Agile proponents advocate fixed, (though short) time cycles to illustrate their progress to the client and to verify the appropriateness of the evolving system. Although the OSS culture is also to deliver frequently, the inclination is to only deliver when the deliverables are useful and stable. This difference perhaps has its roots in the different drivers present in the two approaches. The primary driver in ASD can be seen to be frequent business deadline-driven releases. On the other hand, the primary driver in OSS is delivery of quality software to the community of developer/users.
– *Good OSS community citizenship*
  Agile developers need to realise that they are no longer the centre point of the development effort, but part of a larger community of developers with a deeply rooted culture—largely based on the Unix culture [10]—that has been around for a number of decades. Agile developers will therefore have to gain an understanding of the OSS culture to ensure that they adhere to the underlying, sometimes unwritten, rules of participating in the OSS community.

These points indicate that agile subcultures within parts of the corporate structure would need to adapt if OSSD is introduced into that structure. Indeed, it would seem necessary to compromise on some fundamental agile principles. Accepting these compromises may be a significant test of the very claim made by agile community, namely of being pre-eminently open to change and adaptation.

Of course, the extent to which the above comes into play in a specific project is dependent on the level of engagement between the agile team and the OSS project[5]. An example would be where the development team is responsible for

[4] Although ASD sessions may often be 'noisy' in sense of frequent discussion, the discussions tend to be focussed on immediate tasks at hand. OSSD interruptions, on the other hand, tend to be disparate and less focussed.

[5] These levels of engagement could be: simply *using* an OSS product; *modifying without sharing* the OSS product; *actively contributing* to an OSS project; or *managing* an OSS project.

developing an intranet application that *uses* the *Tomcat* server and *contributes* to the *MyFaces* library. In the aforementioned example one would expect that the tension-points arising in regard to the *Tomcat* project will be somewhat different to those experienced in regard to the *MyFaces* project.

The foregoing, largely a further elaboration of ideas first mooted in [8], does not purported to be an exhaustive list of possible adaptations that need to be addressed in order to gain synergy between agile on the one hand and OSS on the other. It is merely the starting point for a deeper analysis of the contention points and ways of reconciling them.

4 An Illustrative Example

The preceding sections have highlighted the need to take the tension-points into consideration when defining a process for a project that intends to combine OSSD and ASD. Clearly, many problems can be solved by amending the development process(es). However, the ability to adjust the development process depends on the control that one has over the project. Furthermore, the adjustments to the process will be based on the perspective of the team under consideration.

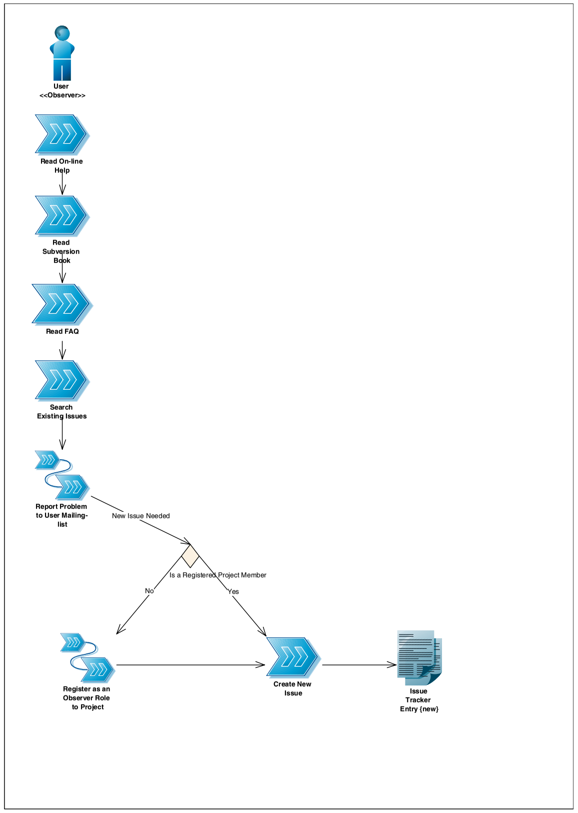
To illustrate the point, consider the following fictitious example: Team Bravo is assigned to develop a conference-room booking system, to be deployed on the intranet of the team's organisation. For the development, team members have decided to use *Tomcat*, *MyFaces*, *Hibernate*, *MySQL* and *GNU Linux*. In addition, they use *Eclipse* and *Subclipse* as development tools. During the course of the project the team extends *MyFaces* with additional components and submits these to the *MyFaces* project. Additionally, a bug is discovered within *Subclipse* and a bug report is filed with the *Subclipse* project, this report containing a *JUnit* test-case to illustrate the problem. Later on, the *Subclipse*-bug is classified as impeding the project and a patch to correct the problem is developed and submitted.

The *Subclipse* project has a predefined process for submitting bug reports. In addition, good project management requires that Team Bravo, too, should follow some internally defined sub-process in submitting a bug report. Clearly, the latter sub-process needs to interface to the *Subclipse* one. A similar situation would hold in the contribution of additional components to the *MyFaces* project.

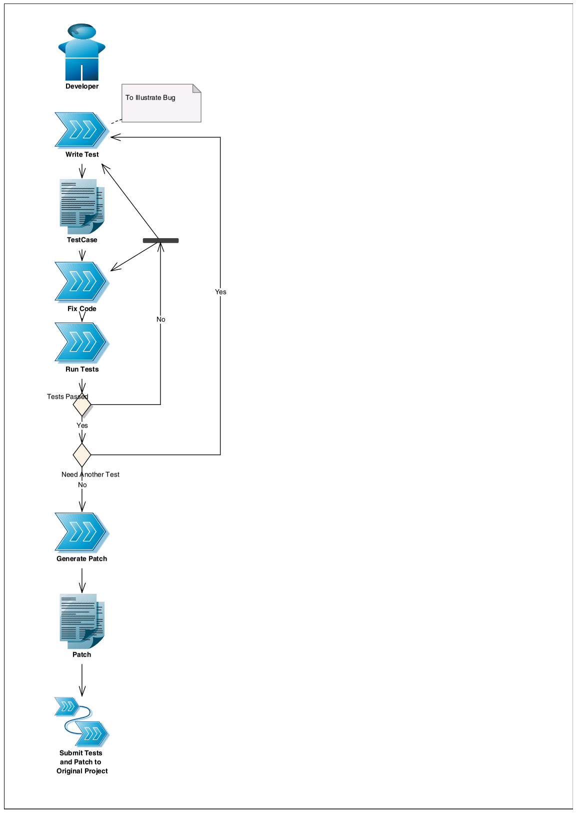
Figure 2(a) depicts the process specified by the *Subclipse* project-page [11] for submitting an *issue* into their issue tracker. The notation used is the *Software Process Engineering Metamodel* (SPEM) version 1.1 [12]. The aforementioned figure represents an Activity diagram to describe the specific *Work Definition*[6]. The *Activities*[7]: *Read On-line Help*, *Read Subversion Book* and *Read FAQ*

[6] "Work Definition: A Model Element of a process describing the execution, the operations performed, and the transformations enacted on the Work Products by the roles. Activity, Iteration, Phase, and Lifecycle are kinds of work definition" [12].

[7] "A Work Definition describing what a Process Role performs. Activities are the main element of work" [12].


User
<<Observer>>
Read On-line Help
Read Subversion Book
Read FAQ
Search Existing Issues
Report Problem to User Mailing-list
New Issue Needed
Is a Registered Project Member
No
Yes
Register as an Observer Role to Project
Create New Issue
Issue Tracker Entry {new}

(a) Workdefinition for Creating a New Issue Tracker Entry in Subclipse



(b) Workdefinition for Resolving Bug in 3rd party library

Fig. 2. Illustrative Workdefinitions

require that a user should first refer to the existing documentation for possible descriptions on how to resolve the problem that they are experiencing. If these activities are deemed unsuccessful, then the user should search the existing issues in the Issue Tracker database. If this, in turn, is also unsuccessful, then the user should report the problem to the *Users mailing-list*.

The *Subclipse* project requires the aforementioned activities as a filtering mechanism to reduce unnecessary entries in the issue-tracker. If need be, the user mailing-list will direct the user to file an entry in the issue-tracker. Embedded in the figure is another work definition: *Register as an Observer Role to Project*. This refers to the additional activities that are required when the user is not yet registered with the project. In the same way *Report Problem to User Mailing-list* encompasses the process of interacting on the user mailing-list.

The internal process that Team Bravo has to follow to resolve the bug is illustrated in Figure 2(b). The figure indicates that Team Bravo uses a test-driven approach to write the bug-fix, as required by their ASD-compliant development process. *Submit Patch to Original Project* represents an abstract sub-process to follow when reporting the problem and providing a solution to a 3rd party's project. In the above example the abstract sub-process should be superseded by the *Subclipse: Creating a New Issue Tracker Entry* process specified in Figure 2(a) and described above. In this way, the general project process can be customised to incorporate interfaces to other projects.

The foregoing endorses the notion of defining a *process per project*, as advocated by a number of methodologists, including Cockburn [13]. In the case of the conference-room booking system project, not only did the overall process depend on the project's internal sub-processes, but it also had to take account of the sub-processes of other external projects. In practice, the set of external projects to be incorporated may vary from one project to the next. This is evidently a typical consequence of incorporating an OSSD approach into a corporate development effort.

5 Proposals

The previous section gave a practical example of the kind of inter-project interaction scenario which a corporate development team incorporating OSSD could face. Numerous additional illustrative examples and scenarios could no doubt be cited. However, the present limited example already introduces a number of ideas that lead to concrete proposals for dealing with these tensions.

1. Note that, in general, there will be interaction points between sub-processes of the internal project and sub-processes of the various external (OSS) projects. The tensions previously mentioned, i.e. tensions between OSSD and ASD and/or traditional corporate culture, are most likely to be encountered at these interaction points. It would seem, therefore, that one can at least start to deal with these tensions, by articulating—either formally or informally—a sub-process at each such interaction point.

Taking this approach, one is able to specify the corporate development process with minimal (but non-zero) concern for the external projects process. The external projects processes are then only plug-ins that are realised in one's own process on an "as-needed" basis. This contains the tensions between the different, possibly opposing processes and encapsulates them at the defined interface points.

2. Another possible practice to consider, is the introduction of an additional role to the development process: that of a *liaison officer*. This role should be adopted whenever one either uses or contributes to an OSS project. In essence, someone would be designated as responsible for acting as a communication conduit between the projects. The responsibility of this role would be to gain and maintain an appropriate level of understanding of an external project with regard to the respective processes and the evolution of the artifacts. This knowledge would then be disseminated to the rest of the team, as required by the given project.

   An example of this would be assigning developer Jane to liaise on project Subclipse on the team's behalf. Whenever a new release is available, she will inform the rest of the team and aid with the adoption thereof by the team. This role might be seen as a substitute for the product representative from commercial companies from which software product are acquired.

   Furthermore, when deemed necessary, the role may be assigned to multiple persons, for example, on a per module basis for large external projects. The need to assign this responsibility to multiple persons may be particularly important in a context where an ASD approach such as Extreme Programming is being used, since the latter places emphasis on maintaining a level of human redundancy.

   The role of *liaison officer* would vary for each given project and for each external project, its importance being determined by factors such as those listed in the previous section.

6 Conclusion and Future Work

The implications of introducing OSSD into a corporate environment have been considered, and particular references has been made to the implications of accommodating both OSSD and ASD. An example illustrated the kind of situation facing corporate software developers who attempt to develop in an ASD style while collaborating with distributed partners.

In the example, the focus of the development team was to *use* OSS products. However, had the ASD team formed all or part of the *core* of an OSS project, then a number of other issues would need to be addressed. These have not been considered here, and are left for future study. However, it is clear that the *process-per-project* notion will feature strongly in any such consideration.

Clearly, when mixing in different ingredients from the software development processes and/or practices on offer, it would be wise to be weary about the resulting brew. It could turn out to poison or paralyse the project. On the other hand, it might be a elixir—an enabler of a successful and enduring project.

References

1. Dinkelacker, J., Garg, P.K., Miller, R., Nelson, D.: Progressive open source. Technical Report HPL-2001-233, Hewlette-Packard Laboratories, Palo Alto (2001)
2. Raymond, E.S.: The cathedral and the bazaar. First Monday (1998) (Accessed: 2007/06/27), http://www.firstmonday.org/issues/issue3_3/raymond/index.html
3. Rosen, L.: Open Source Licensing: Software Freedom and Intellectual Property Law. Prentice Hall Professional Technical Reference (2005)
4. Free Software Foundation: Frequently asked questions about the GNU GPL (2006) (Accessed: 2007/06/27), http://www.gnu.org/licenses/gpl-faq.html
5. Raymond, E.S.: Discovering the obvious: Hacking and refactoring. Weblog entry (2003) (Accessed: 2007/06/27), http://www.artima.com/weblogs/viewpost.jsp?thread=5342
6. Feller, J., Fitzgerald, B.: Understanding Open Source Software Development. Addison-Wesley, Reading (2002)
7. Cunningham, W.: Manifesto for Agile Software Development (2001)(Accessed: 2007/06/27), http://www.agilemanifesto.org
8. Theunissen, W., Boake, A., Kourie, D.: Open source and agile software development in corporates: A contradiction or an opportunity? Jacquard Conference, Zeist, Holland (2005)
9. Warsta, J., Abrahamsson, P.: Is open source software development essentially an agile method? In: Proceedings of the 3rd Workshop on Open Source Software Engineering, 25th International Conference on Software Engineering, Portland, Oregon, USA, pp. 143–147 (2003)
10. Raymond, E.S.: The Art of Unix Programming. Addison-Wesley, Reading (2003)
11. Subclipse Project Team: Subclipse issue tracker (2006) (Accessed: 2007/06/27), http://subclipse.tigris.org/project_issues.html
12. Object Management Group: Software process engineering metamodel specification. formal 05-01-06, Object Management Group (2005)
13. Cockburn, A.: Agile Software Development. Pearson Education, Inc., London (2002)

Capable Leader and Skilled and Motivated Team Practices to Introduce eXtreme Programming

Lech Madeyski[1] and Wojciech Biela[2]

[1] Institute of Applied Informatics, Wroclaw University of Technology, Wybrzeze Wyspianskiego 27, 50370 Wroclaw, Poland
Lech.Madeyski@pwr.wroc.pl
http://madeyski.e-informatyka.pl/

[2] ExOrigo Sp. z o.o., Krucza 50, 00025 Warsaw, Poland
Wojciech.Biela@exorigo.pl
http://www.biela.pl

Abstract. Applying changes to software engineering processes in organisations usually raises many problems of varying nature. Basing on a real-world 2-year project and a simultaneous process change initiative in Poland the authors studied those problems, their context, and the outcome. The reflection was a need for a set of principles and practices to help introduce eXtreme Programming (XP). In the paper the authors extend their preliminary set, consisting of the Empirical Evidence principle, exemplified using DICE®, and the practice of the Joint Engagement of management and the developers. This preliminary collection is being supplemented with the Capable Leader, as well as the Skilled and Motivated Team practices based on the DICE® framework as well.

Keywords: Extreme programming, Agile adoption, Process change, Software process improvement, DICE® framework.

1 Background

The paper is based on a real-world software project and the attempt to bring agile practices to that project and organisation. The organisation is a medium-sized company that operates in several distant locations in Poland. The study relates to a 2-year project developed by 3 programmers (the whole team consisted of 8 programmers). The goal of that project was a B2C web platform for a trust fund agent.

One of the authors joined the team, after a year from the project start, to resolve various issues that arose in the development environment. At that time, his only knowledge and experience concerning agile practices came from the e-Informatyka project led by the co-author, their long discussions and of-the-books knowledge. The problems were addressed using a collection of agile techniques.

Test-Driven Development (TDD) was something new and was a complete success. The recognised overhead in writing tests was compensated by the decrease in the bug rate (which is in line with [1]) and the ability to refactor the system quickly. It was not without resistance and a lot of coaching was necessary, but eventually developers found TDD to be very effective and rewarding.

Refactoring was used in the past, but not that explicitly and often. Without doing regular refactorings (while having unit tests in place), many change requests from the client would have met very strong resistance from the team and would have caused a lot more pain.

Pair Programming (PP) was introduced at some point and it really helped the team to share knowledge and halve the time of introducing a new person to the project. Developers realized that PP also helped produce better code in terms of design quality and the number of bugs discovered later. However, not all of the empirical studies [2,3,4,5,6,7,8] support the positive impact of PP on software quality, as was observed by the team. Substantial problems arose when the consequences of pairing - higher costs - were exposed to the client. On the basis of several empirical studies [9,10,11,12,13,6], one may conclude that pair programming effort overhead is probably somewhere between 7% and 84%, whilst the team's observation was close to a 50% overhead.

In-process design sessions required a lot of coaching and basic programming recommendations [14] were introduced. More advanced principles like Separation of Concerns and Dependency Injection were introduced as well. There was some resistance in this matter as with TDD. But the return on investment in this case was usually very high, so discussions on this were swift.

Problem Decomposition was, alongside TDD, the most successful technique brought to the team. Divide the problems into pieces that you can grab, solve the problems (estimate, plan, design or code, whatever the case), and get back to the whole. Whether used at the release planning level or at the implementation detail level, it performed brilliantly.

Continuous Integration and task automation was an obvious benefit, the team saved many hours of dull deployment work. Also many man-made mistakes were omitted due to task automation.

Darts were something completely new, but this great incentive ultimately glued the team together. People started talking to each other. They suddenly had another motivation to complete their tasks (they could have a game). Additionally it provided yet another reason, other than the biological one mentioned by Beck [15], to take your eyes off the computer screen, get up, and clear your head. Eventually, the management accepted it (and had a game themselves). In the authors' opinion, such group toys are a must have for any development team.

Communication was and still is an issue because the team is remote. A wiki was set up, which helped a lot. Direct communication with the customer was encouraged. A conference area was set up, with Skype installed. Much of the outdated documentation was disregarded, instead user stories were recorded for further discussions. This also met resistance, as the customer was in the habit of doing things the traditional way.

The preliminary results, emphasising the need for a concrete set of principles and practices that would complement the main body of eXtreme Programming (XP) and support the fragile process of introducing XP practices, have been presented and discussed within the agile community [16]. These results consisted of an agile principle (*Empirical Evidence*) and practice (*Joint Engagement*) proposal to aid the process change. The *Empirical Evidence* principle recommends to ground on empirical evidence when introducing changes. One of the widely accepted sources of empirical evidence concerning introducing changes is the DICE® framework [17], created by The Boston Consulting Group. However, other sources of empirical evidence are welcome as well. The *Joint Engagement* practice is guided by the *Empirical Evidence*, as well as the *Accepted Responsibility* principle [18]. Following the *Joint Engagement* practice we begin the change process at various structural levels of an organisation [16]. This preliminary set of principles and practices will be further extended in the next section.

2 Keep the DICE® Rolling

The DICE® framework is a simple empirical evidence-based formula, based on 225 change initiatives study, for calculating how well an organisation is or will be implementing its change initiatives [17]. The DICE® framework comprises a set of simple questions that help score projects on each of the five factors: project duration (D), team's integrity (I), commitment of managers (C1) as well as the team (C2), and additional effort (E) required by the change process. Each factor is on a scale from 1 to 4. The lower the score, the better. Thus, a score of 1 suggests that the factor is highly likely to contribute to the program's success, and a score of 4 means that it is highly unlikely to contribute to the success [17]. In DICE®, a project with an overall score between 7 and 14 is considered a *Win*, between 14 and 17 is a *Worry* and between 17 and 28 is a *Woe*. The DICE® formula is $D+2*I+2*C1+C2+E$.

The authors used the DICE® and its C1 and C2 factors previously when proposing the *Empirical Evidence* principle and *Joint Engagement* practice duo [16]. The project team's performance integrity factor (I) concerns the ability to complete the process change initiative on time and depends on the team members' skills. According to Sirkin et al. [17]:

> If the project team is led by a highly capable leader who is respected by peers, if the members have the skills and motivation to complete the project in the stipulated time frame, and if the company has assigned at least 50% of the team members' time to the project, you can give the project 1 point. If the team is lacking on all those dimensions, you should award the project 4 points. If the team's capabilities are somewhere in between, assign the project 2 or 3 points.

Building upon the *Empirical Evidence* principle and the DICE® framework the authors recognise that the project team's performance integrity factor (I) may lead to two new practices: the *Capable Leader* and the *Skilled and Motivated Team* practices. Following those practices one improves their DICE® integrity (I) factor and thus increases the likelihood of success. These practices are clearly in line with the *Improvement* principle [6] from XP. Furthermore, other XP principles like *Diversity*, *Flow*, *Quality*, *Accepted Responsibility* [18] are closely related to the ideas behind those practices.

2.1 Capable Leader

This practice proposal turns the attention to the role of the team leader. It may concern both the project's development team leader responsible for implementing the development practices in a project, as well as the leader for the organisational change process which happens throughout the organisation willing to adopt agile practices.

In the adoption of agile practices the role of the team leader is very significant. New ideas and enthusiasm often die on its own. Things get back to where they were before and the chance for a process change is inadvertently lost, as the same organisation will probably not want to try it twice. There needs to be a person who will advocate for the change, explain and remove obstacles. The project team should be led by a highly capable and motivated leader who is respected by his peers. The team leader should serve the team, never the opposite. On a daily basis he is not a manager, but like the Scrum Master in Scrum [20], rather a normal team member, who has to put on different hats according to the current needs of the team. One time he needs a coach's hat, afterwards he takes out a developer's hat, next minute a manager's hat, then again a team catalyst's hat or maybe an agile evangelist's hat. His ultimate role is to help the team improve, not force them to improve, but rather enable them to do it on a daily basis. But be aware that the team leader cannot make the team dependent on his person, among other problems that would simply cause the Truck Factor to go down dramatically. Contrary, if the team lead should leave the team for a week, nothing dramatic should happen.

Both the team and the team leader need specific resources to be assigned for the change process. For example if they do not have enough time and opportunities to roll out the changes then no amount of wisdom nor tooling will help. Other than that there is no fixed set of resources they need, the whole team together has to identify their requirements.

2.2 Skilled and Motivated Team

This practice in turn focuses on the need to grow a team of motivated professionals and care for them. The major problem is the difference between a team and a work-group and how to help a work-group become a team. Again, this may concern the team of developers implementing a product or more broadly the entire team of the individuals responsible for the change process.

It is quite obvious that skilled professionals are more effective than the weaker ones, but the point is to explicitly aim for having the best people on board. One should not stop there, because very talented individuals often don't work well together, this is why this practice focuses on teams rather than individuals. The people forming such a team need to be motivated to effectively work towards common goals. That is what differentiates them from a work-group of clever masterminds working alone in the same room (often barely talking to each other). Agile processes adoption is no exception from that rule. It is good to form the team in such a way that it has a critical mass of agile believers and practitioners. Then one needs to help them work together making heavy usage of team retrospectives [19]. Act, inspect and adapt. If it is not possible to change the crew, then coach them extensively, and make even more use of retrospectives.

Following the proposed practices helps to assure the highest capability, skills and motivation of both the team leader and the team members. It means that we do our best to have the project team led by a competent and motivated individual who is respected by his peers. Moreover, it is best when the team members have the skills and motivation to complete the project in the accepted time frame [17]. To achieve these aims the organisation has to assign a reasonable part of the team members' time to the change process. Among other activities that would have an impact on the team's integrity factor are e.g. assuring an appropriate degree of financial support, emphasising the understanding of the potential contribution of the change process to the situation of the team, or a particular team member. Individuals at the management and at the developer level should be educated and involved in the process (*Joint Engagement* practice [16]). They have to willingly accept their diverse responsibilities in the change process (*Accepted Responsibility* principle [18]). The current form of the proposed collection of principles and practices to introduce XP, against a background of the main body of XP, is presented in the Figure 1.

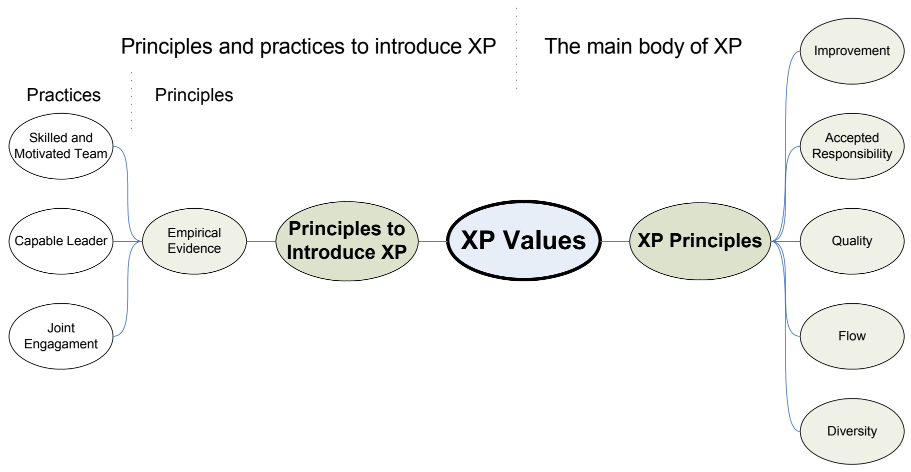


Fig. 1. The set of principles and practices to introduce XP against a background of the main body of XP

3 Conclusions

Introducing agile practices is a challenge for every organisation. In the search for methods that would ease the adoption process, the authors began to identify principles and practices to introduce XP [16]. In the paper they identified further two potential practices based on the Empirical Evidence principle: the *Capable Leader* and the *Skilled and Motivated Team* practices. Such practices might not be very surprising for some. However, as the history of XP shows certainly there is value in labelling and arranging well-known behaviours into such concrete forms of *values*, *principles* and *practices* specifically aimed at solving a concrete problem. Those two practices work best when applied together, because of a synergy effect, as in the case of many other XP practices. The team leader can be more effective with a motivated team, while the team lead by a competent and smart individual will also do much better. This close relation is also emphasised by the integrity (I) factor of the DICE® framework.

Other agile practitioners are highly encouraged to contribute to this presented set of principles and practices to cover more and more of this unstable ground, as well as to empirically evaluate the ideas in different contexts.

Acknowledgements

The authors expresses their gratitude to the agile community for their feedback concerning the proposed collection of principles and practices. This work has been financially supported by the Ministry of Science and Higher Education, as a research grant 3 T11C 061 30 (years 2006-2007).

References

1. Williams, L., Maximilien, E.M., Vouk, M.: Test-Driven Development as a Defect-Reduction Practice. In: ISSRE 2003: Proceedings of the 14th International Symposium on Software Reliability Engineering, Washington, DC, USA, pp. 34–48. IEEE Computer Society, Los Alamitos (2003)
2. Müller, M.M.: Are Reviews an Alternative to Pair Programming? Empirical Software Engineering 9(4), 335–351 (2004)
3. Madeyski, L.: Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality. In: Zieliński, K., Szmuc, T. (eds.) Software Engineering: Evolution and Emerging Technologies. Frontiers in Artificial Intelligence and Applications, vol. 130, pp. 113–123. IOS Press, Amsterdam (2005)
4. Madeyski, L.: The Impact of Pair Programming and Test-Driven Development on Package Dependencies in Object-Oriented Design - An Experiment. In: Münch, J., Vierimaa, M. (eds.) PROFES 2006. LNCS, vol. 4034, pp. 278–289. Springer, Heidelberg (2006)
5. Hulkko, H., Abrahamsson, P.: A Multiple Case Study on the Impact of Pair Programming on Product Quality. In: ICSE 2005: Proceedings of the 27th International Conference on Software Engineering, pp. 495–504. ACM Press, New York (2005)

6. Arisholm, E., Gallis, H., Dybå, T., Sjøberg, D.I.K.: Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. IEEE Transactions on Software Engineering 33(2), 65–86 (2007)
7. Madeyski, L.: On the Effects of Pair Programming on Thoroughness and Fault-Finding Effectiveness of Unit Tests. In: Münch, J., Abrahamsson, P. (eds.) PROFES 2007. LNCS, vol. 4589, pp. 207–221. Springer, Heidelberg (2007)
8. Madeyski, L.: Impact of pair programming on thoroughness and fault detection effectiveness of unit test suites. Software Process: Improvement and Practice (accepted), DOI: 10.1002/spip.382
9. Nosek, J.T.: The Case for Collaborative Programming. Communications of the ACM 41(3), 105–108 (1998)
10. Williams, L., Kessler, R.R., Cunningham, W., Jeffries, R.: Strengthening the Case for Pair Programming. IEEE Software 17(4), 19–25 (2000)
11. Nawrocki, J.R., Wojciechowski, A.: Experimental Evaluation of Pair Programming. In: ESCOM 2001: European Software Control and Metrics, London, pp. 269–276 (2001)
12. Müller, M.M.: Two controlled experiments concerning the comparison of pair programming to peer review. Journal of Systems and Software 78(2), 166–179 (2005)
13. Nawrocki, J.R., Jasiński, M., Olek, L., Lange, B.: Pair Programming vs. Side-by-Side Programming. In: Richardson, I., Abrahamsson, P., Messnarz, R. (eds.) EuroSPI 2005. LNCS, vol. 3792, pp. 28–38. Springer, Heidelberg (2005)
14. Bloch, J.: Effective Java: Programming Language Guide. Addison-Wesley, Reading (2001)
15. Beck, K.: Test Driven Development: By Example. Addison-Wesley, Reading (2002)
16. Madeyski, L., Biela, W.: Empirical Evidence Principle and Joint Engagement Practice to Introduce XP. In: Concas, G., Damiani, E., Scotto, M., Succi, G. (eds.) XP 2007. LNCS, vol. 4536, pp. 141–144. Springer, Heidelberg (2007)
17. Sirkin, H.L., Keenan, P., Jackson, A.: The Hard Side of Change Management. Harvard Business Review 83(10), 108–118 (2005)
18. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley, Reading (2004)
19. Derby, E., Larsen, D.: Agile Retrospectives: Making Good Teams Great. Pragmatic Bookshelf (2006)
20. Schwaber, K., Beedle, M.: Agile Software Development with SCRUM. Prentice Hall, Englewood Cliffs (2001)

Platform-Independent Programming of Data-Intensive Applications Using UML*

Grzegorz Falda[1], Piotr Habela[1], Krzysztof Kaczmarski[2], Krzysztof Stencel[3], and Kazimierz Subieta[1]

[1] Polish-Japanese Institute of Information Technology, Warsaw, Poland
[2] Faculty of Mathematics and Information Science, Warsaw University of Technology, Warsaw, Poland
[3] Institute of Informatics Warsaw University, Warsaw, Poland
`{gfalda,habela,stencel,subieta}@pjwstk.edu.pl`,
`K.Kaczmarski@mini.pw.edu.pl`

Abstract. The shift of development effort onto the model level, as postulated by MDA, provides an opportunity for establishing a set of modelling constructs that are more intuitive and homogeneous than its platform-specific counterparts. In the paper UML is confronted with the needs specific for data-intensive applications and propose a seamlessly integrated platform-independent language with powerful querying capability, which would allow specifying a complete application behaviour. The proposal is aimed at high level of compliance with existing modelling standards – as such it is based on UML behavioural elements and on OCL for expressions. The motivation behind this approach is presented, the challenges implied by it are discussed, and the role of the model runtime implementation is indicated.

Keywords: UML, executable modelling, query language, action language, MDA, database applications.

1 Introduction

The approach of model-driven software development and the Model Driven Architecture (MDA) initiative in particular sketch the vision of the next big step in raising the level of abstraction and flexibility of programming tools. While any method that treats modelling activities as central can be considered "model-driven", the key expectation behind MDA is achieving a productivity gain through the automating of software construction based on models. This results in a significant shift of expectations regarding modelling constructs – from being merely a semi-formal mean for outlining and communicating project ideas, to machine-readable specification demanding precise semantics. Thus, MDA creates a spectrum of model applications, which is often described using the following three categories:

* This work is supported by the European Commission 6-th Framework Programme, Project VIDE - VIsualize all moDel drivEn programming, IST-033606-STP.

- *Sketches*, that represent the traditional use of UML and similar language as a help in understanding the problem and communicating ideas and solutions to other developers. Those kinds of models do not need to be complete nor fully formalized.
- *Blueprints*, that follow the traditional distinction between design and its realization – as in case of other engineering domains. The distinction of design and coding is maintained in terms of artifacts and is also reflected in assignment of those tasks to different groups of developers.
- *Executable models*, that require the presence of precise semantics and – by the automation of executable code production – blur the distinction between modelling and programming.

The last case is especially connected with the MDA initiative of the Object Management Group and has motivated significant restructuring and extension of the UML standard as experienced in its version 2 [1]. The most far-reaching variant of this vision is to replace existing programming languages with platform-independent modelling languages in majority of applications [2] (the same way the former once replaced assembly languages). This requires the presence of sophisticated transformation tools encapsulating the knowledge on particular target platform technologies, and depending on mature and widely adopted modelling standards – at least at the Platform Independent Model (PIM) level. In that case the application code produced would not be the subject of direct editing at all. The amount of work at the Platform Specific Model (PSM) level could also be reduced to minimum.

That vision is inherently challenging due to the transformations between heterogeneous high level languages involved (especially – if multi-tiered software and data processing are considered). This is probably why the idea of so highly automated MDA has not been extensively applied to the business applications so far [3]. At the same time, however, applying strict MDA to that area seems especially compelling given the uniformity and reuse it could potentially provide there. This idea underlies the development of our platform-independent language aimed at the data-intense business application area, which is being created as one of the central elements of our project of visual modelling toolset VIDE (*Visualize all model-driven programming*).

In the paper we describe our approach to that problem, which is based on the following postulates:

- UML Structures unit seems to be rich and versatile enough to be considered as a foundation for a data model used in platform-independent development. A number of semantic details needs to be clarified to achieve that aim though.
- To make the model complete, the means of imperative programming need to be available at the PIM level. To raise the intuitiveness and productivity compared to the mainstream platform-specific technologies, the statements and queries should be integrated into a single language in a truly seamless way.
- An execution engine for PIMs is needed as a reference implementation. It is also essential as a modelling tool component serving for platform-neutral model validation.
- Representing an application code in the form of standard metamodel instances and flexibly combining textual and visual notations for the behavioural modelling of introduced constructs can provide a significant advantages over plain, purely textual languages.

The rest of the paper is organized as follows. Section 2 describes the expectations and concerns regarding the executable modelling approach and explains the motivation behind our approach. In Section 3 the UML 2 standard is presented from the point of view of precise specification of data-intensive applications. Section 4 describes the idea of a UML-based programming and query language and summarizes the current results in its development. In Section 5 we outline the role of the language within a broader toolset and development process and indicate further challenges. Section 6 concludes the paper.

2 Motivation

This should not be a surprise that pragmatic approaches to the problem outlined may depend on the programming notions known from existing programming and database languages. Specifying just a structural aspect of the model (using e.g. UML Class diagrams) is not sufficient if a high degree of code generation is the aim. Delegating the details of the behaviour specification to constraint definitions – as explained e.g. in [4] has the quality of the higher level of abstraction. However, realizing behavioural modelling this way in general is problematic from the point of view of complexity of model transformations. Moreover, in case of more complex behaviour this could be highly complicated and hard to accept by developers who are familiar with the traditional, imperative style of specification. Even at the side of imperative programming there is a dilemma regarding the selection of particular modelling notions to be supported in the executable models. What needs to be balanced is the ease of translation into other languages and making the language familiar for the developers knowing mainstream platform-specific languages, against the aim of achieving a higher level of abstraction and hiding the heterogeneity of type systems and programming paradigms.

When speaking about the current modelling standards (especially UML2 [1], MOF2 [5] and OCL2 [6]) and their development towards the vision of executable modelling, it is necessary to mention the critique this vision of model driven development faces – see e.g. [7]. While its motivation of raising the level of abstraction and controlling the level of details is recognised, the overall approach to dealing with complexity is considered problematic. A question of maintainability given the number of model representations is raised. There is even a doubt expressed if the MDA does not just push the complexity into later phases of the development process instead of reducing it (as the round-trip engineering requiring translation of lower-level notions into a higher level of abstraction is problematic). The size of the current UML specification is a concern, Especially, given that some areas essential for business applications are missing or weakly addressed there. This includes for example user interface specification, workflow/ business process definition and data modelling. Moreover, the techniques that could support the stakeholders' involvement into the development process are also found missing from the language and not much visible in MDA in general [8]. The demand for a reference implementation and a human-readable operational semantics is emphasized – also for assuring the proper implementation of UML transformation tools [9].

There are also varying opinions on the role and usefulness of visual programming at the level of detail suggested by UML Actions and Activities units. In [8] an observation is made that most developers prefer text-based solutions for modelling and the focus of many tool vendors on diagram-based solutions is questioned. Fowler [10] and several other practitioners express concerns about visual programming as they indicate the diagrammatic way of code construction is incomparably slower. Indeed, majority of action languages in existence today [11, 12] are purely textual. However, if the difficulties related with visual coding at this level of granularity could be overcome, the visual notation may be advantageous under the following criteria:

- More control of the editing process, giving the possibility to assist the developer and to avoid some coding errors,
- More expressive distinction of different language constructs,
- Ability to more clearly visualize scopes and name visibilities – especially for complex expressions,
- Ability to incorporate domain specific user-defined symbols to make the code easier to follow e.g. during the validation by the domain experts.
- More potential for annotation and substitution texts / symbols use.

Given the above considerations and the implementation and transformation issues explained later, we chose to build the core of our VIDE language (called VIDE-L in the sequel) on the notions known from programming languages (expressed in terms of UML Actions and Structured Activities) rather than starting from flow-oriented activity models, state machines [11] or interactions. While this approach can be considered conservative from the point of view of the modellers' community, we note the following advantages compared to traditional programming languages:

- Depending on executable semantics for UML and the data model it assumes to support its adoption as a canonical model for various modelling and integration efforts.
- Capability of avoiding the "impedance mismatch" existing between database and programming languages in mainstream platform-specific technologies.
- Flexibility of code composition, validation, transformation and annotation gained through its representation in the model repository.
- Ease of switching syntactic options to offer an optimum combination of visual and textual notation for making the coding intuitive and productive for developers who know UML.

3 Standard Base

What makes the UML a natural choice of a standard's base for the intended language is the popularity of the standard and its recent restructuring aimed precisely towards the executable modelling paradigm. Another fact that strengthens the position of this standard is recent development of the modelling tool implementation framework based on the UML 2.x metamodel [13], which may support uniform handling and exchange of UML models among various tools.

However, that selection itself is only a first step on the road for defining a platform-independent language for the area of application assumed. It is necessary to note the following factors:

- At the origin of the UML when it served rather only as an analysis and design language, some degree of ambiguity regarding the semantic details could be even considered desirable, as it leaves more freedom of applying its modelling constructs to varying technologies. The details of e.g. inheritance mechanism, parameter passing or object lifecycle could remain irrelevant on the level of abstractionassumed by those models or could be interpreted locally in terms of the technology of choice. This is not the case for precise PIM development, hence the efforts to provide UML with precise executable semantics specification [14].
- Moreover, the multi-purpose nature of UML implies that not all of its elements are capable of having a precise executable semantics defined. Moreover, from among the concepts having such capability, a subset should be selected to make the resulting language acceptably simple and suitable for its area of application (e.g. taking into account the needs of target platforms).

While it is impossible to provide a complete specification of the VIDE language here, in the rest of this section we try to present the most important decisions on selecting and detailing such a UML subset and describe motivations behind them.

The foundational problem (especially given the purpose of our language) is specifying the data definition language. We start from the complete UML Classes unit and perform the selection to achieve a data model that is expressive and universal but at the same time realistic in terms of its implementation and handling by the language statements. To this end, our motivation is to let the developer get rid of the object-relational mapping complexity. Hence we assume an object model with classes, static generalisation/specialisation supporting for substitutability and disallowing inheritance conflicts in terms of the multi-inheritance. Further work on achieving a greater flexibility of the inheritance hierarchy is aimed at exploiting the notion of dynamic inheritance in UML which we plan to realize in the form of dynamic object roles [15]. However, since it would require extending the behavioural part of the language either, we postpone this to the next version of the language.

The role of the UML Classes unit in VIDE can be summarized as follows. The current selection of UML notions used by the language seems to be the shortest way for achieving the expressive power of a programming language. The selection includes the core notions of UML Classes, Structured Activities and Actions units. Class model provides the structures that establish a context (in terms of features available to the behaviour: attributes, links, operations) under which a given behaviour is specified. It also provides a place for behaviour definition in the form of methods implementing operations of UML classes. Using VIDE for specifying behaviour in other contexts than class operations is being considered for the future (it seems to be feasible to adapt because of the strict UML compliance of VIDE constructs).

A feature that makes the language more distinct from popular OO programming languages is the realization of the Association notion. Compared to its complete definition a number of limitations have been introduced. Particularly, we skip the support for non-binary associations and association-classes. Although useful in conceptual modelling, they are problematic due to the complexity involved in their implementation. The weak adoption of CORBA Relationship Service [16] that supported similar notions seems to support this observation. On the other hand, the language will automate the creation and referential integrity of updates of links that instantiate bi-directional associations. The current specification of UML provides big number of options for relationships among objects described by the Property notion: this includes unidirectional and bidirectional associations, plain attributes (Property not belonging to an Association) and the possibility to describe each property with the *aggregation* attribute distinguishing 3 aggregation kinds[1]. This may be considered redundant. Moreover, what blurs that distinction is allowing the UML notation to use the attribute and association notation virtually interchangeably. Those issues are considered important since our language demands the possibility of expressing nested data structures (like e.g. XML documents) – hence we need to distinguish several options for connecting two complex objects: plain bidirectional association, plain unidirectional association, bidirectional composite association, unidirectional composite association (the latter substitutable with non-primitive attribute).

There are also several considerations related to the differences in data modelling and accessing between programming languages and database environments. In programming languages the class definition does not usually determine the name of variables that will store its instances. On the other hand, this is quite natural for database schemas.

The aims and patterns of encapsulation are also rather different in case of database schema. While the current version supports just the visibility specification for classes' features, we consider future extending of the encapsulation mechanism using the notion of updateable views, which may require more precise declarations at the side of UML [1].

In contrast to programming languages like Java we do not assume the garbage collection of the objects expressed in our language; instead explicit *DestroyObjectAction* of UML is supported.

4 Language Development

While "query language" is listed in the standard specification as one of the OCL possible purposes, the use of the language in VIDE-L is significantly different compared to the purpose OCL was originally designed for. So far, the expressions of OCL have been used mainly for constraint specification (where eventually were evaluated into Boolean values) or e.g. for calculating the initial values of attributes etc. In our case the area of application is much broader, since anywhere a programming construct needs to be extracted (e.g. selecting objects to be updated, removed, linked or passed as a parameter in an operation call), the expressions in OCL are used. This means the result of such expression does not necessarily need to be just an r-value.

VIDE-L language as a whole makes the similar simplifications in dealing with complex / primitive data and reference / value distinctions as e.g. Java. However, it achieves a bit higher expressiveness thanks to introducing dedicated statements for link updating and by supporting two parameter passing modes from among the ones assumed by UML: *in* and *inout*.

[1] Note that the meaning of this attribute (i.e. if the owner of the property plays the "whole" or the "part" role) is unfortunately dependent on whether the property is a member of association or not).

The use of queries as described above leads to a seamlessly integrated language, which is in contrast with embedding queries of a separate language as strings and dealing with resulting heterogeneity of type systems, syntaxes, binding phases etc. which is the issue e.g. in the ODMG standard [17] and Java-based specifications that evolved from it.

Those problems are to a big extent absent in our case, however, to achieve the goal we needed to deal with some overlap and heterogeneity resulted from this rather novel use of OCL and from the fact that UML and OCL specifications have been recently developed separately. Among those it is worth to note:

- Varying style of variable declarations: UML uses multiplicities and the ordering and uniqueness flags. OCL in turn does not support them and depends on the collection type constructors instead.
- Introducing the seamless support for OCL expressions for UML behaviour makes the following actions redundant: ReadStructuralFeatureAction, ReadSelfAction, ReadExtentAction, ReadLinkAction etc.

Apart from the language semantics, also its syntax plays an important role for the productivity and ease of its adoption. It is necessary to note that the decisions on the concrete syntax that UML2 specification leaves open for developers is not necessarily just a plain selection of the list of visual or textual symbols. The elements not having a concrete syntax specified (which refers roughly to Actions and Structured Activities units) are fairly universal and fine-grained. This encourages the designers of particular action languages to consider creation of various higher level language constructs that are useful for the intended area of application and whose mapping onto UML element instances is not necessarily "one-to-one".

Indeed, although we tried to provide the statements that rather directly represent respective UML Actions and Activities primitives, a number of useful programming language constructs required a more complex mapping. Those cases include:

- Reusing generic Structured Activities elements to provide useful statements for loops and conditional instructions. For example, ConditionalNode does not provide dedicated construct for "else" or "otherwise" clauses. On the other hand, we do not take advantage of the ConditionalNode's capability of providing results (i.e. serving as expressions).
- Providing useful shortcuts like the +=, -=, *= and /= assignments is especially useful when considering iterative processing of results provided by expressions over data sources. Those shortcuts also miss dedicated support from UML Actions and while it is of course easy to construct a metamodel instance of the desired semantics, the reverse mapping into a code demands for annotation or stereotype to ease it.
- Macroscopic updates. While (which is also in the spirit of UML behaviour) we avoid macroscopic updates (e.g. updating many objects with a single statement without resorting to an iterating instruction), we found the following exceptions useful. First, we allow collections to be the input of object removal statement. Second, we allow to assign a collection to a multi-valued attribute or variable instead of the need of inserting its elements one by one. Due to the constraints imposed by UML compliance, this required an implicit use of iterative construct (that is, the ExpansionRegion).

While not providing a formal specification of the language here, we present below several illustrative code examples referring to the schema defined by the class diagram in Fig. 1, which is assumed to be defined inside a package named *Students*.

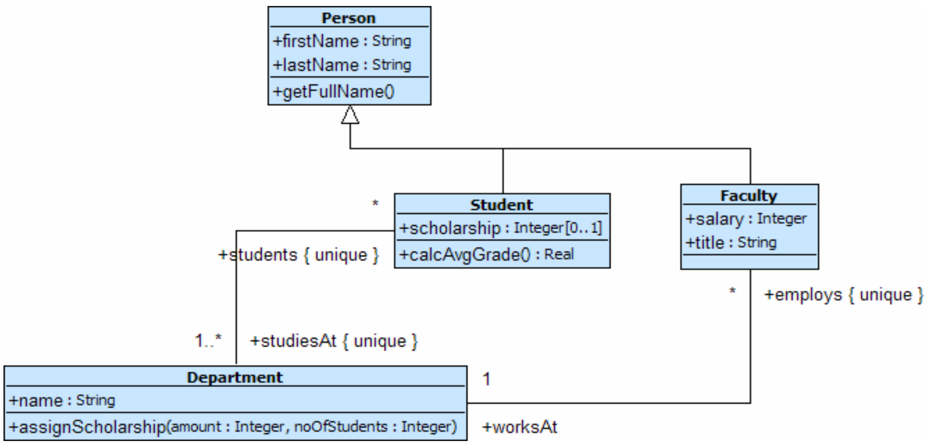


Fig. 1. Exemplary schema for code samples (the package name is "Students")

The first example illustrates a simple method of a pure query nature (i.e. having no side effects). Those kinds of methods could be called inside pure OCL statements. This distinction is possible to maintain in VIDE-L, as the calls of methods marked as having side-effects can be delegated to the CallOperationAction rather than handled at the OCL side. However, we currently do not enforce it in our language. Note also the OCL-style **context** declaration which specifies to which operation in the class model the given method body refers to. In the final version of our prototype the modelling environment will provide more assistance for this, so this header will not need to be directly used by the programmer.

```
context Students::Person.getFullName() : String
body {
  return firstName+' '+lastName;
}
```

The second example shows a more complex updating method, which uses link navigation and performs iterative updates of the objects selected by an OCL expression.

```
context Students::Department.assignScholarship(
 in amount : Integer, in noOfStudents : Integer) body {
  students->sortedBy(s |
        -s.calcAvgGrade()).subSequence(1,noOfStudents)
    foreach { s |
           if s.scholarship->size()=0
             then s.scholarship insert amount;
```

```
                **else** s.scholarship += amount;
              **endif**
            }
    }
```

The third example illustrates the link manipulation that moves employees to another department (the reverse links will be maintained automatically).

```
Department->allInstances()->select(name='SE').employs
foreach { f |
  f unlink worksAt;
  f link worksAt
   to Department->allInstances()->select(name ='IS');
}
```

The fourth example shows an ad-hoc query which is in this case a pure OCL.

```
Department->allInstances()->select(name='IS')
  ->collect(d |
      employs.title->asSet()->collect(t |
        Tuple{ title = t, avgSal =
          d.employs->select(title=t).salary->avg()}))
```

Note that two of those examples depend on the class extent retrieval. While this can be natural for some flavours of object schemas (like e.g. in ODMG), for typical cases we assume a presence of an object that will be pre-existing with respect to the application execution (rather than explicitly instantiated later) to provide an entry point to the application. For this purpose we have introduced the class stereotype «Module».

As can be seen from the above diagram and code samples, the textual syntax can be considered a bit eclectic, as it is influenced by three trends: UML type and multiplicity declarations, OCL with its specific syntax which influenced also the VIDE-L statements to take a more postfix-style syntactic patterns, plus some solutions coming from Java as the most popular general-purpose programming language. The positive aspect is that the syntax seems well suited for extensive contextual support when coding, which is to be provided by type checking mechanisms. This can be especially visible where query expressions are involved. Compare the marked steps of the code of example 2:

```
/*1*/students /*2*/->sortedBy(s |
      -s.calcAvgGrade()).subSequence(1,noOfStudents)
  foreach { s |
        if s.scholarship /*3*/->size()=0
          then s.scholarship insert amount;
          else s.scholarship += amount;
        endif }
```

with analogous code expressed with a syntax drawn from ODMG OQL [17] and popular programming languages:

```
foreach ( select s from students s
          order by s.calcAvgGrade()
          desc)[1..noOfStudents] as std {
            if ( not exists( s.scholarship )  )
              s.scholarship insert amount;
            else s.scholarship += amount;
}
```

It can be observed that the way the OCL syntax is arranged makes it easier and more natural to provide hints[2] than in case of the select-from-where pattern. At point 1 a list of names visible in the scope (starting from the most local ones) and statements could be presented to support that step of code creation. Similarly, at point 2 the list of available collection information can be presented (since the expression *students* returns a collection of objects) as well as the properties of the Student object (because OCL allows for building path expressions in 1:n direction). A slighter advantage in terms of the contextual hits can be achieved at point 3, where the selection of proper operator (OCL operation *size()*) may be performed from among of few choices determined by the context of expression *s.scholarship*.

The visual notation considerations are rather outside the scope of this paper. We just note the dilemma between choosing the traditional diagrammatic style of syntax (and aiming at "keyboard-less programming") or staying closer to textual style of coding though supporting it with visualization. The textual coding of this level of granularity is rather predominant. We are aware of only one action language depending on visual notation – namely Scrall [19] – which deals with a similar problem in terms of combining the visual and textual notation. The similarities with our language include:

- The idea of controlling the level of detail by collapsing and expanding code elements and resorting to textual code where it is more suitable.
- Considering data processing as the purpose of the language.

The following differences can be indicated:

- Scrall assumes relational data. It provides some high level operators, but does not provide a complete query language functionality comparable with OCL.
- Scrall supports the flow-style of behaviour specification which is made possible by the visual notation. VIDE currently does not directly cover this powerful feature, but the compliance with UML provides the capability of achieving it in futureextensions by embedding VIDE-L code inside the diagrams supporting UML Complete Activities.

5 Challenges of the Development Process

Apart from creating a standard-compliant language of adequate expressive power, MDA solutions need to address the problem of model transformations to automate code creation. Translating between high-level languages usually involves big complexity. Examples of the potential problems that need to be faced in that area include:

[2] This applies to some extent also to XQuery language [18].

- Translating PIM-defined application logic onto the multi-tier solutions of target platforms. The number of possible options in such translations complicates the process and / or undermines the idea of full platform-independence of the main model.
- Introducing a platform-independent specification of the presentation layer which is of high importance in various business applications.
- Dealing with data management – including schema definition on the target platform and hiding the "impedance mismatch" between today's database and programming languages behind a uniform platform-independent modelling language. When dealing with the code to be handled by a DBMS it is not only necessary to preserve its original semantics, but also to guarantee that the opportunities foroptimization will not be lost in the course of translation, which is essential for achieving acceptable performance and resource consumption in data-intensive applications.

In some applications, an alternative to those complex translations could be the idea of "model driven runtime" [20]. This means that a platform is available that is capable of directly executing models (e.g. represented in the form of the UML metamodel instance), so running an application does not require transformation to some other programming and / or query language. The cited paper arguments that the difference between having different platform-specific models derived and having many different model runtimes deployed is less substantial than it may appear. The described solution deals with simpler scenarios of application development (some Web applications are given as an example), where no complex application logic occurs and the presentation layer is closely driven by the schema of underlying database. Moreover, the runtime described in that paper deals with different kind of behavioural models as it "interprets OCL-annotated class diagrams and state machines".

Of course this solution is not always acceptable since the use of existing platform specific tools and environments is required by customers for the applications being created. That's why VIDE assumes developing respective model compilers.

However, we have provided a runtime for direct execution of models, as we have found it important for the following reasons:

- Current standardization efforts of UML should be backed with a reference implementation to verify the consistency of the language and to disambiguate its semantics through an operational definition.
- Availability of the engine that would allow direct execution of models seems to be a feature of primary importance for model-driven development tools, as a mean of model simulation (also in terms of tracking and debugging particular elements of the application at the PIM level and in terms of PIM artifacts). Since our current runtime engine provides rather straightforward implementation for particular model constructs compared to typical target platforms, it is an interesting option for the future development of complete model simulation and debugging environment.

Among other challenges to be faced by VIDE toolset is the integration with business modelling. This is important to meet the demand for business-process driven software development approaches and the Service Oriented Architecture viewpoint on the applications. A similar, but separate problem is an attempt to improve the business stakeholders' involvement into modelling and application prototyping.

The above considerations set the following assumptions for the current work on the VIDE project:

- UML compliant PIM, provided with the means and level of precision of a programming language becomes the central artefact of the software construction.
- Model execution capability allows to validate the system functionality directly from the tool (i.e. without the steps of explicit code generation and its deployment).
- Appropriate elements of model behaviour may be distinguished as externally available service interfaces and equipped with a complete Web service descriptions for the purpose of model's direct execution.
- For the scenarios that allow it, the model may be rather directly deployed in the flavour of a MDR using its execution engine (purely object-oriented database system prototype).
- If creating application functionality on the Java platform is the aim, a model compiler (currently under design) will be used to generate Java code defining the application logic and using data persistency through the JDO interface [21].
- At the side of initial phases of a software development process, a significant amount of work has been allocated to describe gradual, incremental transition from informal requirements set and computation independent model towards precise PIM.

6 Conclusions and Future Work

The aim of the research outlined in the paper can be considered challenging for several reasons. The first challenge is to provide adequate and advantageous means of software specification, taking into account new kinds of user profiles assumed by the MDA development process. It has to recognize and properly balance the needs of such user groups as modellers familiar with CASE tools, programmers familiar with traditional programming and query languages, and non-IT stakeholders seeking for model accessibility. Another challenge is the need of alignment with modelling standards on one side and target platform technologies at the other side. Those considerations draw two important goals for the next step of our research:

1. Completing the existing textual prototype with the implementation of selected concepts of visual notation and gaining feedback from users.
2. Investigating the possibilities of developing model compilers from the kind of modelling constructs VIDE employs, onto the implementation technologies used in the industry.

References

1. Object Management Group: Unified Modeling Language: Superstructure version 2.1.1, formal/2007-02-05 (February 2007)
2. Mellor, S.J., Scott, K., Uhl, A., Weise, D.: MDA Distilled: Principles of Model-Driven Architecture. Addison-Wesley, Reading (2004)
3. McNeile, A.: MDA: The Vision with the Hole (2003), `http://www.metamaxim.com`

4. Warmer, J., Kleppe, A.: Object Constraint Language, The: Getting Your Models Ready for MDA. Addison-Wesley, Reading (2003)
5. Object Management Group: Meta Object Facility (MOF) Core Specification version 2.0, formal/06-01-01 (January 2006)
6. Object Management Group: Object Constraint Language version 2.0, formal/06-05-01 (May 2006)
7. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal: Model-Driven Software Development 45(3) (2006)
8. Ambler, S.W.: A Roadmap for Agile MDA. Ambysoft, upd (2007), `http://www.agilemodeling.com/essays/agileMDA.htm`
9. Thomas, D.A.: MDA: Revenge of the Modelers or UML Utopia? IEEE Software 21(3), 15–17 (2004)
10. Fowler, M.: UML as Programming Language (2003), `http://www.martinfowler.com/bliki/UmlAsProgrammingLanguage.html`
11. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model-Driven Architecture Addison Wesley. Addison Wesley, Reading (2002)
12. Wilkie, I., King, A., Clarke, M., Weaver, C., Rastrick, C., Francis, P.: UML ASL Reference Guide ASL Language Level 2.5 Manual Revision D, Kennedy Carter Limited (2003), `http://www.omg.org/docs/ad/03-03-12.pdf`
13. Eclipse Modeling Project, Model Development Tools. Eclipse Foundation, `http://www.eclipse.org/modeling/mdt/`
14. Object Management Group: Semantics of a Foundational Subset for Executable UML Models. Request For Proposal. ad/2005-04-02
15. Jodłowski, A., Habela, P., Płodzień, J., Subieta, K.: Dynamic Object Roles - Adjusting the Notion for Flexible Modeling. In: Proc. of the International Database Engineering and Application Symposium (IDEAS), pp. 449–456. IEEE Computer Society, Coimbra (2004)
16. Object Management Group: Relationship Service Specification version 1.0. formal/00-06-24 (April 2000)
17. Cattel, R.G.G., Barry, D.K. (eds.): Object Data Management Group, The Object Database Standard ODMG, Release 3.0. Morgan Kaufmann, San Francisco (2000)
18. World Wide Web Consortium: XQuery 1.0: An XML Query Language. W3C Recommendation 23 (January 2007), `http://www.w3.org/TR/xquery/`
19. Starr, L.: Starr's Concise Relational Action Language version 1.0 (August 2003), `http://www.modelint.com/downloads/mint.scrall.tn.1.pdf`
20. Pleumann, J., Haustein, S.: A Model-Driven Runtime Environment for Web Applications. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 190–204. Springer, Heidelberg (2003)
21. Java Data Objects Expert Group: Java^{TM} Data Objects 2.0. JSR 243 Final 23 (February 2006), `http://java.sun.com/javaee/technologies/jdo/`

Towards UML-Intensive Framework for Model-Driven Development

Darius Silingas[1,2] and Ruslanas Vitiutinas[1,3]

[1] No Magic, Inc., Lithuanian Development Center
Savanoriu av. 363-IV, LT-44242 Kaunas, Lithuania
[2] Kaunas University of Technology,
Information Systems Chair
Studentu 50-313a, LT-51368 Kaunas, Lithuania
[3] Vytautas Magnus University,
Faculty of Informatics,
Vileikos 8-409, LT-44404 Kaunas, Lithuania
{Darius.Silingas,Ruslanas.Vitiutinas}@nomagic.com

Abstract. The paper describes a conceptual framework for model-driven development based on a concise application of UML and modeling tool functionality. A case study of modeling software for library management is presented as an illustration of how to apply the proposed framework. Modeling tool features such as model transformations, code generation cartridges, model validation, dependency matrix, model metrics, model comparison, and model refactoring are presented as enablers for efficient model-driven development. The presented ideas and samples are based on industrial experience of authors who work as trainers and consultants for the modeling tool MagicDraw UML.

Keywords: UML, MDA, Model-Driven Development, MagicDraw.

1 Introduction

Model-Driven Architecture (MDA) is a new software development trend, which gains popularity in industry. Unified Modeling Language (UML) is considered a key tool for preparing source models for code generation in MDA environments. Recently, the language has undergone major changes moving from UML 1.4 version to UML 2. The rational behind most of these changes was to provide a better infrastructure for MDA [1], [4]. However, recent MDA tools make rather limited use of UML 2 elements focusing mostly on class diagrams and activity or state diagrams. In the paper we present the framework for concisely developing UML 2 models for domain analysis, requirements specification, architectural decomposition, detailed design, implementation, and testing. We also indicate what UML tool functionality is useful, and which UML elements should be used for modeling implementation. Most of the presented ideas come from industrial experience of the authors, who work as trainers and consultants for the MagicDraw UML modeling tool.

2 Conceptual Framework for Model-Driven Development

MDA defines three abstraction levels of modeling: *Computation-Independent Modeling* (CIM), *Platform-Independent Modeling* (PIM), and *Platform-Specific Modeling* (PSM). We define the following major modeling tasks that should be completed while developing software system:

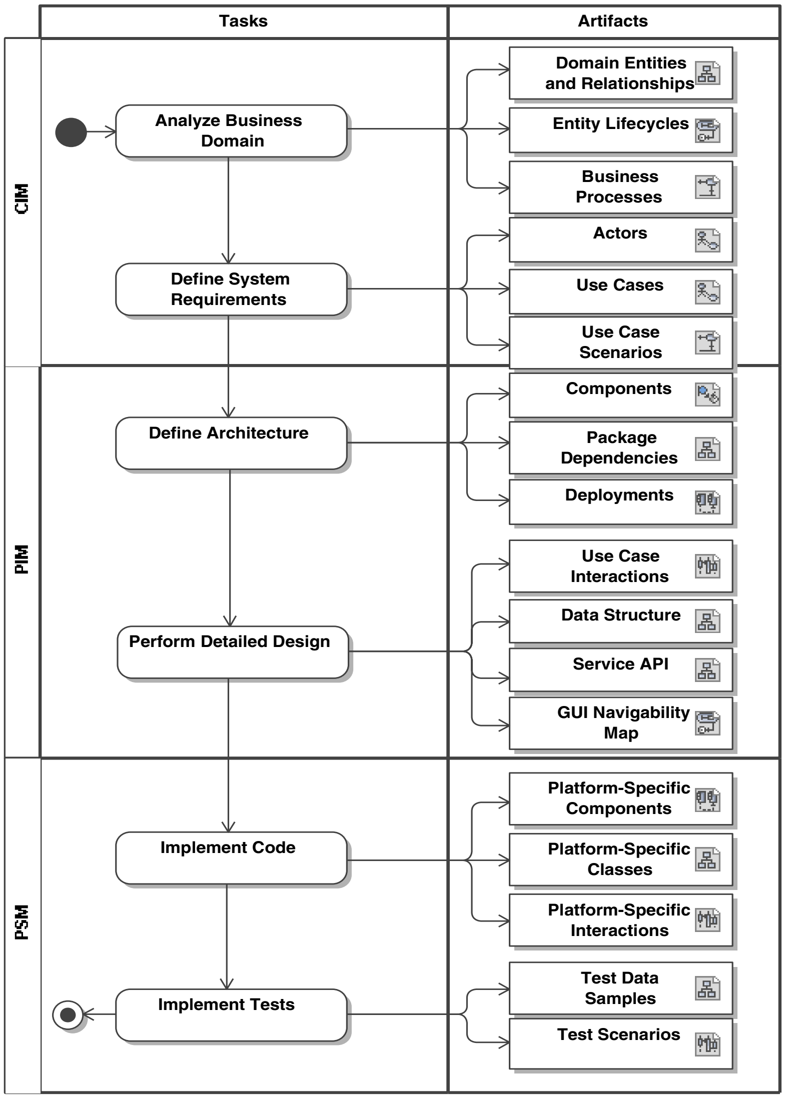


Fig. 1. Conceptual framework for model-driven development

1. Analysis of business domain (purely focusing on CIM);
2. Definition of system requirements (mostly focusing on CIM);
3. Definition of high-level system architecture (mostly focusing on PIM);
4. Detailed design (mostly focusing on PIM);
5. Implementing code (mostly focusing on PSM);
6. Implementing tests (mostly focusing on PSM).

Each of these tasks should create of set of UML-based modeling artifacts. We present these tasks and artifacts in a conceptual framework, which is defined by UML activity diagram presented in Fig. 1. Although the framework defines the sequential logical flow of tasks, we want to emphasize that the nature of modern modeling is iterative – you need to repeat the tasks in iterations and come back for model updates.

3 MagicLibrary: A Case Study

In this chapter we will present more detailed description of modeling tasks and present major modeling artifacts for a case study system that will serve as illustrations for applying the conceptual modeling framework introduced in the previous chapter.

3.1 A Case Study Problem Statement

A large organization maintains a library, which contains books, audio and video records. The organization made a decision to implement software system MagicLibrary dedicated for facilitating library usage and management.

MagicLibrary should support three types of users – librarian, reader, and administrator. Both reader and librarian are able to search for library items. Each library item is assigned to one or more categories and contains a list of keywords (optional). Item may be found either by browsing the category tree or searching for items by their property values. If reader finds a desirable item, he makes a reservation for it. If the item is immediately available then the reader is informed that he may contact librarian for loaning it according to the made reservation.

If the item is currently loaned out or assigned to another reservation then the reservation is put to the ordered waiting list. When the waiting reservation becomes available the system notifies the user. Notifications are sent either by e-mail or SMS according to the user preferences. Available reservations are automatically cancelled if reader doesn't come to take the item on loan for a period of time defined in system settings. Librarian registers the loan of the item and sets the due return date.

Librarian also registers the return of the loaned item. If a reader has kept the loaned item after the due date, he is given a penalty.

Reader may review his profile, which contains his reservations, loans, requests, and his personal data. Librarian is responsible for managing inventory data: titles, items and categories. Librarian is also responsible for managing MagicLibrary users and configuring system settings like default loan period, available reservation timeout, max reservations per reader, etc.

3.2 Analyze Business Domain

The purpose of business domain analysis is to understand how the business system works before going into development of software systems that automate some of business-defined procedures. We think that the basic views of business domain are identification of business entities and their structural relationships, business processes, and the lifecycles of business entities that have important states that allow different business actions.

We recommend starting with definition of business entities and their relationships using simple class diagram using classes displaying only name compartment and named associations, see Fig. 2. Additionally, you may define association role cardinalities for better understanding of relationship nature. Such diagram serves as visual dictionary of business terminology. The terms defined in it should be used consistently in all other diagrams and model elements. Of course, while modeling, you will need to come back and update this diagram to reflect the discovered changes.

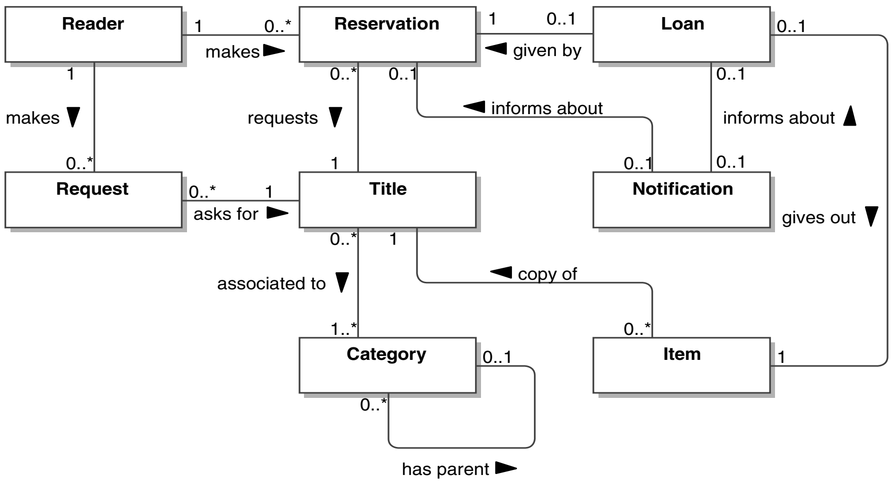


Fig. 2. Library business domain entities and relationships class diagram

A business entity lifecycle can be presented by an analysis-level state diagram indicating what behaviors are possible on which entity state, what actions trigger transitions between states, and what activities should be executed as side effect of transition, entering a state or exiting from it. In concise modeling, states machine model should be assigned to entity class using classifier's behavior property that is available since UML 2 [1]. A sample state machine, defined in Fig. 3, should be created inside class *Item*, and assigned as a behavior for that class.

Very important modeling artifacts are business processes that can be defined using either pure UML activity diagrams or *Business Process Modeling Notation* (BPMN) [3]. In such diagrams, the focus is on the sequence of tasks, separation of responsibilities, decision points, triggering business events, communications between processes in different organizations, and exchanged data, mostly documents.

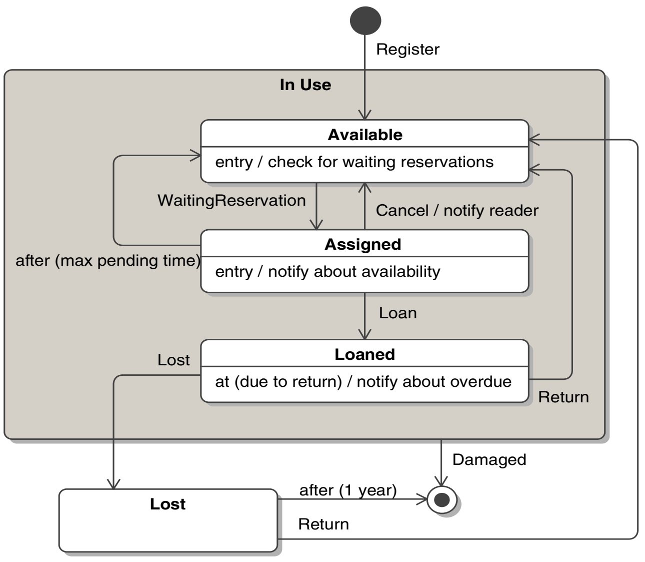


Fig. 3. Analysis-level state diagram presenting lifecycle of *Item* entity

3.3 Define System Requirements

Software development is driven by the expressed requirements. Most of the major approaches to requirements analysis are based on use case method. In the famous 4+1 architectural views model, use cases are defined as a central modeling artifact [5]. We suggest the following use case modeling workflow:

1. Define actors (in a separate diagram) grouping them into primary (main users), secondary (administration, maintenance, support), and system actors.
2. Define main system use cases in a sketch use case diagram.
3. Group the created use cases into packages according to their coherence.
4. Prepare use case package overview diagram, showing which actors uses which use case package. Alternatively, use case model overview can be done using so called dependency matrix, which in tabular form shows the relationships between actors and use cases grouped by packages.
5. Prepare use case package details diagram, showing package use cases, their associations with actors, relationships between use cases including uses cases from different packages (shown outside the package symbol), Fig. 4.
6. Prepare activity diagrams visualizing scenarios of complex use cases, Fig. 5. In the model, the activities should be nested within appropriate use cases and assigned as their behaviors. Some actions may call reusable activities.
7. Document use cases according to pre-defined templates, e.g. *Rational Unified Process* or *Process Impact* defined use case templates.

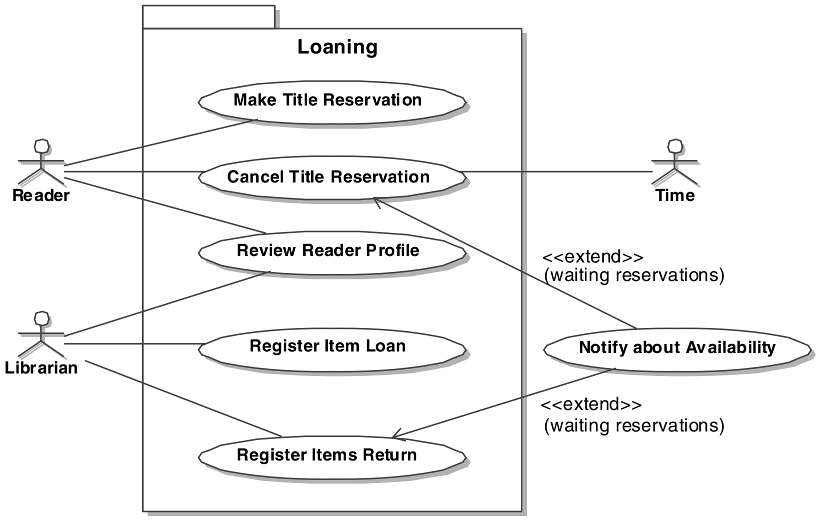


Fig. 4. Use case diagram showing details of use case package *Loaning*

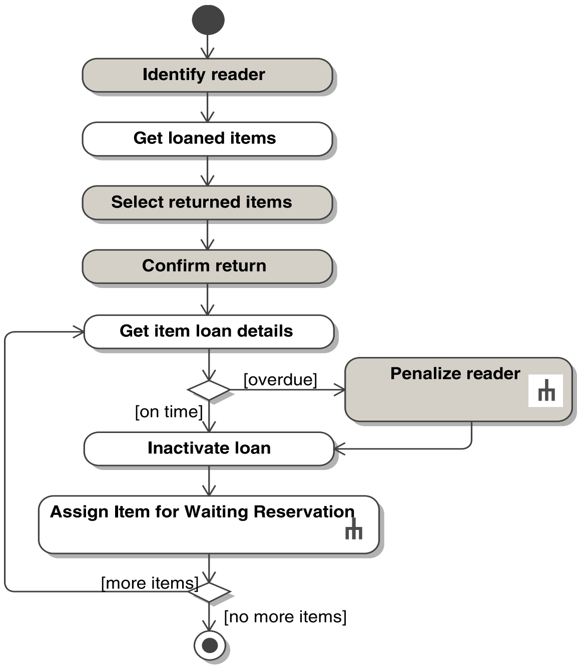


Fig. 5. Activity diagram showing scenarios for use case *Register Items Return*

However, use cases define just the user-level functional requirements. In practice, functional requirements are usually decomposed into smaller-grained functional requirements; also many types of non-functional requirements are specified [6]. For enabling structural modeling of requirements and their relationships, we suggest to prepare a custom class diagram enhancement for requirements modeling. A similar approach is taken in SysML [2]. This approach also enables requirements traceability using tool-generated dependency matrices.

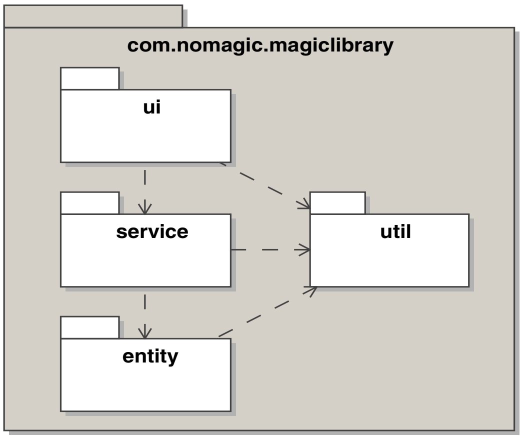


Fig. 6. Package diagram showing dependencies between top-level packages

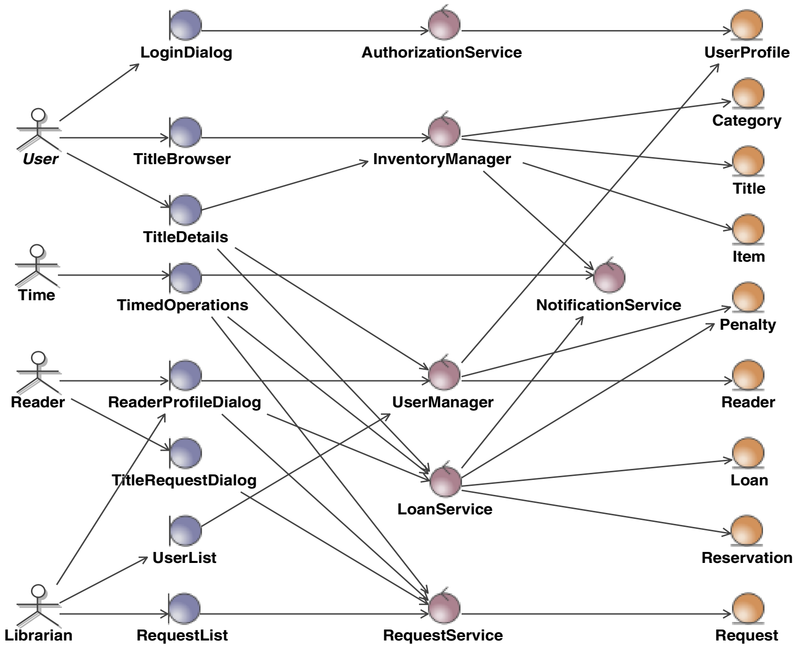


Fig. 7. Robustness analysis of relationships between components in layers

3.4 Define Architecture

Design of the software system starts by specifying its high-level architecture, which is usually modeled in structural component, package dependency, and deployment diagrams. Most of the modern business-oriented software systems are built on the layered architecture pattern. For such systems, we recommend to start architectural design with package dependency and robustness analysis diagrams [7]. Package dependency diagram shows how the system is organized into layers, Fig. 6.

Robustness analysis diagram defines the major components in different layers – interface boundaries, controll services, data entities, – and their relationships crossing layer boundaries, Fig. 7. It is important to show which actor uses which interface boundaries. The inspiration for the data entities are the conceptual entities, for the controll services – use case packages, and the interface boundaries need to be invented

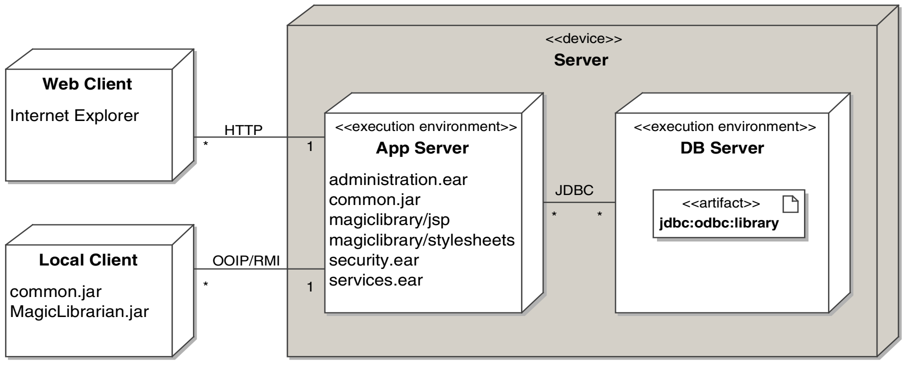
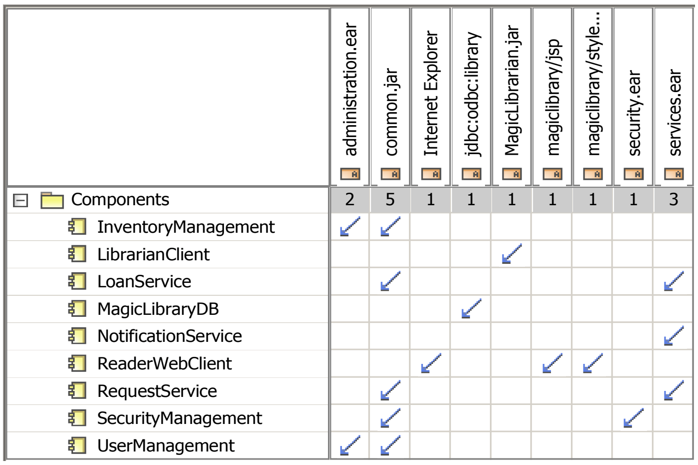


Fig. 8. Deployment architecture



	administration.ear	common.jar	Internet Explorer	jdbc:odbc:library	MagicLibrarian.jar	magiclibrary/jsp	magiclibrary/style...	security.ear	services.ear
Components	2	5	1	1	1	1	1	1	3
InventoryManagement	↙	↙							
LibrarianClient					↙				
LoanService		↙							↙
MagicLibraryDB				↙					
NotificationService									↙
ReaderWebClient			↙			↙	↙		
RequestService		↙							↙
SecurityManagement		↙						↙	
UserManagement	↙	↙							

Fig. 9. Component-artifact manifestation

at this point and may be refined later. Each of the defined components should be placed into appropriate packages. In the subsequent detailed design task, it is necessary to refine each component by modeling its details.

It is common at architectural design level to define at least the major ideas about the system deployment. This usually includes platform-specific information. At this point, it is important to define only the major deployment artifacts, topology of network, and communication protocols, Fig. 8.

While UML 2.0 districts the deployment of components, and insists of deploying artifacts, it is necessary to separate logical components and physical artifacts. The relationships between those two sets are modeled by manifestation relationships. We recommend dependency matrices for representing and editing manifestations, Fig. 9.

3.5 Perform Detailed Design

After the structural components are identified, it is time to provide more details for each of them. Different aspects are important for different layers:

- Data entity layer should focus on specification of data class attributes and details of associations, such as navigability and role names, Fig. 10.
- Service layer should focus on specification of API, which is defined in public operations of services. We recommend specifying service operations while designing interactions for use case scenarios. With this approach, it is easier to see the context in which operations are used, which allows define appropriate responsibilities and their contracts (operation parameters, returns, visibility and other properties). We consider it as a modeling substitute for the test-driven programming approach since we define the structural class features – operations, while designing behavioral scenarios, Fig. 11. Later service API can be visualized using class diagrams showing only operations and dependencies between services.

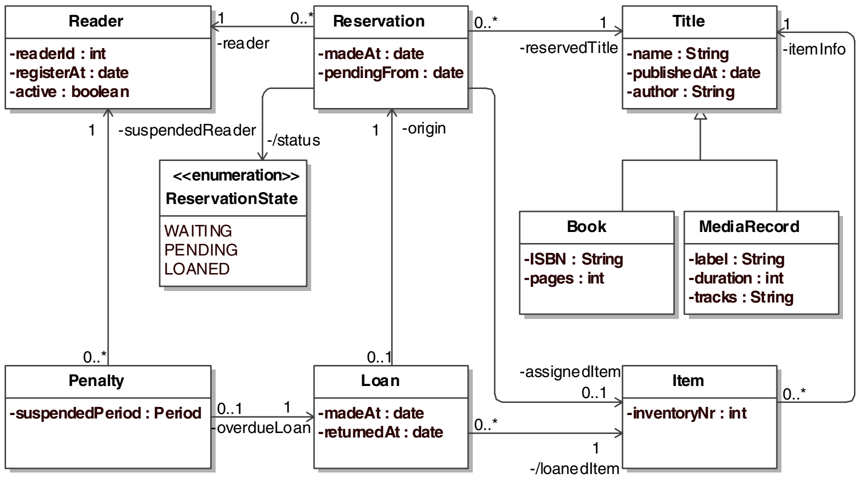


Fig. 10. Class diagram emphasizing data entity attributes and relathioships

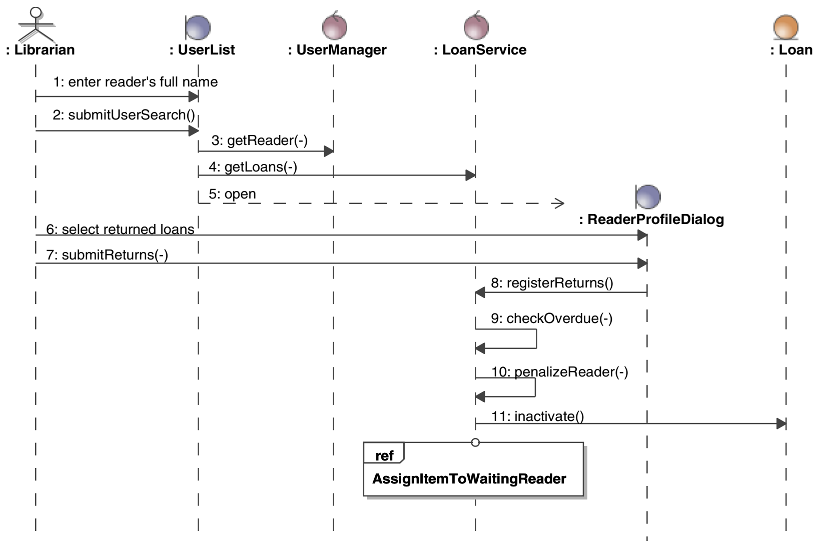


Fig. 11. Sequence diagram showing interaction of identified components for implementing Register Item Return “happy day” scenario

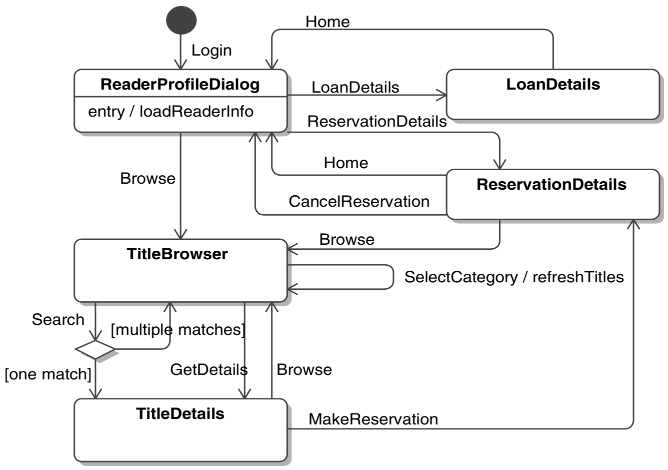
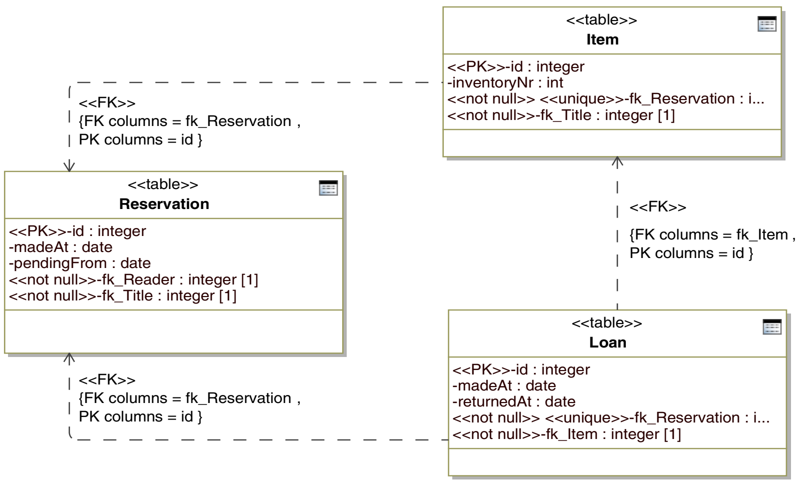


Fig. 12. User interface navigation schema for actor *Reader*

- User interfaces should focus on GUI navigability maps and definition of inner structure of GUI elements. The former is best modeled with state diagram, where each state defines separate GUI component and the transition triggers define GUI events, Fig. 12. The later is usually done using graphical prototypes that are not based on UML. The composite structure diagram introduced in UML 2 is a potential candidate for this task but the tool support for modeling this in user-friendly way is still not available.

3.6 Implement Code

In model-driven approach, we use the created model artifacts for generating code. For enabling code generation, we need to enrich PIM level models with platform-specific



<<table>> Item
<<PK>>-id : integer
-inventoryNr : int
<<not null>> <<unique>>-fk_Reservation : i...
<<not null>>-fk_Title : integer [1]
<<FK>>
{FK columns = fk_Reservation , PK columns = id }
<<table>> Reservation
<<PK>>-id : integer
-madeAt : date
-pendingFrom : date
<<not null>>-fk_Reader : integer [1]
<<not null>>-fk_Title : integer [1]
<<FK>>
{FK columns = fk_Item , PK columns = id }
<<table>> Loan
<<PK>>-id : integer
-madeAt : date
-returnedAt : date
<<not null>> <<unique>>-fk_Reservation : i...
<<not null>>-fk_Item : integer [1]
<<FK>>
{FK columns = fk_Reservation , PK columns = id }

Fig. 13. A fragment of PSM for relational database structure

Table 1. A fragment of DDL script generated from PSM for database design

```
CREATE SCHEMA MagicLibrary;
...
CREATE TABLE Reservation (
  id integer PRIMARY KEY,
  madeAt date,
  pendingFrom date,
  fk_Reader integer NOT NULL,
  fk_Title integer NOT NULL,
  FOREIGN KEY(fk_Reader) REFERENCES Reader (id),
  FOREIGN KEY(fk_Title) REFERENCES Title (id)
);
CREATE TABLE Item (
  id integer PRIMARY KEY,
  inventoryNr int,
  fk_Reservation integer NOT NULL UNIQUE,
  fk_Title integer NOT NULL,
  FOREIGN KEY(fk_Reservation) REFERENCES Reservation (id),
  FOREIGN KEY(fk_Title) REFERENCES Title (id)
);
CREATE TABLE Loan (
  id integer PRIMARY KEY,
  madeAt date,
  returnedAt date,
  fk_Reservation integer NOT NULL UNIQUE,
  fk_Item integer NOT NULL,
  FOREIGN KEY(fk_Reservation) REFERENCES Reservation (id),
  FOREIGN KEY(fk_Item) REFERENCES Item (id)
);
...
```

information. It is recommended to do it via semi or fully automated model-to-model transformations. Code generation from PSM models is implemented either by plug-able code generation cartridges or model-to-code transformations. As a sample, we present a fragment for PSM model for relational database design, Fig. 13 and DDL script generated from this PSM model, Table 1.

3.7 Implement Tests

For implementation of tests, a modeler may use object diagrams for visualizing data structures that should be used in testing. The sequence diagram, which is not a very good tool for modeling algorithms, is perfectly suitable for defining test scenarios. The assertion fragment introduced in UML 2 is also a very useful construct for that purpose. Both these diagrams are suitable artifacts for code generation, [8].

4 Enabling Toolkit for Model-Driven Development

For enabling efficient model-driven development, the UML modeling environment should provide multiple features. Below we list features that are already supported in industrial modeling tools like MagicDraw UML:

- Concise integration of model elements through UML-define properties;
- Model validation;
- Plug-able patterns;
- Plug-able model transformations;
- Plug-able code generation cartridges;
- Project decomposition to several modules;
- Interactive modeling teamwork;
- Tracking and analysis of model element relationships;
- Automated relationship matrices representing model element relationships;
- Model metrics, calculated according to the model content;
- Customizable model documentation reports.

The following features are also highly demanded, but are not yet implemented (or implemented very poorly) in the state-of-the-art modeling tools:

- Model refactoring;
- Merging changes between different model versions;
- Modeling unit tests for supporting test-driven modeling approach.

With the mentioned tools at their hands, the modelers can apply a model-driven development in efficient ways allowing increase of development speed and quality.

5 Summary

We have introduced the conceptual UML-intensive framework for model-driven development and provided a case study based model examples illustrating essential work products resulting from the tasks presented in the framework. In the context of these artifacts we discussed the ideas of how to increase modeling efficiency with desirable automation features. Finally, we have briefly introduced the modeling environment features that need to be supported for enabling efficient and agile software development while applying the proposed framework. We hope that the presented information is a valuable starting point for more detailed industrial and academic research on the topic of UML-intensive model-driven development.

References

1. Object Management Group. UML 2.1.1 Unified Modeling Language: Superstructure, Specification (2007)
2. Object Management Group. Systems Modeling Language (SysML), Specification (2006)
3. Object Management Group. Business Process Modeling Notation (BPMN), Specification (2006)
4. Meservy, T.O., Fenstermacher, K.D.: Transforming Software Development: An MDA Road Map. Computer 38(9), 52–58 (2005)
5. Kruchten, P.B.: The 4+1 View Model of architecture. Software, IEEE 12(6), 42–50 (1995)
6. Wiegers, K.E.: Software Requirements, 2nd edn. Microsoft Press, Redmond (2003)
7. Lange, C.F.J., Chaudron, M.R.V.: UML Software Architecture and Design Description. In: van Leeuwen, J. (ed.) Software, March-April 2006, vol. 23(2), pp. 40–46. IEEE, Los Alamitos (2006)
8. Nebut, C., Fleurey, F., Le Traon, Y., Jezequel, J.-M.: Automatic Test Generation: A Use Case Driven Approach. IEEE Transactions on Software Engineering 32(3), 140–155 (2006)

UML Static Models in Formal Approach

Marcin Szlenk

Warsaw University of Technology,
Institute of Control & Computation Engineering,
Nowowiejska 15/19, 00-665 Warsaw, Poland
M.Szlenk@ia.pw.edu.pl

Abstract. The semantics of models written in UML is not precisely defined. Thus, it is hard to determine, how a given change in a model influences its meaning and, for example, to verify whether a given model transformation preserves the semantics of the model or not. In the paper a formal (mathematical) semantics of key elements of the UML static models is presented. The aim is to define the basic semantic relations between models: a consequence (implication) and equivalence. The goal of the definitions and examples presented in the article is to form a very basic, concise, theoretical foundation for the formal comparison of the UML static models, based on their meanings.

Keywords: Software modeling, UML, Formal reasoning.

1 Introduction

Unified Modeling Language (UML) [9,12] is a visual modeling language that is used to specify, construct and document software systems. The UML has been adopted and standardized by the *Object Management Group* (OMG). The UML specification [12], published by OMG, is based on a metamodeling approach. The metamodel (a model of UML) gives information about the abstract syntax of UML, but does not deal with semantics, which is expressed in a natural language. Furthermore, because UML is method-independent, its specification tends to set a range of potential interpretations rather than providing an exact meaning.

As far as software modeling is concerned, we can distinguish two types of models: dynamic and static. The dynamic model is used to express and model the behaviour of a problem domain or system over time, whereas the static model shows those aspects that do not change over time. UML static models are mainly expressed using a class diagram that shows a collection of classes and their interrelationships, for example generalization/specialization and association.

After the first UML specification was published, various propositions of UML formalization have appeared. The semantics of class diagrams was expressed using such formal languages as Z [4,5], PVS-SL [1], description logic [2] and RAISE-SL [6]. Some of the works were restricted to the semantics of models, while the others were concerned with the issues of reasoning about models and model transformation. It seems that the subject of reasoning about UML static models still lacks a formal approach to the problem of the semantic equivalence of two models. There are some informal approaches but they result in unverifiable model transformation rules (see e.g. [3,8]).

In the paper the syntax and semantics of a UML static model restricted to key elements of a class diagram are formally defined. The definitions which are presented here adhere to the UML 2. Using the proposed formalization, we show how one can reason about UML static models in a fully formal way, especially about their equivalence.

2 Metalanguage

As a language for defining the semantics of UML static models, we use basic mathematical notation. In this section we briefly outline only the list and function notation, as they may vary in different publications.

For a set A, $\mathcal{P}(A)$ denotes the set of all the subsets of A, and A^* denotes the set of all the finite lists of elements of A. The function $\mathbf{len}(l)$ returns the length of a list l. For simplicity, we add the expression $A^{*(2)}$, which denotes the set of all finite lists with a length of at least 2. The function $\pi_i(l)$ projects the i-th element of a list l, whereas the function $\overline{\pi}_i(l)$ projects all but the i-th element. The list $[a_1, \ldots, a_n]$ is formally equal to the tuple $(a_1, \ldots, a_n)$. For a finite set A, $|A|$ denotes the number of elements of A.

The partial function from A to B is denoted by $f\colon A \rightharpoonup B$, where the function $\mathbf{dom}(f)$ returns the domain of f. The expression $f\colon A \to B$ denotes the total function from A to B (in this case it holds $\mathbf{dom}(f) = A$).

3 Syntax

The key concepts used in UML static models: class, association and association class are considered here. All of them are types of classifier.[1] Taxonomical relationships among them, which are defined in the UML metamodel, are shown in Fig. 1. It is worth emphasizing that an association class is a single model element which is both an association and a class.

Definition 1 (Classifiers). *With* Classifiers *we denote a set of all the classifiers (the names of classes, associations and association classes) which may appear in a static model.*

Below we formally define abstract syntax of simple UML static models. The syntax is defined in a way which reflects the relationships from Fig. 1. It makes both the definition of the syntax and the semantics (discussed later) more concise.

Definition 2 (Model). *By a* (static) model *we understand a tuple*

$$\mathrm{M} = (\mathrm{classes}, \mathrm{assocs}, \mathrm{ends}, \mathrm{mults}, \mathrm{specs}), \textit{ where:} \tag{1}$$

1. M.classes *is a set of classes:*

$$\mathrm{M.classes} \subseteq \mathrm{Classifiers}. \tag{2}$$

[1] Association is included as a type of classifier since the introduction of UML 2.0.

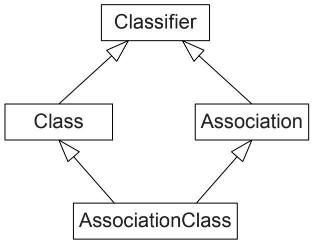


Fig. 1. The part of the hierarchy of classifiers in the UML metamodel [12]

2. M.assocs *is a set of associations:*

$$\text{M.assocs} \subseteq \text{Classifiers}. \tag{3}$$

For the model M*, a set of association classes and a set of all classifiers are thus respectively defined as:*

$$\text{M.asclasses} =_{def} \text{M.classes} \cap \text{M.assocs}, \tag{4}$$

$$\text{M.classifiers} =_{def} \text{M.classes} \cup \text{M.assocs}. \tag{5}$$

3. M.ends *is a function of association ends. The function maps each association to a finite list of at least two, not necessarily different, classes participating in the association:*

$$\text{M.ends}\colon \text{M.assocs} \to \text{M.classes}^{*(2)}. \tag{6}$$

The position on the list M.ends(as) *uniquely identifies the association end. An association class cannot be defined between itself and something else [12, p. 47]:*

$$\forall ac \in \text{M.asclasses} \cdot \forall i \in \{1, \ldots, \mathbf{len}(\text{M.ends}(ac))\} \cdot \\ \pi_i(\text{M.ends}(ac)) \neq ac. \tag{7}$$

4. M.mults *is a function of multiplicity of association ends. Multiplicity is a non-empty set of non-negative integers with at least one value greater than zero. The default multiplicity is the set of all non-negative integers (*$\mathbb{N}$*). The function assigns to each association a list of multiplicity on its ends:*

$$\text{M.mults}\colon \text{M.assocs} \to (\mathcal{P}(\mathbb{N}) \setminus \{\emptyset, \{0\}\})^{*(2)}. \tag{8}$$

As before, the position on the list M.mults(as) *identifies the association end. The multiplicity must be defined for each association end:*

$$\forall as \in \text{M.assocs} \cdot \mathbf{len}(\text{M.mults}(as)) = \mathbf{len}(\text{M.ends}(as)). \tag{9}$$

5. M.specs *is a function of specializations. The function assigns to each classifier a set of all (direct or indirect) its specializations:*

$$\text{M.specs}\colon \text{M.classifiers} \to \mathcal{P}(\text{M.classifiers}). \tag{10}$$

The specialization hierarchy must be acyclical [12, p. 53], what means that a classifier cannot be its own specialization:

$$\forall cf \in \text{M.classifiers} \cdot cf \notin \text{M.specs}(cf). \tag{11}$$

By default a classifier may specialize classifiers of the same or a more general type [12, p. 54], i.e. class may be a specialization of class; association may be a specialization of association; association class may be a specialization of association class, class or association. Formally:

$$\forall cl \in \text{M.classes} \cdot \text{M.specs}(cl) \subseteq \text{M.classes}, \tag{12}$$

$$\forall as \in \text{M.assocs} \cdot \text{M.specs}(as) \subseteq \text{M.assocs}, \tag{13}$$

$$\forall cf \in \text{M.classifiers} \cdot \text{M.specs}(cf) \nsubseteq \text{M.asclasses} \Rightarrow \\ cf \notin \text{M.asclasses}. \tag{14}$$

An association specializing another association has the same number of ends:

$$\forall as_1, as_2 \in \text{M.assocs} \cdot as_2 \in \text{M.specs}(as_1) \Rightarrow \\ \mathbf{len}(\text{M.ends}(as_1)) = \mathbf{len}(\text{M.ends}(as_2)), \tag{15}$$

which are connected to the same classifiers as in a specialized association or to their specializations [12, p. 39]:

$$\forall as_1, as_2 \in \text{M.assocs} \cdot as_2 \in \text{M.specs}(as_1) \Rightarrow \\ \forall i \in \{1, \ldots, \mathbf{len}(\text{M.ends}(as_1))\} \cdot \pi_i(\text{M.ends}(as_2)) \in \\ \{\pi_i(\text{M.ends}(as_1))\} \cup \text{M.specs}(\pi_i(\text{M.ends}(as_1))). \tag{16}$$

The above definition does not include directly attributes of classes. However, an attribute has the same semantics as an association. An example of attributes and corresponding associations are shown in Fig. 2.[2]

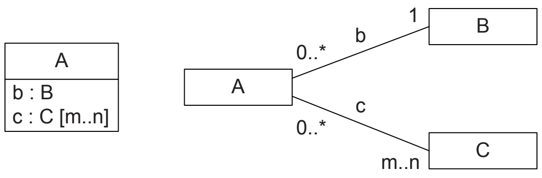


Fig. 2. Attributes and associations

Definition 3 (Models). Models *denotes a set of all the models, as in definition 2.*

4 Semantics

A classifier describes a set of instances that have something in common [9]. An instance of a class is called an object, whereas an instance of an association is

[2] This unification of attributes and associations is new to UML 2.0.

called a link. A link is a connection between two or more objects of the classes at corresponding positions in the association. An instance of a class association is both an object and a link, so it can both be connected by links and can connect objects.

Definition 4 (Instances). Instances *denotes a set of all the potential instances of the classifiers from the set* Classifiers.

The existing instances of a classifier are called its extent. If two classifiers are linked by a specialization relationship, then each instance of a specializing (specific) classifier is also an instance of a specialized (general) classifier [12]. In other words, the extent of the specific classifier is a subset of the extent of the general one.

The classifier extent usually varies over time as objects and links may be created and destroyed. Thus, the classifiers' extents form a snapshot of the state of a modelled problem domain or system at a particular point in time.

Definition 5 (State). State *is a pair*

$$\mathrm{S} = (\mathrm{instances}, \mathrm{ends}), \textit{ where:} \tag{17}$$

1. S.instances *is a partial function of extents. The function maps each classifier to a set of its instances (extent):*

$$\mathrm{S.instances}\colon \mathrm{Classifiers} \rightharpoonup \mathcal{P}(\mathrm{Instances}). \tag{18}$$

2. S.ends *is a partial function of link ends. The function assigns to each instance of an association, i.e. link, a list of instances of classes (objects) which are connected by the link:*

$$\mathrm{S.ends}\colon \mathrm{Instances} \rightharpoonup \mathrm{Instances}^{*(2)}. \tag{19}$$

The position on the list uniquely identifies the link end, which on the other hand, corresponds to an appropriate association end.

Definition 6 (States). *With* States *we denote a set of all the states as in the definition 5.*

The static model shows the structure of states (of a given domain or system) or, from a different point of view, defines some constraints on states. Thus, the model can be interpreted as the set of all such states in which the mentioned constraints are satisfied. Below we define the relationship between models and states as a relation of satisfaction: $\mathrm{Sat} \subseteq \mathrm{Models} \times \mathrm{States}$. If $\mathrm{Sat}(\mathrm{M}, \mathrm{S})$ holds then the constraints expressed as model M are satisfied in the state S. Next, we formally define the meaning of a model as the set of all states in which the model is satisfied.

Definition 7 (Satisfaction). *Let* $\mathrm{S} \in \mathrm{States}$ *and* $\mathrm{M} \in \mathrm{Models}$. *The model* M *is* satisfied in the state S *and we write*

$$\mathrm{Sat}(\mathrm{M}, \mathrm{S}), \textit{ if and only if:} \tag{20}$$

1. S *specifies the extents of all classifiers in* M *(and maybe others, not depicted in the model* M*)*[3]*:*

$$\text{M.classifiers} \subseteq \mathbf{dom}(\text{S.instances}). \tag{21}$$

2. *An instance of a given association only connects instances of classes participating in this association (on the appropriate ends):*

$$\begin{aligned}&\forall as \in \text{M.assocs} \cdot \forall ln \in \text{S.instances}(as) \cdot \\ &\quad \mathbf{len}(\text{M.ends}(as)) = \mathbf{len}(\text{S.ends}(ln)) \wedge \\ &\quad \forall i \in \{1, \ldots, \mathbf{len}(\text{M.ends}(as))\} \cdot \\ &\qquad \pi_i(\text{S.ends}(ln)) \in \text{S.instances}(\pi_i(\text{M.ends}(as))).\end{aligned} \tag{22}$$

3. *Instances of an association satisfy the specification of multiplicity on all association ends.*[4] *For any $n-1$ ends of n-ary association ($n \geq 2$) and $n-1$ instances of classes on those ends, the number of links they form with instances of the class on the remaining end belong to the multiplicity of this end [12, p. 40]:*

$$\begin{aligned}&\forall as \in \text{M.assocs} \cdot \\ &\quad \forall i \in \{1, \ldots, \mathbf{len}(\text{M.ends}(as))\} \cdot \forall p \in \text{product}(as, i) \cdot \\ &\qquad |\{\, ln \in \text{S.instances}(as) : \overline{\pi}_i(\text{S.ends}(ln)) = p \,\}| \in \\ &\qquad \pi_i(\text{M.mults}(as)),\end{aligned} \tag{23}$$

where:

$$\text{product}(as, i) =_{def} \bigtimes_{j=1,\ j \neq i}^{\mathbf{len}(\text{M.ends}(as))} \text{S.instances}(\pi_j(\text{M.ends}(as))). \tag{24}$$

4. *An extent of an association includes, at most, one link connecting a given set of class instances (on given link ends):*[5]

$$\begin{aligned}&\forall as \in \text{M.assocs} \cdot \forall ln_1, ln_2 \in \text{S.instances}(as) \cdot \\ &\quad ln_1 \neq ln_2 \Rightarrow \exists i \in \{1, \ldots, \mathbf{len}(\text{M.ends}(as))\} \cdot \\ &\qquad \pi_i(\text{S.ends}(ln_1)) \neq \pi_i(\text{S.ends}(ln_2)).\end{aligned} \tag{25}$$

5. *An instance of a specializing classifier is also an instance of the specialized classifier:*

$$\begin{aligned}&\forall cf_1, cf_2 \in \text{M.classifiers} \cdot cf_2 \in \text{M.specs}(cf_1) \Rightarrow \\ &\quad \text{S.instances}(cf_2) \subseteq \text{S.instances}(cf_1).\end{aligned} \tag{26}$$

[3] This issue is discussed in terms of "complete" and "incomplete" class diagrams in [5].

[4] The meaning of multiplicity for an association with more than two ends lacked precision in terms of its definition in UML prior to version 2.0. Possible interpretations are discussed in detail in [7].

[5] This condition does not have to be true for an association with a {bag} adornment. However, for the sake of simplicity, such associations are not considered here.

Definition 8 (Meaning). *Let* $\mathrm{M} \in \mathrm{Models}$ *and* $\mathcal{M}\colon \mathrm{Models} \to \mathcal{P}(\mathrm{States})$ *be the function which is defined as:*

$$\mathcal{M}(\mathrm{M}) =_{def} \{\, \mathrm{S} \in \mathrm{States} : \mathrm{Sat}(\mathrm{M}, \mathrm{S}) \,\}. \tag{27}$$

The value $\mathcal{M}(\mathrm{M})$ *refers to the* meaning of M.

5 Consequence

The mathematically defined semantics of a UML model allows for the reasoning about the properties presented in a model. The properties which are implied from the semantics of a given model, and are expressed in this model somehow implicitly, may be shown directly in the form of a different model. The relationship between two such models is defined below as a relation of semantic consequence: $\Rightarrow \subseteq \mathrm{Models} \times \mathrm{Models}$. If for a given problem domain or system the properties expressed in the model M_1 are true and it holds $\mathrm{M}_1 \Rightarrow \mathrm{M}_2$, then for the forementioned problem domain or system the properties expressed in the model M_2 are also true.

Definition 9 (Consequence). *Let* $\mathrm{M}_1, \mathrm{M}_2 \in \mathrm{Models}$*. The model* M_2 *is a* (semantic) consequence of M_1 *and can be expressed as*

$$\mathrm{M}_1 \Rightarrow \mathrm{M}_2, \textit{if and only if } \mathcal{M}(\mathrm{M}_1) \subseteq \mathcal{M}(\mathrm{M}_2). \tag{28}$$

The relation of consequence $\Rightarrow$ is transitive in the set Models $((\mathrm{M}_1 \Rightarrow \mathrm{M}_2 \wedge \mathrm{M}_2 \Rightarrow \mathrm{M}_3) \Rightarrow (\mathrm{M}_1 \Rightarrow \mathrm{M}_3))$, so to show that one diagram ia a consequence of another, it can be proved in several simpler steps. Below one of the basic reasoning rules is formally presented. Some other examples are shown in Fig. 3. Many reasoning rules, including proof of their correctness, are presented in detail in [10].

Theorem 1 (Extending multiplicity). *Let* $\mathrm{M}_1, \mathrm{M}_2 \in \mathrm{Models}$ *be such that:*

$$\begin{aligned} \mathrm{M}_1.\mathrm{classes} &= \mathrm{M}_2.\mathrm{classes}, & \mathrm{M}_1.\mathrm{mults} &\neq \mathrm{M}_2.\mathrm{mults}, \\ \mathrm{M}_1.\mathrm{assocs} &= \mathrm{M}_2.\mathrm{assocs}, & \mathrm{M}_1.\mathrm{specs} &= \mathrm{M}_2.\mathrm{specs}, \\ \mathrm{M}_1.\mathrm{ends} &= \mathrm{M}_2.\mathrm{ends}, \end{aligned} \tag{29}$$

and the models include the association $AS \in \mathrm{M}_1.\mathrm{assocs}$*, such that the multiplicity on its end* k *in the model* M_1 *is a proper subset of the multiplicity on this end in the model* M_2*:*

$$\pi_k(\mathrm{M}_1.\mathrm{mults}(AS)) \subset \pi_k(\mathrm{M}_2.\mathrm{mults}(AS)), \tag{30}$$

whereas the multiplicity specifications on the other ends of the association AS and on the ends of the other associations are the same in both models:

$$\overline{\pi}_k(\mathrm{M}_1.\mathrm{mults}(AS)) = \overline{\pi}_k(\mathrm{M}_2.\mathrm{mults}(AS)), \tag{31}$$

$$\forall as \in \mathrm{M}_1.\mathrm{assocs} \setminus \{AS\} \cdot \mathrm{M}_1.\mathrm{mults}(as) = \mathrm{M}_2.\mathrm{mults}(as). \tag{32}$$

Then the model M_2 *is a consequence of* M_1 *(see Fig. 4).*

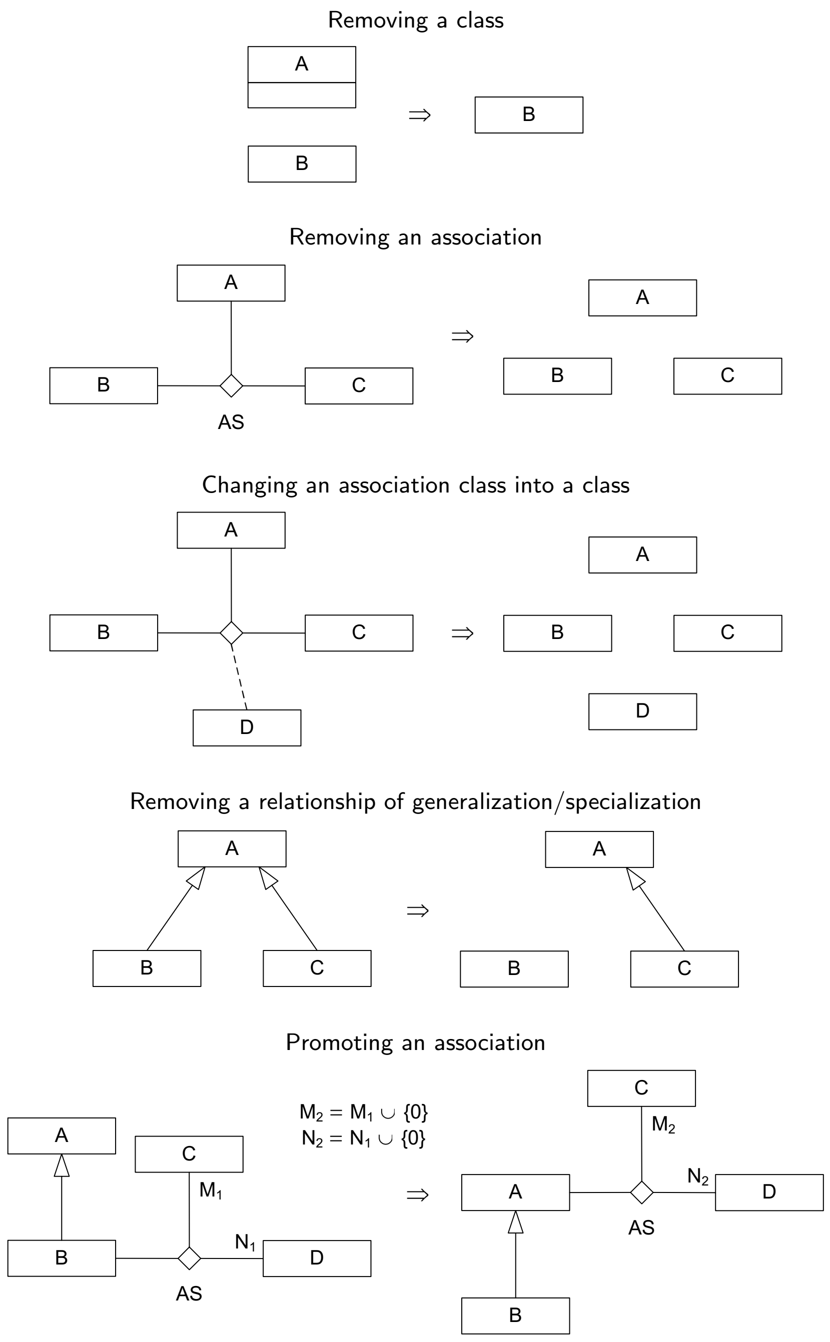


Fig. 3. Examples of a consequence relationship

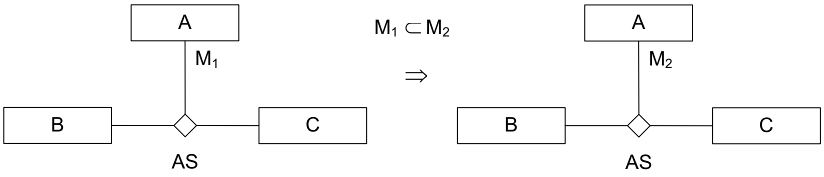


Fig. 4. Extending multiplicity

Proof. Let $\mathrm{S} \in \mathcal{M}(\mathrm{M}_1)$ (i.e. $\mathrm{Sat}(\mathrm{M}_1, \mathrm{S})$ holds). Within the framework of the proof that $\mathrm{Sat}(\mathrm{M}_2, \mathrm{S})$ holds we will show that point 3 of the definition 7 is satisfied. The satisfaction of the remaining points of this definition is implied directly from $\mathrm{Sat}(\mathrm{M}_1, \mathrm{S})$ and the condition (29).

Let $as \in \mathrm{M}_2.\mathrm{assocs}$, $i \in \{1, \ldots, \mathbf{len}(\mathrm{M}_2.\mathrm{ends}(as))\}$ and $p \in \mathrm{product}(as, i)$ (from the condition (29) the function 'product' has the same form for both models M_1 and M_2). If $as \neq AS$ or $i \neq k$, then the forementioned point is satisfied from $\mathrm{Sat}(\mathrm{M}_1, \mathrm{S})$ and the conditions (31) and (32). Otherwise, from $\mathrm{Sat}(\mathrm{M}_1, \mathrm{S})$ it holds:

$$|\{\, ln \in \mathrm{S}.\mathrm{instances}(AS) : \overline{\pi}_k(\mathrm{S}.\mathrm{ends}(ln)) = p \,\}| \in \pi_k(\mathrm{M}_1.\mathrm{mults}(AS)) \qquad (33)$$

and from the condition (30):

$$|\{\, ln \in \mathrm{S}.\mathrm{instances}(AS) : \overline{\pi}_k(\mathrm{S}.\mathrm{ends}(ln)) = p \,\}| \in \pi_k(\mathrm{M}_2.\mathrm{mults}(AS)). \qquad (34)$$

5.1 Refinement

A refinement is a relationship that represents a fuller specification of something that has already been specified at a certain level of detail or at a different semantic level [9]. Fig. 5 shows seven simple models of the same problem domain but at different levels of detail (at different stages of development). Each consecutive model includes some more details about the modelled domain and thus can be treated as a refinement of any of the previous models. At the same time, if a given model is a refinement of another then they both are related by a consequence relationship, as it is shown in the figure. Generally, if $\mathrm{M}_1 \Rightarrow \mathrm{M}_2$ holds then the model M_1 is at least as detailed (complete or precise) description of a given problem domain or system as the model M_2.

6 Equivalence

Two models with exactly the same meaning are a specific case of the relation of consequence. Such cases are defined below as a relation of semantical equivalence: $\Leftrightarrow\ \subseteq \mathrm{Models} \times \mathrm{Models}$. If $\mathrm{M}_1 \Leftrightarrow \mathrm{M}_2$ holds, then the models M_1 and M_2 are completely interchangeable descriptions of a given problem domain or system (or their parts).

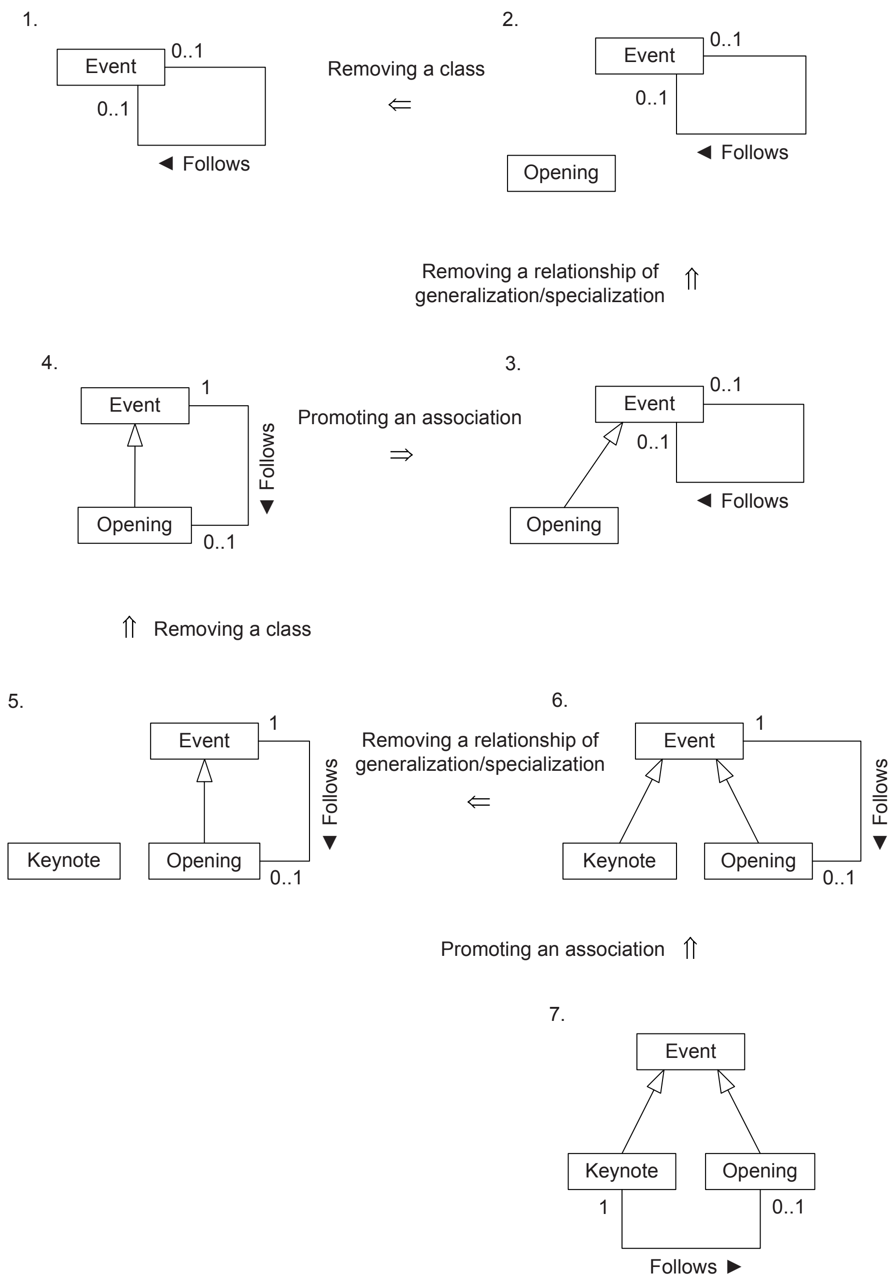


Fig. 5. Refinement vs. consequence relationship

Definition 10 (Equivalence). *Let* $\mathrm{M}_1, \mathrm{M}_2 \in \mathrm{Models}$*. The model* M_1 *is* (semantically) equivalent to M_2 *and can be expressed as*

$$\mathrm{M}_1 \Leftrightarrow \mathrm{M}_2, \text{ if and only if } \mathrm{M}_1 \Rightarrow \mathrm{M}_2 \wedge \mathrm{M}_2 \Rightarrow \mathrm{M}_1. \tag{35}$$

6.1 An Example of Equivalence

For the specialization of an association, the UML metamodel [12] defines only two syntactical constraints (see the definition 2): an association specializing another association has the same number of ends and its ends are connected to the same classifiers as in a specialized association or to their specializations. In fact, these constraints partially reflect the semantics of a specializing association, which instances are the specific cases of instances of a specialized association. Between two such associations, however, other dependencies which have not been taken into account in the UML metamodel and which can be shown using our formalization also exist. Below we present one example of such dependencies.

Theorem 2 (Association Specialization vs. Multiplicity). *Let* $\mathrm{M}_1, \mathrm{M}_2 \in \mathrm{Models}$ *be such that:*

$$\begin{aligned} \mathrm{M}_1.\mathrm{classes} &= \mathrm{M}_2.\mathrm{classes}, & \mathrm{M}_1.\mathrm{mults} &\neq \mathrm{M}_2.\mathrm{mults}, \\ \mathrm{M}_1.\mathrm{assocs} &= \mathrm{M}_2.\mathrm{assocs}, & \mathrm{M}_1.\mathrm{specs} &= \mathrm{M}_2.\mathrm{specs}, \\ \mathrm{M}_1.\mathrm{ends} &= \mathrm{M}_2.\mathrm{ends}, \end{aligned} \tag{36}$$

and the models include the associations $AA, AB \in \mathrm{M}_1.\mathrm{assocs}$*, such that* AB *is a specialization of* AA*:*

$$AB \in \mathrm{M}_1.\mathrm{specs}(AA), \tag{37}$$

and for the multiplicity on the end k *of the association* AB *the below condition holds:*[6]

$$\begin{aligned} &\pi_k(\mathrm{M}_2.\mathrm{mults}(AB)) = \\ &\quad \pi_k(\mathrm{M}_1.\mathrm{mults}(AB)) \cap \{0, \ldots, \max(\pi_k(\mathrm{M}_1.\mathrm{mults}(AA)))\}, \end{aligned} \tag{38}$$

whereas the multiplicity specifications on the other ends of the association AB *and on the ends of the other associations are the same in both models:*

$$\overline{\pi}_k(\mathrm{M}_1.\mathrm{mults}(AB)) = \overline{\pi}_k(\mathrm{M}_2.\mathrm{mults}(AB)), \tag{39}$$

$$\forall as \in \mathrm{M}_1.\mathrm{assocs} \setminus \{AB\} \cdot \mathrm{M}_1.\mathrm{mults}(as) = \mathrm{M}_2.\mathrm{mults}(as). \tag{40}$$

Then the model M_1 *is equivalent to* M_2 *(see Fig. 6).*

Proof. Firstly, we will prove that $\mathrm{M}_1 \Rightarrow \mathrm{M}_2$ and then $\mathrm{M}_2 \Rightarrow \mathrm{M}_1$ hold.

$\mathbf{M_1 \Rightarrow M_2}$**.** Let $\mathrm{S} \in \mathcal{M}(\mathrm{M}_1)$. Within the framework of the proof that $\mathrm{Sat}(\mathrm{M}_2, \mathrm{S})$ holds we will show that point 3 of the definition 7 is satisfied. The satisfaction of the remaining points of this definition is implied directly from $\mathrm{Sat}(\mathrm{M}_1, \mathrm{S})$ and the condition (36).

Let $as \in \mathrm{M}_2.\mathrm{assocs}$, $i \in \{1, \ldots, \mathbf{len}(\mathrm{M}_2.\mathrm{ends}(as))\}$ and $p \in \mathrm{product}(as, i)$ (from the condition (36) the function 'product' has the same form for both

[6] If X is an infinite subset of $\mathbb{N}$, then we assume $\{0, \ldots, \max(X)\} =_{def} \mathbb{N}$.

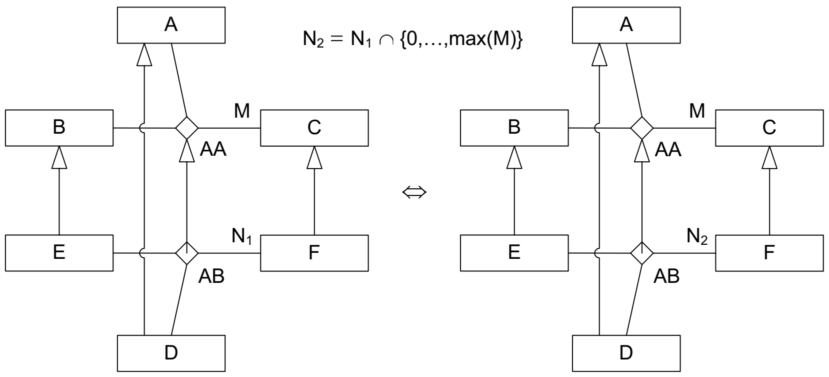


Fig. 6. Association specialization vs. multiplicity

models M_1 and M_2). If $as \neq AB$ or $i \neq k$, then the mentioned point is satisfied from $\mathrm{Sat}(\mathrm{M}_1, \mathrm{S})$ and the conditions (39) and (40). Otherwise, if we use the symbols:

$$\alpha_{AA}(i,p) =_{def} |\{\, \mathit{ln} \in \mathrm{S.instances}(AA) : \overline{\pi}_i(\mathrm{S.ends}(\mathit{ln})) = p\,\}| \text{ and} \tag{41}$$

$$\alpha_{AB}(i,p) =_{def} |\{\, \mathit{ln} \in \mathrm{S.instances}(AB) : \overline{\pi}_i(\mathrm{S.ends}(\mathit{ln})) = p\,\}|, \tag{42}$$

it remains to be shown that the below holds:

$$\alpha_{AB}(k,p) \in \pi_k(\mathrm{M}_2.\mathrm{mults}(AB)). \tag{*}$$

Because $\mathrm{Sat}(\mathrm{M}_1, \mathrm{S})$ holds, therefore:

$$\alpha_{AA}(k,p) \in \pi_k(\mathrm{M}_1.\mathrm{mults}(AA)) \text{ and} \tag{43}$$

$$\alpha_{AB}(k,p) \in \pi_k(\mathrm{M}_1.\mathrm{mults}(AB)), \tag{44}$$

and from condition (37) and from point 5 of the definition 7:

$$\mathrm{S.instances}(AB) \subseteq \mathrm{S.instances}(AA). \tag{45}$$

Then the following inequality holds:

$$\alpha_{AB}(k,p) \leq \alpha_{AA}(k,p) \tag{46}$$

and from the equation (43):

$$\alpha_{AB}(k,p) \in \{0, \ldots, \max(\pi_k(\mathrm{M}_1.\mathrm{mults}(AA)))\}. \tag{47}$$

From the equation (44):

$$\alpha_{AB}(k,p) \in \pi_k(\mathrm{M}_1.\mathrm{mults}(AB)) \cap \{0, \ldots, \max(\pi_k(\mathrm{M}_1.\mathrm{mults}(AA)))\} \tag{48}$$

and the property (*) holds on the assumption (38).

$\mathbf{M_2} \Rightarrow \mathbf{M_1}$. From the assumptions of our theorem:

$$\pi_k(\mathrm{M}_2.\mathrm{mults}(AB)) \subseteq \pi_k(\mathrm{M}_1.\mathrm{mults}(AB)). \tag{49}$$

If the above sets are equal, then $\mathrm{M}_1 = \mathrm{M}_2$. Otherwise, the assumptions of theorem 1 are satisfied.

Example 1 (Imprecise Multiplicity Specification). Fig. 7 illustrates a situation at a hypothetical scientific conference, where participants can be the authors (or co-authors) of no more than two papers submitted to the conference. If the submitted paper is accepted for the presentation during the conference, it is presented by one of its authors. By virtue of theorem 2, one author cannot have more than two presentations.

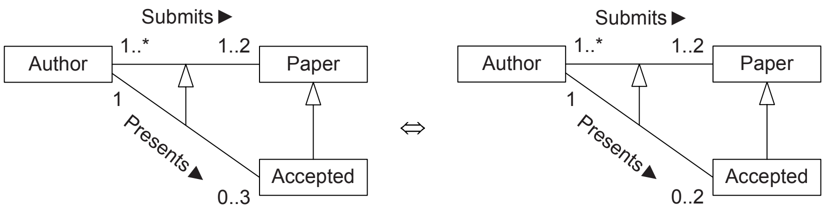


Fig. 7. Imprecise multiplicity specification

7 Conclusion

Theoretical research work in the area of UML formalization, although rather difficult to be applied directly in software engineering practice, can be useful in facilitating a better understanding of UML modeling concepts and can contribute to improving the UML specification itself. In the paper we have proposed a concise formalization of basic UML static models and have shown they can be helpful in formal reasoning. The presented formalization can easily include other elements of UML static models, which have not been addressed here, for example an aggregation, a composition or abstract classifiers [10]. It was also used to define the problem of the semantic consistency of individual models [11].

References

1. Aredo, D., Traoré, I., Stølen, K.: Towards a Formalization of UML Class Structure in PVS. Research Report 272, Department of Informatics, University of Oslo (1999)
2. Berardi, D., Calì, A., Calvanese, D., De Giacomo, G.: Reasoning on UML Class Diagrams. Technical Report, Dipartimento di Informatica e Sistemistica, Università di Roma (2003)
3. Egyed, A.: Automated Abstraction of Class Diagrams. ACM Transactions on Software Engineering and Methodology 11(4), 449–491 (2002)

4. Evans, A.: Reasoning with UML Class Diagrams. In: Second IEEE Workshop on Industrial Strength Formal Specification Techniques (WIFT 1998) (1998)
5. France, R.: A Problem-Oriented Analysis of Basic UML Static Requirements Modeling Concepts. In: Proceedings of OOPSLA 1999, pp. 57–69 (1999)
6. Funes, A., George, C.: Formalizing UML Class Diagrams. In: Favre, L. (ed.) UML and the Unified Process, pp. 129–198. Idea Group Publishing (2003)
7. Génova, G., Llorens, J., Martínez, P.: The Meaning of Multiplicity of N-ary Association in UML. Software and Systems Modeling 2(2), 86–97 (2002)
8. Gogolla, M., Richters, M.: Equivalence Rules for UML Class Diagrams. In: UML 1998 - Beyond the Notation, First International Workshop, pp. 87–96 (1998)
9. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley, Reading (2004)
10. Szlenk, M.: Formal Semantics and Reasoning about UML Conceptual Class Diagram (in Polish). PhD Thesis, Warsaw University of Technology (2005)
11. Szlenk, M.: Formal Semantics and Reasoning about UML Class Diagram. In: Proceedings of DepCoS-RELCOMEX 2006, pp. 51–59 (2006)
12. UML 2.1.1 Superstructure Specification (formal/2007-02-05). Object Management Group (2007)

Does Test-Driven Development Improve the Program Code? Alarming Results from a Comparative Case Study

Maria Siniaalto[1] and Pekka Abrahamsson[2]

[1] F-Secure Oyj,
Elektroniikkatie 3, FIN-90570 Oulu, Finland
Maria.Siniaalto@f-secure.com
[2] VTT Technical Research Centre of Finland,
P.O. Box 1100, FIN-90571 Oulu, Finland
Pekka.Abrahamsson@vtt.fi

Abstract. It is suggested that test-driven development (TDD) is one of the most fundamental practices in agile software development, which produces loosely coupled and highly cohesive code. However, how the TDD impacts on the structure of the program code have not been widely studied. This paper presents the results from a comparative case study of five small scale software development projects where the effect of TDD on program design was studied using both traditional and package level metrics. The empirical results reveal that an unwanted side effect can be that some parts of the code may deteriorate. In addition, the differences in the program code, between TDD and the iterative test-last development, were not as clear as expected. This raises the question as to whether the possible benefits of TDD are greater than the possible downsides. Moreover, it additionally questions whether the same benefits could be achieved just by emphasizing unit-level testing activities.

Keywords: Test-Driven Development, Test-first Programming, Test-first Development, Agile Software Development, Software Quality.

1 Introduction

Test-driven development (TDD) is one of the core elements of Extreme Programming (XP) method [1]. The use of the TDD is said to yield several benefits. It is claimed to improve test coverage [2] and to produce loosely coupled and highly cohesive systems [3]. It is also believed to encourage the implementation scope to be more explicit [3] and to enable more frequent integration [4]. On the other hand, it is claimed that rapid changes may cause expensive breakage in tests and that the lack of application or testing skills may produce inadequate test coverage [5]. TDD has also received criticism over not being very suitable for systems such as multithreaded applications or security software, since it cannot mechanically demonstrate that their goals have been met [6]. However, the scientific empirical evidence behind all of these claims is currently sparse, and thus it is difficult to draw meaningful conclusions. The studies dealing with TDD have mainly focused on developer productivity and external code quality, whereas the TDD's impacts on program code have received less attention. The existing empirical evidence supports the claim that TDD yields improved external quality (see a recent summary of the TDD studies in [7]). However, it is not clear what has been the baseline for the comparison in those studies, e.g. did any unit level tests exist previously? The results of the studies, which address TDD's design impact, are presented in Section 2 in more detail.

Despite the lack of solid empirical evidence, both the industry and academia are keenly adopting test-driven development approaches. The purpose of this study is to investigate whether and how the structure of the program code changes or improves with the use of TDD. Five semi-industrial software development projects, containing both students and professionals as research subjects, were involved in the comparison of TDD and iterative test-last (ITL) approaches. Two of the projects used iterative test-last (ITL) development technique and three utilized TDD. The metrics used for evaluating the code are the traditional and widely-used suite of Chidamber and Kemerer (CK-metrics) [8] strengthened with McCabe's cyclomatic complexity metric [9]. To obtain a balance, the dependency management metrics proposed by Martin [10] for studying the code's package structure, were chosen as well.

The results of this study partially contradict the current literature. In particular, the case empirical evidence shows that TDD does not improve all the areas of the program design as expected. The results imply that TDD may produce less complex code, but on the other hand, the package structure may become more difficult to change and maintain.

The remainder of the paper is organized as follows. In the following section, the related work will be introduced. This is followed by an introduction to the metrics used to study the impact of TDD on program structure. Section 4 presents the empirical results, outlines the research design used to complete the study in a scientifically valid manner as well as detailing the threats to the validity of the empirical results. Section 5 discusses the novelty of the results in the light of existing studies and identifies the implications of the presented results. The conclusion section summarizes the principal results and proposes the potential future research avenues.

2 Related Work

In this section, the existing empirical evidence on the TDD's impact on the program design is presented. A total of five studies is included.

Janzen and Saiedian [11] compared the TDD and the ITL approaches using students as research subjects. They calculated several structural and object-oriented metrics in order to evaluate the differences in the internal quality of the software. As most of these results were within acceptable limits, there were some concerns regarding the complexity and coupling in the TDD code.

Kaufmann and Janzen [12] conducted a controlled experiment with students as research subjects comparing the design quality attributes of the TDD and the test-last approaches (whether the test-last was used iteratively is not known). The design quality was assessed with several structural and object-oriented metrics. They did not find any differences in the code complexities, but they report that there were indications that the design quality of the TDD code was superior. However, they also admit that this finding may be due to the better programming skills of the subjects applying the TDD.

Steinberg [13] reports on the findings of the use of unit testing in the TDD style in an XP study group. Although, Steinberg concentrates on discussing the results from an educational point of view, his study also provides concrete experiences about the effects of the novel use of TDD and is thereby included in this study. He notices that the students tended to write more cohesive code when using TDD and the coupling was looser, since the objects had more clearly defined responsibilities. Evidence to support these last two claims was not provided, however.

In our initial study [7], we explored the effect of TDD on program design in semi-industrial setting comparing two ITL and one TDD projects. The design impact was evaluated using traditional object-oriented metrics. The initial results indicated that TDD does not always produce highly cohesive code. However, we concluded that the cohesion results might have been affected by the fact that all the developers in the TDD project were less experienced when compared to the subjects in the ITL projects.

Müller [14] studied the effect of test-driven development on program code. He included five TDD software systems of which three were student projects and compared them with three open source-based conventional software systems. He assessed the impact of TDD on the resulting code with Chidamber and Kemerer's [8] object-oriented metric suite and his own newly developed metric called assignment controllability. Müller reports that CK-metrics did not show any impact on the use of TDD but that the new assignment controllability metric showed a difference i.e. the number of methods where all assignments are completely controllable is higher for systems developed by TDD.

3 Metrics to Study Changes in Program Structure

The demand for quality software has resulted in a large set of different metrics some of which have been validated and some have not. Many of these metrics have been a subject of criticism and their empirical validity has been questioned. There is an on-going debate on which metrics are the best indicators of the software quality and whether some particular metric even maps to the quality attribute it is supposed to represent. However, in many cases the authors presenting the criticism have not been able to propose a metric that would have solved the problem and would thereby have been widely adopted. Due to these reasons, we wish to emphasize that the aim of this study is not to validate or comment on the validity of any particular metric. The aim is to study whether and how TDD affects the code and its structure, and therefore both traditional and novel metrics were chosen for this study.

3.1 Traditional Metrics

The object-oriented metric suite, proposed by Chidamber and Kemerer [8], and McCabe's Cyclomatic complexity [9], were chosen as traditional representatives since they have been and still are widely used. CK-metrics, validated in [15], measure the different aspects of object-oriented construct. The suite contains six individual metrics: weighted methods per class (WMC), depth of inheritance tree (DIT), number of children (NOC), coupling between objects (CBO), response for a class (RFC) and lack of cohesion in methods (LCOM). The suite was strengthened with Henderson-Sellers's lack of cohesion (LCOM*) [16. McCabe's' cyclomatic complexity measures the number of independent paths through a program module, and it is proposed to profile system's testability and maintainability. Although it is one of the most used and accepted of the static software metrics, it has also received criticism e.g. [17].

WMC measures the number of methods in a class and it is proposed to predict how much time and effort is required to maintain the class. DIT measures the depth of each class within its hierarchy, and its result shows how many ancestor classes can potentially affect this class. NOC presents the number of subclasses for each class. CBO presents the number of classes to which the class is coupled and it can be used as an indicator of whether the class hierarchy is losing its integrity. RFC presents the number of methods that can be executed in response to a message to the class. It is proposed as an indicator of the complexity and testing effort. LCOM assesses the similarity of the class methods by comparing their instance variable use pairwise. It is proposed to identify classes that are likely to behave in a less predictable way, because they are trying to achieve many different objectives. The biggest flaws of LCOM are that it indicates a lack of cohesion only when fewer than half of the paired methods use the same instance variables and the fact that a zero value does not necessarily indicate good cohesion, though a large value suggests poor cohesion [15, 16]. LCOM* measures the correlation between the methods and the local instance variables of a class. A low value of LCOM* indicates high cohesion and a well-designed class. A cohesive class tends to provide a high degree of encapsulation.

3.2 Dependency Management Metrics

The dependency management metrics proposed by Martin [10] measure and characterize the dependency structure of the packages. The suite includes: afferent coupling (C_a), efferent coupling (C_e), instability (I), abstractness (A) and normalized distance from the main sequence (D'). C_a counts the number of classes outside the package that depend on the classes inside the package whiles C_e counts the number of classes inside the package that depend on the classes outside the package. These two values are used when the instability (I) of the package is assessed. It has a specific range [0,1] with value 0 indicating a maximal stability and value 1 indicating maximal instability, i.e. no other package depends on this package. A package with lots of incoming dependencies is regarded as stable, because it requires a lot of work to reconcile the possible changes with all the dependent packages. The abstractness is simply measured by the ratio of abstract classes in a package to the total number of classes in a package.

Martin proposes that the package should be as abstract as it is stable so that the stability does not prevent the class from being extended. The main sequence presents this ideal ratio of stability and abstractness. Fig. 1 presents the main sequence and the zones of exclusion around (0,0) and (1,1). Packages that fall into the zone of pain are very difficult to change because they are extremely stable and cannot be extended since they are not abstract. The zone of uselessness contains packages that are abstract enough but useless since they have no dependents. The packages that remain near the main sequence are considered to balance their abstractness and instability well. D’ has the range [0,1] and it indicates how far a package is from this main sequence. Value 0 indicates that the package is directly on the main sequence whereas value 1 indicates that it is as far away as possible.

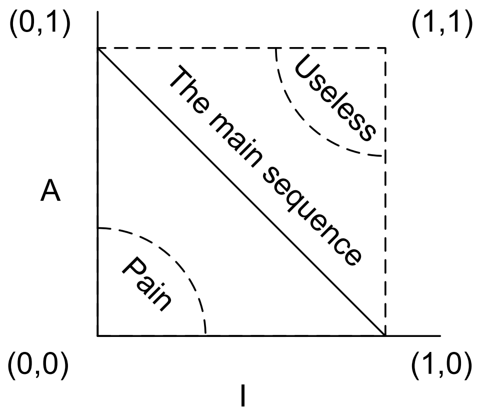


Fig. 1. Distance from the main sequence [10]

4 Empirical Results from a Comparative Case Study

The comparative empirical evaluation of TDD in five small scale software development case studies aims at exploring the effects of TDD on program codes. The layout of the research design for the study is presented first and is followed by the empirical results. The threats to validity are then identified and subsequently addressed.

4.1 Research Design

The research method for the three case projects is the controlled case study approach [18], which combines aspects of experiments, case studies and action research. It is especially designed for studying agile methodologies, and it involves conducting a project with a business priority of delivering a functioning end product to a customer, in close-to-industry settings. At the same time, the measurement data is collected for rapid feedback, process improvement and research purposes. The development is performed in controlled settings and may involve both students and professionals as developers.

All the case projects had the aim of delivery of a concrete software product to a real customer. Two of the projects used ITL development and three utilized TDD: Every project team worked in a shared co-located development environment during the project. The projects were not simultaneous. All the projects continued for nine weeks and followed an agile software development method, Mobile-D™ [19], which provided a coherent framework for this study to compare ITL and TDD. Mobile-D™ is an agile method, which is empirically composed over a series of software development projects in 2003-2006. The method is based on two-month production rhythm, which is divided in five sub phases. Each of the sub phases takes from one to two weeks in calendar time. These phases are called set-up, core functionality one, core functionality two, stabilize and wrap-up & release. Mobile-D™ adopts most of the Extreme Programming practices, Scrum management practices and RUP phases for life-cycle coverage. The method is described in pattern-format and can be downloaded from `http://agile.vtt.fi`. The code development took place in controlled settings using the same Mobile-D™ practices in all the projects. The only difference was that projects 1 and 2 used ITL and projects 3, 4 and 5 used TDD. The implementations were realized with Java programming language. The difficulty of implementation was at the same level in all the projects, as they all were quite simple systems whose main functionalities were to enable data storing and retrieving.

Table 1 provides a summary of the parts of the projects of which they are not convergent to each other.

Table 1. Summary of the case projects

	Case 1	Case 2	Case 3	Case 4	Case 5
# of developers	4	5	4	2	2
Developer type[1]	S	S	S	P	S
Dev. technique	ITL	ITL	TDD	TDD	TDD
Iterations	6	6	6	4	4
Product type	Intranet	Mobile	Intranet	Internet	Intranet
Total Product size (LOC)	7700	7000	5800	5000	8900 (3100 new)

The development teams of the projects 1, 2, 3 and 5 consisted of 5-6th year Master's students. All the team members in projects 1 and 2, which used ITL, had some industrial coding experience, while only one of the developers in project 3 and none of the developers in project 5, which both used TDD, had previously worked in industrial settings. However, all the subjects in project 3 and 5 were either Software Engineering graduates or had a personal interest in programming. The team members in case project 4, which also used TDD, were professional, experienced developers whose normal daily work includes teaching and development assignments in academic settings. The developers were told and encouraged to write tests in all the projects regardless of the development technique used. In addition, in projects 3, 4 and 5 the use of TDD was stated as mandatory. Intranet applications were implemented in projects 1, 3 and 5: in project 1 for managing research data and in projects 3 and 5 for project management purposes. Case project 5 was a follow-up of case project 3. All the systems consisted of server side and graphical user interface. A stock market browsing system to be used via mobile device was implemented in project 2. The biggest part concentrated on the server side and the mobile part mainly handled connecting to the server and presenting the retrieved data. Internet application for information storing was realized in project 4. The implementation contained a server side and a graphical user interface. However, to make the comparison of TDD and

[1] S= Student, P= Professional.

ITL even, graphical user interfaces and the mobile client application part in project 2 were excluded from the evaluation.

4.2 Results

The results of the traditional and dependency management metrics are presented in the following subsections. These results are discussed in more detail in section 5.

4.2.1 Traditional Metrics

The results of the traditional metrics are presented in Appendix 1a and 1b. The significance of the differences between TDD and ITL were evaluated using the Mann-Whitney U-test (Table 2). WMC, RFC and McCabe's cyclomatic complexity were used to assess the complexity of the code in this study. The WMC values do not differ significantly between the development approaches while the RFC values seem to be lower with TDD. The U-test confirms this distinction as statistically significant ($p<0.05$). The McCabe's cyclomatic complexity results are also lower with TDD and are supported by statistical analysis as well.

The inheritance was studied using DIT and NOC. The DIT values are higher in cases in which TDD was used. This difference is statistically significant. The results of NOC cannot make any difference between the development methods and they are quite surprising, as there are only a few outliners for each case.

The coupling was measured using CBO metric. The results do not differ significantly between the development methods used, and the values are fairly low in all the cases.

The LCOM of CK-suite and LCOM* by Henderson-Sellers were used to find out the cohesion characteristics of the code. The original LCOM does not reveal any differences whereas the new LCOM* seem to be higher in TDD cases, though the difference is not statistically significant.

Table 2. Mann-Whitney p-values for the results of the traditional metrics[2]

	WMC	DIT	NOC	RFC	CBO	LCOM	Cyclomatic Complexity	LCOM*
p	0,389	**0,001**	0,396	**0,018**	0,678	0,535	**0.000**	0,061

4.2.2 Dependency Management Metrics

The measures of the dependency management metrics are presented in Appendix 2. The AC value is much higher in the TDD cases 3 and 5 meaning that there are more classes outside the package that depend on the classes inside the package. Case 4 presents an exception to that, and therefore it cannot be concluded that TDD would produce code with high afferent coupling. The EC values of all TDD cases are significantly lower than the corresponding ITL cases, meaning that in TDD cases there are fewer classes inside the package that depend on the classes outside the package. The instability results show that the package structure of TDD code is more stable than the ITL code. This means that the ITL packages are less dependent on other

[2] The values where the difference is statistically significant are presented in bold.

packages. The measure of abstractness gives only slight indications that TDD may produce packages with a higher level of abstraction. The results of the normalized distance from the main sequence clearly differ between the ITL and TDD cases. Fig. 2 presents the scatter plot of the packages case-wise. The packages in the TDD code seem to come closer to the zone of pain meaning that they are more stable but yet not abstract. However, it should be noticed that the number of packages is much smaller in the ITL cases and that can obviously bias these results.

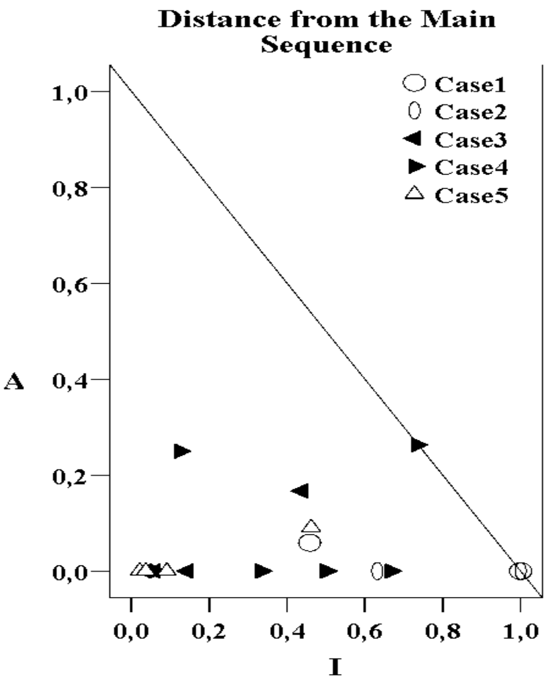


Fig. 2. The distance from the main sequence

4.3 Threats to Validity

The interpretation of results is always more complex when students are used as study subjects. However, the development environment was explicitly designed to relate closely to that of an industrial development setting with strict time-to-market pressures, regular working hours and, significantly, the developers were implementing a real product to be delivered to a real customer. Höst [20] and Runeson [21] suggest that students may provide an adequate model of professionals as similar improvement trends may be identified between both groups. In addition, in industry, teams usually have a mixed set of experiences and skills. The teams in the case projects included in this study represented a similar mix. In addition one TDD team consisted of professional developers only in this study.

The differences between product types and concepts place a threat to internal validity of this study. We excluded graphical user interfaces and the mobile client application part in case 2 to keep the comparison of the projects even. We also proceed in the belief that all product implementations present a similar level of difficulty, as the basic functionality in all the systems was simple data storage and retrieval, which was realized using a model-view-controller structure. The fact, that project 5 is a follow-up of project 3, can somewhat bias the results. However, in project 5, the implementation concentrated on totally new functionalities and the amount of new code is significant in proportion to the total size of the code.

To control the subjects' conformance to implement the tests and to use TDD correctly a person responsible for testing was appointed in all the projects. That person's responsibilities included monitoring the testing implementation. The developers were not aware that the program design was to be compared as the measurements were compiled after the project endings, which reduces possible observation effects. Another limitation relates to the size of the software product as well as the distribution of the project work. All the case projects had less than 10 000 lines of code, their development took around 1000 person hours and a single team in one location developed the products. The impact of TDD on program design, however, should be visible from the very start and thereby present in all of the studied projects.

5 Discussion

The traditional metrics indicated statistically significant differences in DIT, RFC and cyclomatic complexity. These findings partly contradict the findings of Müller [14] and Janzen and Saiedian [11]: Müller reports that in his study, none of the CK-metrics showed differences between the development approaches used, whereas Janzen and Saiedian noticed that the cyclomatic complexity was worse with TDD. Even though DIT was increased with TDD and the difference statistically significant, it should be noted that the level of inheritance was very low in all the cases included in this study regardless of the development approach used. Therefore, it is too early to draw conclusions that TDD encourages to greater use of inheritance. In addition, the target programs were quite small in all the cases resulting in a limited code base, which may be one reason for the low inheritance.

Both, the RFC and cyclomatic complexity, were lower with TDD which may indicate that TDD helps produce less complex code. In this context, it should be noted that the corresponding values for ITL cases were not poor- TDD values just were slightly better. Other traditional metrics did not reveal statistically significant differences. Although the medians of LCOM* results were higher, the statistical significance of this difference is not high according to the U-test.

The results of the dependency management metrics indicate that TDD may cause the software packages to become more stable. The results imply that TDD produces fewer classes inside the package that depend on the classes outside the package. This affects the instability result meaning that TDD produces more packages that are not dependent on other packages but have many dependents. It can be argued that this makes them more difficult to alter *a posteriori*. On the other hand, the high dependency on other packages and the lack of dependents is not desirable either, because it could cause the packages to change more easily. The measure of abstractness gives only slight indications that TDD may produce packages with a higher level of abstraction, although the difference is not significant. The normalized distance from the main sequence, which measures the ratio of instability and abstractness, differed clearly from the proposed ideal ratio, as it indicated that the TDD packages are too stable in proportion to their abstractness. Both these findings lead us to conclude that the package structure of the code produced with TDD may be difficult to change and maintain, because it is likely to be concrete and have many dependents. This finding contradicts the claims in the literature. On the other hand, the number of packages was clearly higher in the cases in which TDD was used, and is likely to affect the results. i.e. in the ITL cases, there were only two packages in both, while in the TDD cases the corresponding values were 4, 8 and 4. Case 5 is based on the "legacy" code of the case 3, and this is probably one reason for the similarity between the results of these two cases. However, these findings indicate that TDD may result in a greater number of packages that are very concrete in relation to their stability. The fact that the results were similar in all the TDD cases regardless of the professionalism and developers' experience is also significant.

6 Conclusion

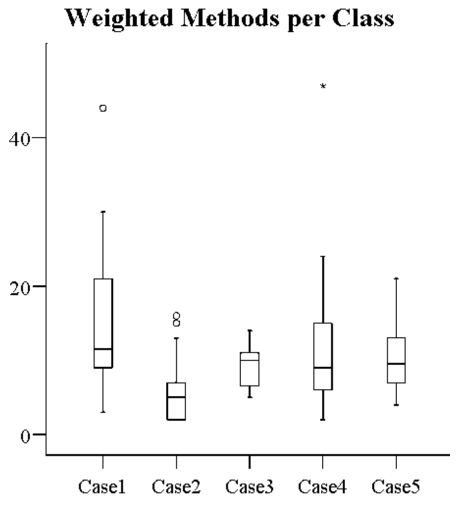
Test-driven development is claimed to be one of the most important practices of agile development, and to address many problems at once. The current empirical research has mainly focused on exploring the external quality effects of TDD. Despite the fact that very little is known about its internal quality effects, academia and industry are eagerly adopting the practice. This study aims at contributing to the empirical body of knowledge by examining the effect of TDD on program design.

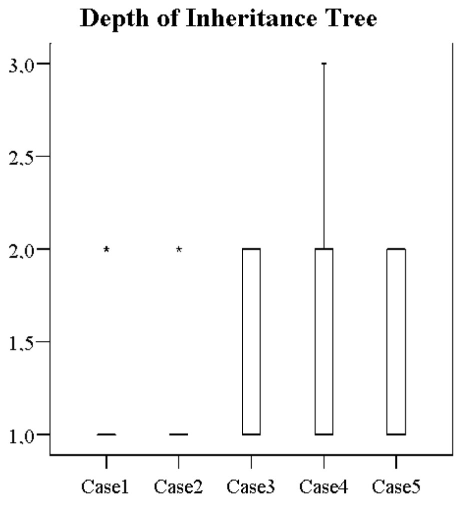
We studied the effect of TDD in five different software projects with students and professionals as research subjects. The results provide some warning that the benefits of TDD are not automatic and as self-evident as expected. Some of the findings imply that TDD may produce a less complex code while other findings indicate the opposite as there are indications that TDD may produce package structures that are more difficult to change. The existing empirical evidence supports the claim that TDD yields improve external quality, especially when employed in an industrial context. This finding clearly conflicts with the case study which identifies certain risks in the adoption of TDD. Therefore, the present authors query whether the reported external quality benefits can be achieved with a more traditional approach to unit-level testing or whether they are really due to TDD itself. We intend to use the results of this study as a baseline for further empirical studies, with experienced developers employing TDD in industrial settings. Our aim is to increase the understanding of test-driven development in different real-life development settings and thereby contribute to the growing body of evidence in the area of agile software development in general and test-driven development in particular. We maintain that whether TDD ultimately improves program design, remains to be answered.

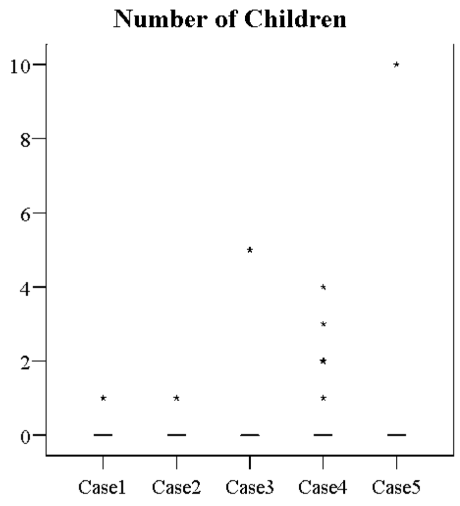
References

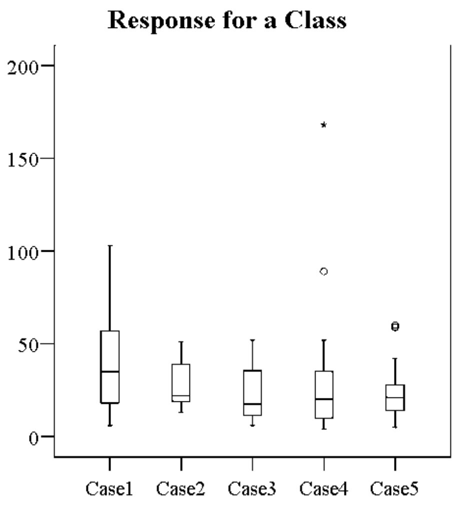
1. Beck, K.: Extreme Programming Explained, 2nd edn. Embrace Change. Addison-Wesley, Boston (2004)
2. Astels, D.: Test-Driven Development: A Practical Guide. Prentice Hall, Upper Saddle River (2003)
3. Beck, K.: Aim, fire. IEEE Software 18(5), 87–89 (2001)
4. Beck, K.: Test-Driven Development By Example. Addison-Wesley, Boston (2003)
5. Boehm, B., Turner, R.: Balancing Agility and Discipline - A Guide for the Perplexed. Addison-Wesley, Reading (2004)
6. Stephens, M., Rosenberg, D.: Extreme Programming Refactored: The Case Against XP. Apress, Berkeley (2003)
7. Siniaalto, M., Abrahamsson, P.: A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage. In: First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), pp. 275–284. IEEE Press, New York (2007)
8. Chidamber, S.R., Kemerer, C.F.: A metrics Suite for Object Oriented Design. IEEE Trans.Software Eng. 20(6), 476–493 (1994)
9. McCabe, T.J.: A Complexity Measure. IEEE Trans.Software Eng. 2(4), 308–320 (1976)
10. Martin, R.C.: Agile Software Development: Principles, Patterns, and Practices. Pearson Education, Upper Saddle River (2003)
11. Janzen, D.S., Saiedian, H.: On the Influence of Test-Driven Development on Software Design. In: 19th Conference on Software Engineering Education and Training (CSEET 2006), pp. 141–148. IEEE Press, New York (2006)
12. Kaufmann, R., Janzen, D.: Implications of Test-Driven Development A Pilot Study. In: 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2003), pp. 298–299. ACM, New York (2003)
13. Steinberg, D.H.: The effect of unit tests on entry points, coupling and cohesion in an introductory Java programming course. XP Universe (2001)
14. Müller, M.M.: The Effect of Test-Driven Development on Program Code. In: Abrahamsson, P., Marchesi, M., Succi, G. (eds.) XP 2006. LNCS, vol. 4044, pp. 94–103. Springer, Heidelberg (2006)
15. Basili, V.R., Melo, W.L.: A validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Trans.Software Eng. 22(10), 751–761 (1996)
16. Henderson-Sellers, B.: Object-Oriented Metrics: Measures of Complexity. Prentice Hall, Upper Saddle River (1996)
17. Shepperd, M.: A critique of cyclomatic complexity as a softwaremetric. Software Engineering Journal (1988)
18. Salo, O., Abrahamsson, P.: Empirical Evaluation of Agile Software Development: The Controlled Case Study Approach. In: Bomarius, F., Iida, H. (eds.) PROFES 2004. LNCS, vol. 3009, pp. 408–423. Springer, Heidelberg (2004)
19. Ihme, T., Abrahamsson, P.: Agile Architecting: The Use of Architectural Patterns in Mobile Java Applications. International Journal of Agile Manufacturing 8(2), 97–112 (2005)
20. Höst, M., Regnell, B., Wohlin, C.: Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. Empirical Software Engineering 5(3), 201–214 (2000)
21. Runeson, P.: Using students as Experiment Subjects - An Analysis of Graduate and Freshmen Student Data. In: Empirical Assessment in Software Engineering (EASE 2003) (2003)

Appendix 1a: The Results of Traditional Metrics

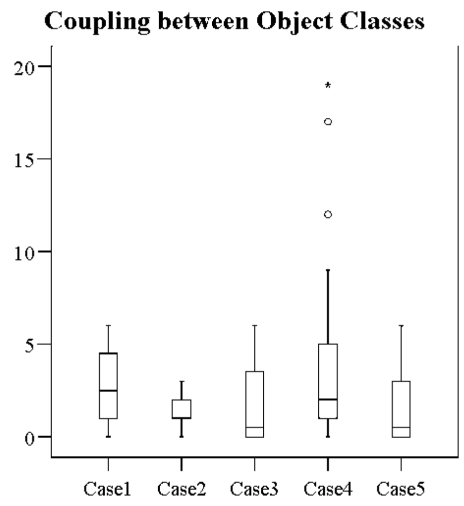


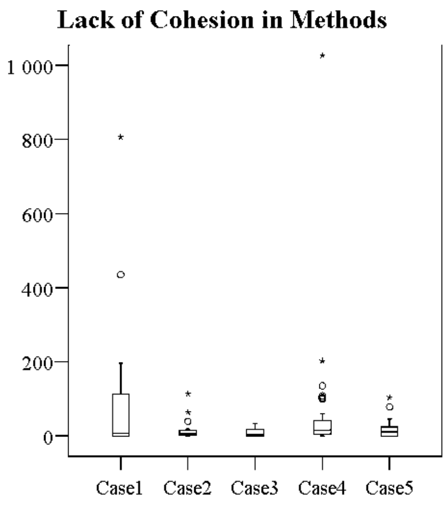


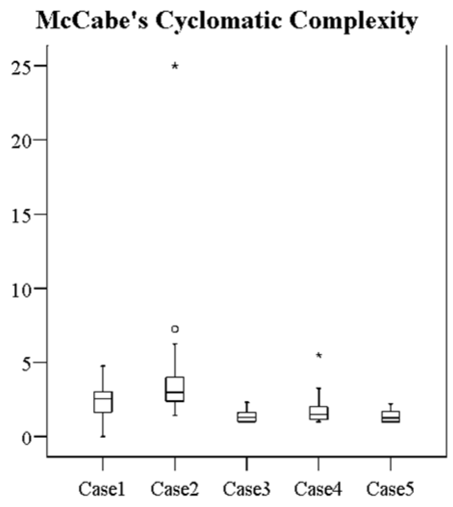


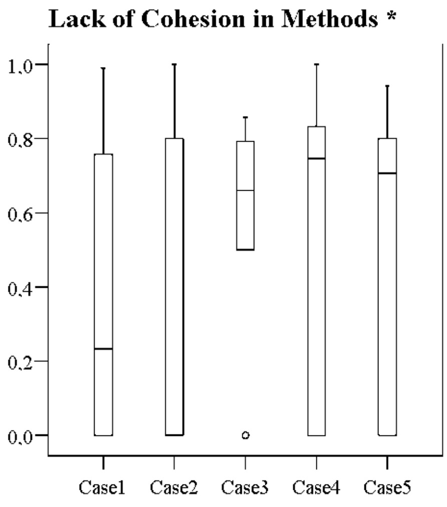


Appendix 1b: The Results of Traditional Metrics

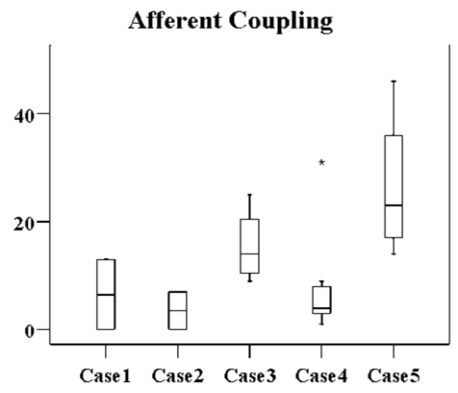


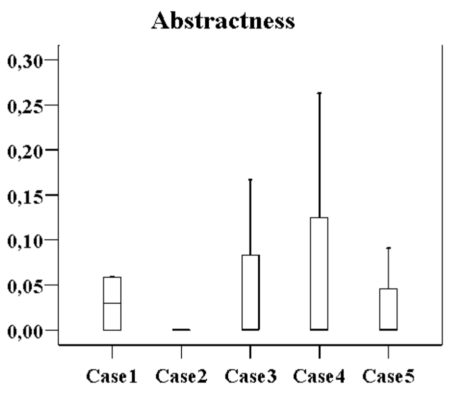


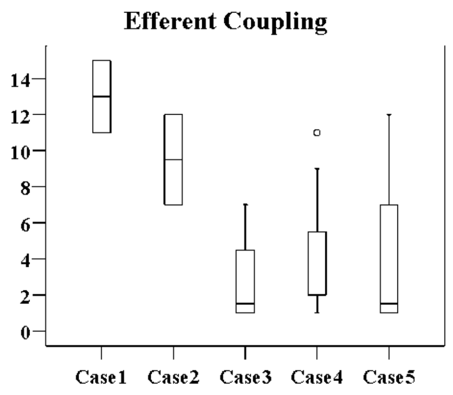


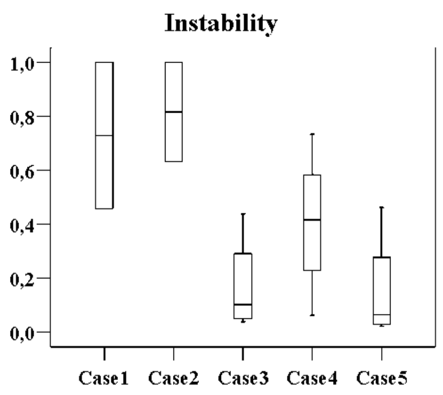


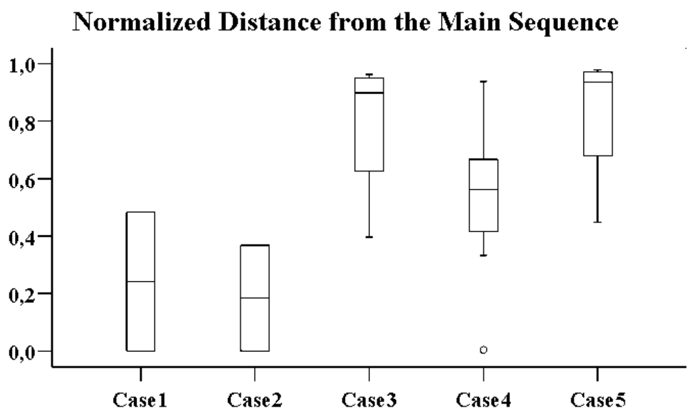
Appendix 2: The Results of Dependency Management Metrics











Measuring the Human Factor with the Rasch Model

Dirk Wilking, David Schilli, and Stefan Kowalewski

Chair for Computer Science 11, RWTH Aachen University

Abstract. This paper presents a test for measuring the C language knowledge of a software developer. The test was grounded with a web experiment comprising 151 participants. Their background ranged from pupils to professional developers. The resulting variable is based on the Rasch Model. Therefore single questions as well as the entire test could be assessed. The paper describes the experiment, the application of the Rasch Model in software engineering, and further concepts of measurement.

Keywords: Human factor, Software engineering experiment, Rasch model.

1 Introduction

Having executed a few experiments (cf. [1],[2]), the authors had severe problems using variables like time, lines of code or cyclomatic complexity. One major point is that lines of code and cyclomatic complexity did not reveal a satisfying correlation with the time needed to fulfill a software task. The scale of the variables appeared problematic, too. For example cyclomatic complexity is only useful to find very difficult functions with high values. The difference between values cannot be interpreted, though. Thus, cyclomatic complexity is regarded only dichotomous in nature and lacks precision. The reliability of the variables, especially time, seems to be problematic. The reason is that while programming, experiment participants appeared to be either lucky to find a solution from scratch or to be unlucky and trying around. Thus, measurement of time has a probabilistic aspect which lowers its precision. The impression arose that a difference in the development ability between participants existed and had a more severe influence on the course of the experiment. The problem encountered when assessing participant knowledge was the imprecise variable of years of development provided by each participant. The problem with this variable is that even someone with 10 years of software development experience might not be good at it. In order to assess the influence of personal ability on software development, a solution for a measurement was sought in other disciplines in the sense of [3].

Regarding human factors in software engineering, the number of sources for this topic is scarce. One part of research based on the human factor is presented in [4], where personality types in projects were identified with a Myers Briggs Type Indicator. A further aspect of human centered research in software engineering is the cognitive aspect found for example in numerous works by Wang (cf. [5], [6]). In this area, software comprehension and reading techniques are important categories. These different approaches are represented in [7] again. In general, the human factor in software engineering is only covered lightly with several directions of interests. HCI or education related research, were the human factor is much more present, are omitted here as they do not focus on the software engineering process.

A general shift of the paradigm guiding software engineering is proposed by Cockburn [8]. A human centered approach is described while omitting any form of quantitative measurement. As this appears a step to far, a mathematical foundation for quantitatively measuring person abilities for software engineering is borrowed in the following.

2 The Dichotomous Rasch Model

In general, a test construction consists of several questions (called items from now on) measuring one variable. In this case, the variable was "C knowledge" which had to be measured using multiple questions. The answer to each question was either correct or not and this was coded as one or zero respectively. Table 1 presents an excerpt of the data. The answers of a participant are coded in one line, while each column shows the answers to the same question. The sum of the correct answers for a column can be considered a difficulty statistics of a question[1]. Simple questions are correctly answered by more participants than difficult questions. In addition to the item difficulty, the sum of a single line expresses the ability of a person. The more correct answers are given, the higher is the ability of the person concerning the measured variable. In addition, items as well as persons can be ordered based on these sums.

Table 1. Coding of correct and incorrect answers

Participant	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9	i_{10}	i_{11}	i_{12}	i_{13}	i_{14}	i_{15}	i_{16}	i_{17}
71	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
131	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
34	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
114	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
126	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
134	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
9	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
37	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
70	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
84	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
20	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0

[1] Under the assumption that the Rasch Model holds.

The Rasch Model incorporates item difficulty σ_i and person ability Θ_v as parameters influencing the probability of a correct answer. Thus, the probability of a correct answer p_1 in table 1 for a person v and an item i (a specific cell) is calculated with:

$$\log \frac{p_1}{p_0} = \Theta_v - \sigma_i$$

with p_0 as probability of a wrong answer. Rearranged to p_1 it is

$$p_1 = \frac{\exp(\Theta_v - \sigma_1)}{1 + \exp(\Theta_v - \sigma_i)}$$

This model constitutes the base for further tests and algorithms ([9],[10]) as used in section 3.5.

2.1 Examples of Questions

The main challenge when creating a test for a variable is to find appropriate questions of different difficulty for the variable. As in this case questions regarding the C language were gathered, technical concepts of that language were mainly asked. These questions cover the preprocessor, pointers, call by value and call by reference, dereferencing, dynamic memory allocation, function pointers, operator precedence and so on. An example of a question is:

```
Please write down the output of this program.

int digit = 100;
int *No;

No = &digit;
printf("%d", No);
```

This question (item 10) aims at dereferencing, as in this case, the address of the variable “digit” is saved to the variable “No”. Thus, the result was not a concrete value, as the address of “digit” is not given in the program fragment. Accordingly, all answers had to be open questions, in order to allow complex answers. A benefit of this step is that guessing the correct result is difficult, as simply inserting values of the program fragment often did not lead to a correct result. A drawback is that each answer had to be evaluated on its own and no automated analysis was done. Additionally, for some answers it was difficult to judge them as correct or incorrect. This was countered with strict definitions of correct answers.

Another example of a question is:

```
What kind of data structure can be stored with this definition?

char *(*(var[10]))(char *string1, char *string2)
```

Here, the participant's knowledge of function pointers is assessed (item 17). This represents the most difficult question which was supposed to need a higher degree of language knowledge. Nevertheless, it was assumed that programmers that were used to the C language directly saw the structure within the code line. Beginners and intermediate developers were assumed to not being used to functions pointers and thus giving an incorrect answer.

2.2 Advantages of the Rasch Model

One benefit of the Dichotomous Rasch Model is its simplicity together with a wide spectrum of post mortem analysis steps for a test. One example is the LR test, which compares the Rasch Model with a perfect, saturated model. If the likelihood of the Rasch Model significantly deviates from the saturated model, the Rasch Model does not hold and is rejected. Other tests focus on the homogeneity of persons and items. For example items are assumed to measure the same variable and thus are considered homogeneous in this aspect. If an easy item was too difficult for persons with higher person ability, the item might not measure the same variable and thus should be excluded.

2.3 Application to Software Engineering

Regarding software engineering programming experiments, the effect strength of novel techniques appears problematic. Novel techniques are always of major interest, but their strength sometimes is so small, that other factors mask its effect. One of the masking factors is assumed here to be the developer's programming ability. Experiences with software development projects, language knowledge, algorithm knowledge, development environment knowledge, and other person related abilities may have an effect on the time a participant needs to develop a program. Indirectly, this is shown by performance estimation of developers as done in [11]. A factor of three is reported as difference in performance with natural outliers to be found sometimes. Regarding this from a software engineering view, a length of a development task might be depending on the person executing the programming task. Finding a technique with an influence of approximately the same strength as a factor of three appears at least problematic.

The use of the Rasch Model within practical software engineering is limited. First of all, a developer's knowledge is subject to change during a project. A static assessment using a single test in the beginning thus is not appropriate. In addition, software engineering practices, projects structures, roles etc. can be changed swiftly in contrast to the abilities of a developer. Thus, person related variables are not actively changeable and out of scope.

For academic purposes, person based measuring might allow a prediction of development effort. A scatter plot of language knowledge with lines of code or program memory usage might reveal a prediction for software development. In addition, execution of a test consisting of a few questions is more economic than a pretest consisting of a complete development task. Lastly, it must be pointed out that person abilities are regarded an additional measurement variable to "hard" variables like program reliability, project progress and time. It is regarded as an additional control device for the internal validity of studies.

3 Experimental Evaluation for the Variable C Language Knowledge

3.1 Overview

Before the actual evaluation was done, a pretest of 40 questions was done with members of the chair. Here, redundant questions (in terms of difficulty) were identified and removed. In addition, the test was subjectively regarded too difficult and easier questions were preferred. Finally, 17 questions were chosen for the final test.

3.2 Participants and Background

The experimental evaluation of the C variable was done as an online-experiment. The request for participation was posted in different bulletin boards. Regarding the participants, the choice of bulletin boards for a call for participation was critical. The aim was to include non-programmers, non-C programmers and C-only programmers in the test. The following (german speaking) bulletin boards were selected: mikrocontroller.net, c-plusplus.de, chip.de, computerbase.de, informatik-forum.at, and others with an additional group of local dormitories. Most bulletin boards were selected because of the community they represented. In addition to C language specific bulletin boards (mikrocontroller.net), general programming boards (c-plusplus.de, informatik-forum.at), general technical boards (chip.de), and non-technical boards (dormitories) are represented.

Regarding the participants' background, 151 participants are included in the study. Their general programming experience and special C language related experience is shown in Figure 1. The general knowledge of programming includes

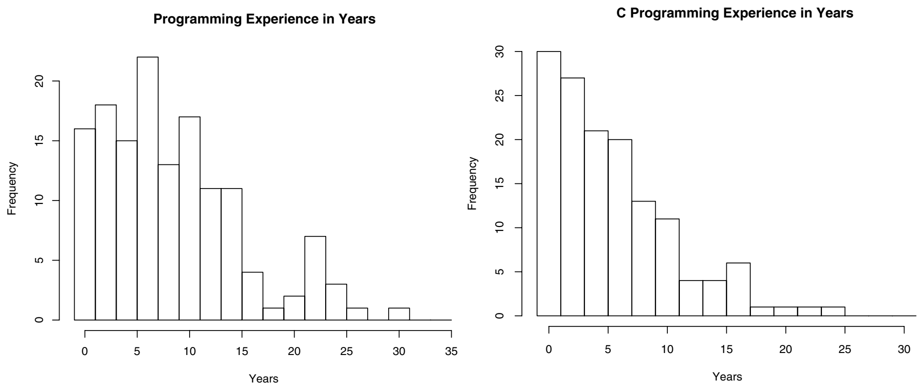


Fig. 1. Histogram of the number of years the participants were programming

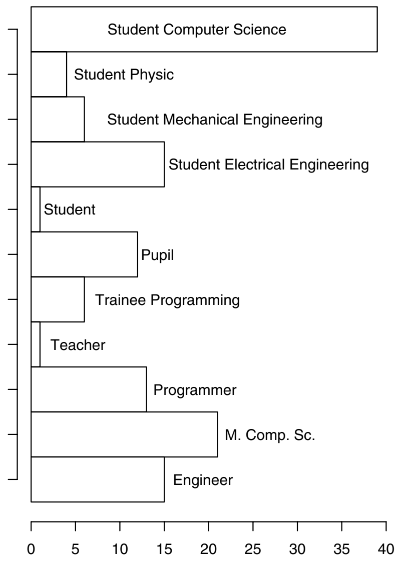


Fig. 2. Frequency of background categories for participants

some long term programmers with a language experience of over 20 years. Regarding C, knowledge of this language is not as common compared to the general programming knowledge, which was expected.

The participants' occupation is shown in Figure 2. The majority has an educational background with most in this category being students of a technical specialisation. About a third of the participants had a professional background of software development with a few engineers included in that category.

3.3 Tasks and Procedure

The task the participants had to fulfill was to answer 17 questions in an online questionnaire. In addition, some questions asked for the background of a person. This was done to check the external validity of the experiment and to check for a correlation of the measured variable and for example the years of programming. All answers were treated anonymously. At the end of the experiment, a price of € 50 was given to a randomly selected participant in order to increase motivation. The average time to fill out the questionnaire was 22 minutes.

3.4 Internal and External Validity

The internal validity of the experimental study suffers from the low control possibilities during the experiment. As the test itself was only a simple website and accordingly no additional software could be installed, participants could have used various sources like the internet sources, books, and other persons to fill out the questions. Additionally, it could not be controlled if a person filled out the questionnaire at a different computer twice.

The external validity relies on the type and quality of questions. These were based on real source code which was checked with a compiler. The kind of questions aimed at several language aspects with the language itself being standardized. The participants had a different experience level of C as described in section 3.2. In order to ground the variable, 151 participants appears as an acceptable number. One drawback is that the participants could not be selected, but their participation was based on motivation possibly leading to above average values as only "good" developers participated.

3.5 Results

The first step of analysis is the computation of the difficulty of each item. This is shown in table 2. The values are shown as logits of the probability:

$$Logit : \log \frac{p(X_{vi} = 1)}{p(X_{vi} = 0)}$$

with X_{vi} as the answer for a person v and an item i. A difficulty of zero indicates an item where the probability of correctly answering it is the same as the probability of incorrectly answering it. For the item parameter, negative values indicate easier items and positive values difficult ones. For calculating the item parameters, several algorithms and statistical programs exist. The program to calculate parameters used here is MULTIRA[2] and the results were validated using the eRm [12] package from the statistics software R with an additional self written script implementing the UCON algorithm as described in [10].

The next step consists of checking how well each item fits to the measurement model. This can be done by so called infit and outfit statistics. Outfit is a chi-square statistic which is sensitive for unexpected observations. Infit is a weighted chi-square statistic sensitive to unexpected patterns of answers (cf. [10]). Positive values indicate an underfit, while negative values shown an overfit of the according item. For the t-standardized row, values greater two are generally regarded problematic. Negative values lesser two are accepted because although they indicate a misfit to the model, an increased discriminatory power of an item is desirable. The problematic values are marked with a question mark. These items should be reworked or simply removed from the test as done below.

A graphical representation of the fitness is shown in figure 3. Here, two artificial groups are created by dividing the participants at the median person ability value. For each group, the parameter value is calculated and the corresponding values are used as coordinates in the plot. Points ideally are on a line indicating the same difficulty for both groups. While some question seem to have a good fit, other certainly need to be refined in the future to increase the quality of the test.

3.6 Test Revision

In order to increase test quality, three items are removed from it. Reasons for bad items were questions which could be misinterpreted. As a correct understanding

[2] http://www.multira.de

Table 2. Item parameters and fitness values

Item	Difficulty	Standard Error	Infit t	Outfit t
Item 1	-2.09	0.298	0.857	0.916
Item 2	-2.09	0.298	-0.413	-0.108
Item 3	-2.001	0.292	-1.152	-0.470
Item 4	-1.606	0.268	3.065!	2.277!
Item 5	-0.349	0.218	0.222	0.3
Item 6	-0.302	0.217	-2.009	-0.709
Item 7	-0.256	0.215	3.130!	3.279!
Item 8	-0.074	0.211	-0.216	-0.580
Item 9	0.057	0.208	2.373!	1.854
Item 10	0.311	0.204	-1.927	-2.311
Item 11	0.393	0.203	-2.446	-2.400
Item 12	0.555	0.201	-0.557	-1.088
Item 13	0.674	0.200	-2.127	-2.283
Item 14	0.831	0.190	0.797	0.210
Item 15	1.649	0.201	-1.530	-1.608
Item 16	1.729	0.202	-0.604	-0.645
Item 17	2.57	0.224	-0.858	-1.015

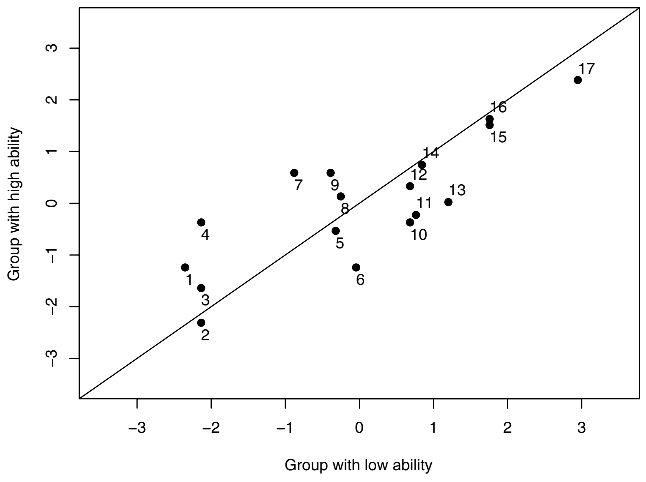


Fig. 3. Goodness of fit plot for two separated groups by median of person parameter

of the questions thus relied on accurate reading instead of C language knowledge, some questions did not focus on the true variable to be measured by the test. Especially for easy items, accurate reading seemed to dominate the difficulty of an item. One easy question was item four:

```
Please compute the variable solution.
In which order was the term computed?

int solution;
solution = 8 / 4 * 2;
```

In this case, operator precedence was asked for which in this case is a simple left to right execution. Although a simple question, describing the actual order was not always done correctly by persons with high ability. Thus, the question had to be removed. Item nine represents problems with correctly answering pointer based questions. Here, person ability did not seem to protect against incorrect code interpretation:

```
Please write down the value of v.

void funct( int *x ){
  *x = 5;
  x = (int *)malloc(10000);
  *x = 10;
}

MainProgram:
int v = 8;
funct(&v);
print-out of v
```

In this example, the participants had to have an eye on the address a value was written to. As the variable `x` gets a new memory location, the second memory writing is not done to the global address.

Table 3. Item parameters and fitness values for revised test

Item	Difficulty	Standard Error	Infit t	Outfit t
Item 1	-2.542	0.324	1.729	1.354
Item 2	-2.542	0.324	0.312	0.918
Item 3	-2.435	0.316	-0.657	0.092
Item 5	-0.517	0.231	1.123	1.648
Item 6	-0.463	0.230	-1.239	-0.398
Item 8	-0.206	0.223	0.636	0.85
Item 10	0.226	0.215	-0.984	-1.699
Item 11	0.318	0.214	-1.940	-2.034
Item 12	0.498	0.211	0.004	-1.156
Item 13	0.630	0.21	-1.241	-1.993
Item 14	0.805	0.209	1.810	0.678
Item 15	1.709	0.21	-1.272	-1.423
Item 16	1.797	0.211	-0.071	-0.102
Item 17	2.722	0.233	0.353	-0.496

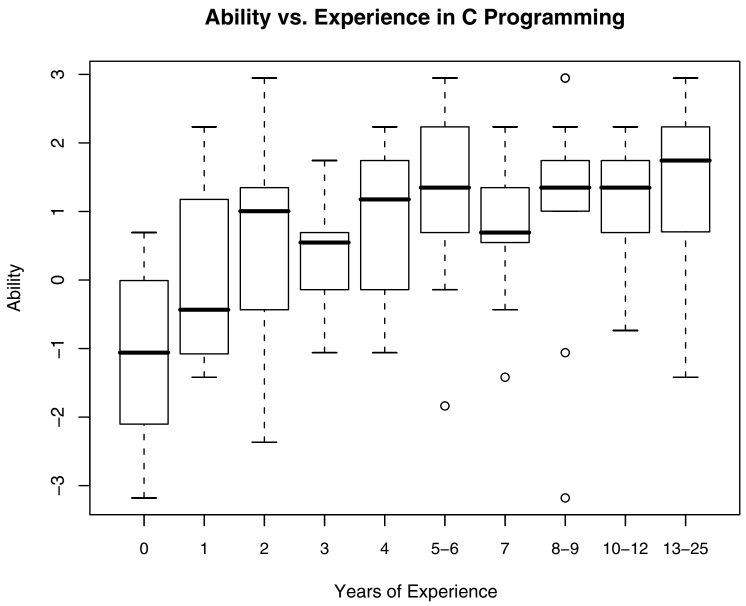


Fig. 4. Boxplots for parameter estimates of C knowledge versus years of programming

By removing them, new parameter and fit values can be calculated as shown in table 3. Three questions were removed in order to get a satisfying fit to the model. The resulting difficulties show a lack of easy items between parameter values of −1 to −3 which has to be fixed in the future.

Figure 4 shows the knowledge of the C language plotted against the number of years of C programming. Here, a rough asymptotic correlation of the new variable with C experience in years can be seen. As this fits the expectation of the variable well, this gives a hint on the validity of the variable.

As the mean value of participant ability is at 0.61, the test was in general too easy. As the test does not use extreme scores, it covers 3 to 93 percent of the participants. Summing up, a c-knowledge metric was created which can be measured in about 16 minutes. The resulting variable is interval-scaled and it is grounded on 136 persons. Its value within experimentation has to be evaluated, though.

4 Further Concepts

C knowledge is rather easy to measure as the language C, its difficulties, syntactical problems, and important technical parts are known. Quantification of this variable thus was a conservative decision compared to other variables which are presented in the following.

4.1 Viscosity

The term viscosity is taken from [13] and [14]. It describes the resistance of a programmer to local changes. Measuring this attitude, although vague in nature, could be possible in a controlled, variable oriented way. The attitude could be assessed using situation oriented surveys where the programmer is required to make a decision which might conflict with his resistance to change. Another way to understand viscosity is the ability to find new and different solutions for problems in order to solve them. Testing this ability becomes difficult as questions for this aspect should allow open answers to gather all possible solutions of a problem.

4.2 Experience

One of the most powerful arguments of doing software engineering and omitting its pitfalls is having sufficient development experience. Measuring this aspect appears extremely difficult and methods to achieve this are mostly part of other disciplines. Nevertheless every proposed factor of influence on a software engineering project should be quantified and tested for effectiveness.

Regarding an assessment as a variable, some general thoughts are given here. A test for experience might have the general form of a situational survey based on decisions that must be made or a risk assessment that must be given. Questions must be prepared in conjunction with software engineering experts in order have a correct base for them. An interdisciplinary approach and continuous adaption of questions to incorporate technical change seem to be appropriate for this difficult variable.

5 Conclusions

This paper presents the Rasch Model as a way to assess person ability in a quantitative way. As a first step, C language knowledge was measured as a variable using multiple questions. An experiment was executed using an online survey attracting 151 participants. Using the rich evaluation possibilities of the Rasch Model, problematic questions could be identified and removed from the test.

In addition, further concepts for measurement are discussed. These comprise abstract ideas like programming viscosity and project experience. Although difficulties are expected when creating tests to assess these variables, an above average effect strength on software projects is expected from these variables.

The questions needed for the test as well as the data can be obtained from the authors.

References

1. Salewski, F., Wilking, D., Kowalewski, S.: The effect of diverse hardware platforms on n-version programming in embedded systems - an empirical evaluation. In: Proceedings of the 3rd International Workshop on Dependable Embedded Sytems 105/2006, Vienna University of Technology (2006)
2. Wilking, D., Khan, U.F., Kowalewski, S.: An empirical evaluation of refactoring. e-Informatica - Software Development Theory, Practice and Experimentation 1(1), 28–44 (2007)

3. Singer, J., Storey, M.A.D., Sim, S.E.: Beg, borrow, or steal (workshop session): using multidisciplinary approaches in empirical software engineering research. In: ICSE, pp. 799–800 (2000)
4. Karn, J., Cowling, T.: A follow up study of the effect of personality on the performance of software engineering teams. In: ISESE 2006: Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering, pp. 232–241. ACM Press, New York (2006)
5. Wang, Y.: On cognitive properties of human factors in engineering. In: Fourth IEEE Conference on Cognitive Informatics, 2005 (ICCI 2005) (2005)
6. Wang, Y.: On the cognitive informatics foundations of software engineering. In: Proceedings of the Third IEEE International Cognitive Informatics 2004 (2004)
7. John, M., Maurer, F., Tessem, B.: Human and social factors of software engineering: workshop summary. SIGSOFT Softw. Eng. Notes 30(4), 1–6 (2005)
8. Cockburn, A.: The end of software engineering and the start of economic-cooperative gaming. Computer Science and Information Systems 1(1), 1–32 (2004)
9. Fischer, G.H., Molenaar, I.W. (eds.): Rasch Models. Springer, Heidelberg (1995)
10. Wright, B.D., Masters, G.N.: Rating Scale Analysis. Mesa Press (1982)
11. Prechelt, L.: The 28:1 grant/sackman legend is misleading, or: How large is interpersonal variation really? Internal Report 18, Universität Karlsruhe, Fakultät für Informatik (1999)
12. Hatzinger, R., Mair, P.: Extended rasch modeling: The erm package for the application of irt models in r. Journal of Statistical Software 20(9), 1–20 (2007)
13. Hoc, J.M., Green, T.R.G., Samurcay, R.: Psychology of Programming. Academic Press Inc., London (1990)
14. Rosson, M.B.: Human factors in programming and software development. ACM Comput. Surv. 28, 193–195 (1996)

Empirical Analysis of a Distributed Software Development Project

Przemyslaw Cichocki and Alessandro Maccari

Nokia Siemens Networks,
Sienna, 39, 00833 Warsaw, Poland
{przemyslaw.cichocki,alessandro.maccari}@nsn.com

Abstract. In spite of the abundant research that promotes different methods for software development, and the current method war amidst agile and disciplined methods, little research is done to actually figure out whether real projects, carried out in industrial environments, benefit more from either approach. This paper analyses a real project team's opinions and feelings about project management techniques, software development methods and cultural difference in a multi-site project where traveling and communication are made difficult by restrictions and low-quality infrastructure. The different sites also worked in different time zones and with different working week patterns. The project team members almost unanimously indicated that the presence of a local team leader with authority and flexibility to cover a role that is not exactly as assigned in the beginning, is key factor for the success of this sort of projects. While there was no consensus on whether the project was agile or disciplined, evidence seems to hint towards a more disciplined approach, probably as a compensation for the higher degree of uncertainty that derives from the distributed setup. While the findings of the case study cannot be extended to other organizations without caution, we do infer a number of conclusions on cultural differences, project management tools and techniques.

Keywords: Agile, Distributed development, Project management techniques evaluation, Human factors.

1 Introduction

Current research on software development methods regularly produces a large amount of material, such as proposals for new software project management methodologies, variations to existing software development methods, enhancements to tools, suggestions for improvement of good practices, and so on.

However, surprisingly little effort is spent on trying to apply research findings to practical case studies and document the feedback thereby obtained for the use of the community. This is probably for either of two reasons: the industrial world does not apply the latest findings of research, or otherwise does not have the time or occasion to report on the findings, which are mainly used (if at all) inside the company where the researchers work.

Agile software development methodologies (like Scrum or Extreme Programming) promised to improve the way we develop software in industry. By enhancing communication and putting individuals, rather than processes or tools, in the headlight, agile methods have brought us closer to the fulcrum of software development: the human being, with his talents, defects, inconsistencies and creativity.

However, agile methodologies work best in small teams that are co-located, or at least they can communicate easily and without boundaries. Whether they can be applied to distributed project teams is still a subject of research. It is also not clear whether agile methodologies can be applied as effectively in teams where cultural differences are vast, and where the working day and week do not overlap completely.

From common sense, it is not without a reason that a number of methodologists advocate usage of more structured and disciplined methodologies [2] in situations such as the one we take into consideration in the paper.

We present the results of a research that is focused on practical application of software project management and software development techniques in a distributed team setup. The team was distributed across three sites in two different cities, each located in a different country. Traveling between the countries was made difficult by strict visa regulations, and the quality of international telephone lines was generally low. The customer and the project manager were located in one of the cities, while the main development centre was located in the other city, alongside with some 70% of the members of the development team.

In order to analyze the impact of the team distribution on the performance of the members of the team, as well as their opinion on the effectiveness of certain software development and project management methodologies and techniques, we carried out a series of interviews with all the individuals that worked on the team for more than two months.

We present a number of findings from our research, and make statements on validity threats, as well as on applicability of the findings to similar organizational environments.

We do not claim that our research is complete. In fact, we believe that it poses the basis for a family of experiments, as advocated by [4], aimed to characterize with greater detail the phenomenon under study.

2 Research Background

2.1 Motivations

The issues that originate the need for this research are the following:

- The lack of rigorous experimental data on the industrial validity of certain project management and software development methodologies and techniques: what techniques do real software developers value most, and why?
- The uncertainty regarding the applicability of any of the known methods and tools to a fully distributed team, which works on tight deadlines and with limited possibilities to interact: will agile methods be more effective than structured ones in this sort of case?
- The absence of definitive findings from existing research on whether an agile approach to project management produces better results (and is better appreciated)

than a structured, disciplined approach, especially in the case of distributed project team that works in a deadline-driven business environment.

- The insufficient number of studies that aim to understand how the usage of certain project management techniques and methods affects the effectiveness of the software developers (as perceived by them) and of the entire distributed team.

Most of these issues could be restated without limiting the scope to the organization where the research work was carried out, or even to the entire telecom software development domain as a whole.

However, the scope of this study will be limited to the organization where the project team under study operates. We believe that a small study such as this one cannot be generalized without exercising a lot of care, and that the conclusions of this research should in general only be deemed valid within the specific environment under consideration. Additional remarks on the validity of the study are made towards the end of this paper.

2.2 Research Goal

This research work aims to answer the following high-level research questions:

What software development methods and tools are deemed to be most effective in a distributed software development team?

What project management techniques and personal qualities of the project manager are most useful in the environment where the project team under study had to operate?

How did cultural differences influence the project team's life, and how can project management methodologies maximize the positive (or minimize the negative) impact of cultural differences?

How agile or disciplined is this sort of project deemed to be, and how is this judged by the project team members?

2.3 Research Philosophy and Approach

Orlikowski and Baroudi provided an excellent classification of philosophies and approaches for information technology research [5].

Following their reasoning, we may list the following facts as characterizing our research work.

- Ontologically, we do not make any assumption on the behavior of project team members, nor on the reflections of project management or software development methodologies on the project organization. We assume that such knowledge is unknown, and try to deduce it from appropriate analysis of collected data.
- Socially, we do not assume any predefined regularity to rule the social reality where the project team members work (and thus live). Anyone who has been involved in software development for a telecom industry can confirm that this assumption is true in most cases!
- Epistemologically, we believe that the phenomena of interest (for instance, the relationship between team members, the consequences of adoption of certain project management techniques and methods, and the intrinsic agility of the project)

can be understood by in-depth enquiries made with the development teams. From such facts, our research work can be classified as interpretive.

The following sections outline the organization of the project team (research environment), the empirical study design, and the method of collection of the information. We also present our initial answer to the research problems stated before.

2.4 Project Team Environment

The project team operates in a real, industrial environment within Nokia Siemens Networks, a major provider of telecommunication infrastructure and services, within a single project. The purpose of the project is to implement and deliver a charging and mediation solution for a customer located in the Middle East. The majority of the project tasks consist in implementing custom add-ons or new features on top of an off-the-shelf product platform.

Normally, in the charging and mediation domain it is difficult to reuse software from other, similar projects that have been carried out in the past. This is due to the fact that charging business models (and thus the technical requirements for the software solution) vary substantially from customer to customer, and it is often cheaper and more convenient to implement such features from scratch rather than reusing work done before. We believe that the all-too-famous NIH (Not Invented Here) syndrome has not played a substantial role in these decisions.

The project under study is part of a larger programme that delivers a series of value added service (VAS) solutions to the same customer. The project team environment is typical of a large company that delivers critical software solutions to a customer that operates in emerging markets: largely driven by deadlines that are so tight to seem unrealistic, and in a generally unstable environment where requirements change often, access to common resources is limited due to the lack of infrastructure, and pressure on the project team members is applied by several stakeholders (both within and without the organization that employs them), and not always with the knowledge or approval of the project manager.

From the technical point of view, the purpose of the mediation solution is to collect Call Data Records (CDR) from different network elements (like mobile switches, GPRS nodes, MMS center. etc.), process their content according to the customer requirements and send them in a format that is readable by the customer's billing system. The solution is based on a certain Nokia Siemens Networks product, on top of which our team implemented software that enabled processing CDRs as per the customer requirements.

At the time when the research work was carried out, the project had lasted around 12 months, and around 1300 man working days had been spent on it. It employed 12 people, out of whom 9 were interviewed for this research.

The team consists of one project manager, one or two technical architect (who own the technical solution and had decision power on the technical architecture) and software engineers numbering between three and eleven. The number of software engineers has varied over time according to the need of the project and according to the number of tasks that have had to be carried out concurrently. The project manager can allocate and release technical resources on a relatively short notice.

Most engineers, who have been assigned tasks that range from implementation to testing and from documentation to deployment at site, belong to two contractor companies that were based in the same city (but in different buildings) in the European Union. One of the technical architects is co-located with the engineers. The project manager is based in the same city in the Middle East as the customer. Depending on need, a number of engineers have been based in the Middle East as well. On average, approximately the project staff was based in Europe at a given point in time, with the remaining half being based in the Middle East.

Travel between the two locations has been encouraged by the project manager, and sufficient budget has been allocated for the project team to travel between the two sites. However, visa restrictions and tight project delivery deadlines have advised against excessive travel.

The quality of the telephone network between the two countries that host the project staff is generally low, which discourages the usage of phone calls as a frequent communication means. In the Middle Eastern location, the quality of internet service is also somewhat low.

The project team included people of four different nationalities. However, the majority of the technical staff (architects, engineers) shared the same nationality and mother tongue. The customer team and other internal stakeholders belonged to a large number of cultures and nationalities. The English language was normally used for communication between the team members and with most project stakeholders.

Customer and Nokia had to build new organizations from scratch. Although Nokia was already present on the market it was only the mobile phone market and the Nokia Networks (part of Nokia dealing with core network – since April 2007 it became part of the Nokia Siemens Networks) was absent in the customer country.

Setting up the companies it is always a big challenge. In this case it was extremely hard because of tight schedules and lack of resources. Both companies had to attract employees not only from customer's country but virtually from all over the world.

Organizational structure was well defined at high level in both cases and borders of responsibility were clearly marked. However when it came to step down into organizational chart it turned out that there were many communications problems. Due to high pressure coming from tight schedules people were overloaded with work and sometimes it was impossible to get the needed information immediately. This lead to delays and in the end in giving up some of the project cycle phases (e.g. performance tests) in order to meet the deadline. This situation applies not only to relation Nokia Siemens Networks – customer but also internally. For example customer's IT department had problems with proper communication with customer's marketing, in Nokia Siemens Networks happened that one team has changed network device settings without notifying other teams that relied upon those configuration.

All of this forced the project team subconsciously to adjust to the major principle of Agile Manifesto – embrace the change [1]. Although that project was following the certain process (classical approach: requirements gathering, solution proposal, implementation, testing, deployment) the team was aware that despite the fact that requirements was signed off the customer can change it at any moment. Obviously project manager was trying to avoid that situation and teach customer that it should follow the certain rules (by change request process for example) but sometimes it was really inevitable – in the end it was our customer and project team should make any effort to fulfill its needs.

3 Survey Characterization

This research work is carried out by means of a survey that involved the majority of the team members who had technical roles (e.g. software developer, testing engineer, requirement engineer, technical architect). The survey is articulated into nine questions, grouped in three categories:

1. Method and tools, including two questions that cover generic aspects of the applied software development method.
2. Project management techniques, including four questions that cover specific aspects of project management.
3. Cultural differences, including three questions that focused on the impact of cultural differences (mostly, the difference between European and Middle Eastern culture).

When designing the survey, we put particular emphasis on structuring questions so that respondents would be encouraged to give a lot of details, and limited interruptions even when the interviewer felt that the response was drifting out of the original scope.

Questions were structured in an open way, which is typical of interpretive research. We made every effort to avoid guiding the respondent towards a specific answer, or towards a yes/no answer. When the answer was brief (e.g. when someone replied in the lines of "everything worked well") we tried to ask further questions, trying to dig out a more detailed opinion.

Most of the respondents were interviewed face to face, with the exception of two respondents who were interviewed remotely by means of email. Interviews lasted around one hour each, and were conducted by both authors of this paper.

4 Results

Below the result of the survey are presented. The interview was conducted with nine team members that were involved into project activities for two months at least.

4.1 Methods and Tools

This part of the survey was dealing with techniques and means that team found useful for development and information sharing in the remotely managed project setup.

Most of the team members were contractors so they were originally employed by the external companies (there were two external companies involved in this project) and then indirectly hired by Nokia Siemens Networks. Due to transition period (Nokia Networks was transforming into Nokia Siemens Networks) it was sometimes hard to create the Nokia Siemens Networks accounts (e-mail, intranet access) for those external employees. This was one of the major factors that people felt prevented them from being fully productive during the project. People did not have access to documentation and other resources like software updates and patches which ended up in problems with keeping project deadlines.

The most difficult and challenging part of every project phase was the scope definition because of the communication problems mentioned above. During the implementation the main stress was put on the proper configuration of the mediation device and testing. There was not much pure software development – mostly simple C++ and shell script development tasks. For this reason, the first question (asking which software development techniques helped most in a remote setting) was answered in an insufficient manner. Either there was no answer or the question was misunderstood by the survey respondents. However, one of the team members found the Extreme Programming technique (pair programming) as useful during the deployment and functional testing.

The second question in this section was referring to the tools that were found useful in the information and knowledge sharing. Perhaps not surprisingly, the most efficient way of communication was deemed to be face to face meetings. Despite the fact that so-called modern channels of communications (fixed and mobile telephony, internet) were at every team member's disposal, this classical way of exchanging information was found as the most reliable and effective.

Unfortunately, this way of communication could not be used very often from the obvious reasons (distance, difficulty in travel, cost), so people have used email and chat for their daily communication.

In particular, email was recognized to be a more formal way of communication. Typical cases when email was mentioned to be effective are getting approval on documents or requirements from the customer (where an email message constitutes a sort of contract), and broadcasting of information (e.g. meeting minutes) to the entire project team. Some respondents questioned the effectiveness of the usage of email for person-to-person communication.

Chat, instead, was deemed more effective and useful for daily, informal information exchange. A typical use case is when random questions must be asked to a certain expert about certain software functionality.

Version control systems (which the project used for documentation and source codes) were also deemed helpful in information sharing. Unfortunately, most of them required access to our company's intranet account, which, as explained in the first paragraph of this chapter, often meant that access to such resources could not easily be achieved by some of the team members.

Finally, we prepared a Wiki website [3], to be used by the entire project, where the project phases were briefly described and the latest documentation (requirement specifications, solution descriptions, test cases, project management plan) was available for download. Wiki was listed also as one of the tools that contributed to better knowledge sharing within the team, although it was mostly perceived as a placeholder where to find project documentation, and not so much as a place where people can actively (by editing the Wiki pages for example) influence the way the knowledge is shared.

4.2 Project Management Techniques

The second part of the survey tries to understand how the project management techniques and method that have been used in this project were perceived by the team members.

The first question in this section is about the project management techniques that did succeed in the interaction between the team, customer and project manager located in different areas. It turned out that it is extremely important for most team members to have a delegated person that would act as a local team leader in every site where the team operates. That person should represent the project manager locally and should be able to take proper actions once the situation requires them (e.g. tensions with customer during project manager absence) and is accountable for their consequences. Additionally he or she should also fulfill a role of the communication gateway to the project manager but need of this was not as strongly desired as the local leadership role.

Another question asked about the general characteristic of the project, focusing on whether people thought it was disciplined or agile. Rather surprisingly, there was no common ground in that matter between the team members. Some engineers claimed that project was agile and some were claiming that it was very (too) disciplined. There was also no pattern in the answers with distinction to the assigned role in the project. The conclusion that we make out of this is that perhaps the concept of agility (and, correspondingly, that of discipline) is not perceived in the same way by people.

The third question asked the team which of the project management phases (distinguished according to the PMI model) was made harder than usual in this project by the distributed team setup. It was pointed out that especially the scope management phase was more difficult to accomplish. This phase needed traveling as it required meetings with customer in order to define the requirements for the given project phase. Due to restriction mentioned earlier (visa, different time zones, shifted weekends) this task was performed very often under time pressure. Team members felt that efficient scope and requirement management could not be carried out at a distance, especially during the requirement elicitation phase and during the inevitable project phases when the customer points towards a scope creep.

The last question of this section asked which personal qualities of the project team members were most helpful in a remote project setup. Not surprisingly, trust and commitment were listed most often in this case. People found it extremely comfortable to work in an environment where everybody could count on other team members to help in case of issues. Also, the fact that some of the team members (e.g. technical architect) acted as a local project manager for the team was deemed important: this allowed people to have clarity about task assignments, and it was possible to make important decisions quickly. Some people pointed out that having a competent substitute person (when one of the key person for the project was on leave – e.g. project manager) is crucial.

It is not easy to extract unique conclusions from this set of answers. However, there seems to be evidence that the members of this team deem a more disciplined approach (rather than a more agile one) necessary in this kind of setup. The emphasis that was put in scope management, and the clearly expressed need for a local team leader seem to point in the direction of a higher amount of discipline. Similarly, the fact that trust and commitment were deemed to be the most useful personal qualities for team members points to the fact that the very nature of agile projects (based on fast prototyping, trial and error and continuous requirement negotiation) does not fit a distributed team setup such as the one we implemented for this project.

4.3 Cultural Differences

In the third part of the survey, we investigated the team members' opinion on the cultural differences in the multinational environment that the project had to work in.

The first question concerned the cultural differences that could be spotted as far as the work approach is concerned. Shifted working days were pointed here as the most obvious difference. They are only three working days that overlap in Europe and Middle East (Monday, Tuesday and Wednesday), and therefore activity planning should take that into account. It happened sometimes that people in both countries have to work overtime in order to finish their task in timely manner.

The issue of mother tongue was mentioned by several team members is not exactly falling into cultural difference category but is worth mentioning. It was noticed that for none of the team members English was a mother tongue. On one hand it was found as an advantage as there was no need of strictly applying the grammar rules, proper vocabulary which made communication process easier. On the other hand the knowledge level of English between team members varied and it was pointed that sometimes it could cause communication problems as well.

The second question queried on how cultural differences influenced everyday activities. The surprising result here is that nobody felt the need to point out any cultural factor that would be disturbing or (even more surprising) stimulating. In two or three cases it was mentioned that a multinational team caused people to be more patient and understanding to other team members. People understood that different nationalities can have different approach to work in terms of pace and quality, and had to adjust their expectations correspondingly.

The last question asked people what they thought should be changed in current project setup in order to benefit more from the cultural differences. One of the issues that surfaced here was the need for careful project management planning. This involves taking into account the shifted working days, leaving enough time for cross-team communication and reviews, allowing people to travel between sites when absolutely necessary, and so on.

The conclusion we may infer from this section is that the fact that the team was distributed among two different sites was judged to be more relevant than the fact that the team had people coming from different cultures and countries. There seemed to be no tangible consequence of cultural difference on daily work.

5 Validity

5.1 Internal Validity

Internal validity is the extent to which the survey results can be extended to similar projects within the organization where the case study was carried out.

It seems like the very specific setup of the project makes it harder to find the general pattern that could be applied to other cases. Several key factors (such as different time zones, customer organizational inefficiencies, project team divided in three sites, visa restrictions in customer country, etc.) are characteristic to this particular case, and make the results of this survey hard to generalize, even inside our own organizational environment.

However, there are some general conclusions that can be drawn from this setup and that we believe may apply to other similar projects in our organization. These include at least the following.

- Need of a local team leader in every site.
- Necessity of good information sharing tools.
- Flexibility in project task assignment (medley of roles).

When generalizing the other results that we explain above, we believe that we should exercise caution, even internally. Further research is definitely needed before safe statements can be made in this respect.

5.2 External Validity

External validity concerns the applicability of the survey results to different organizations, countries, teams and domains.

All the considerations that we have made for internal validity obviously hold when considering external validity. Actually, we believe that even better care should be exercised when extending the validity of the research outside the boundaries of our organization or domain.

The environment where this project operates is fairly unique, as it involves a very aggressive customer (mostly, in terms of deadlines), an organization (Nokia Siemens Networks) that is relatively new to the country where the customer resides, a blend of experience from different fields inside the project team, and a relatively young average age.

For these reasons, we feel that further research is required before any claims are made about external validity. A series of similar case studies should be repeated across different companies, countries and domains, and results compared before any generic claim is made.

6 Conclusions

The analysis of the survey that was conducted among the team members gives indirect clues about what can be deemed helpful in this sort of project setup. We now try to answer the four research questions that were at the base of carrying out this case study.

1. What software development methods and tools are deemed to be most effective in a distributed software development team? Our research cannot give a definitive answer to this question, as there has been little agreement among responders. The issue

requires further research, though we can assert to some degree of certainty that the usage of each tool (chat, email, phone conference, web sites) has to be disciplined in order to avoid generating conflicting messages or annoyance.

2. What project management techniques and personal qualities of the project manager are most useful in the environment where the project team under study had to operate? From the results of our research, we can state that in such environment having a team leader in every site, and being able to trust other team members are by far the most valued aspects. A blend of usage of different tools (email, teleconference, chat, etc.) for different purposes can also solve most of the information sharing issues. The team's emphasis also fell on scope management, which may be an indication that in distributed and dynamic environments where many variables are subject to sudden change, and the possibility to control people is low, effective management of scope is regarded to be the balancing power that makes projects successful.
3. How did cultural differences influence the project team's life, and how can project management methodologies maximize the positive (or minimize the negative) impact of cultural differences? From the results of this research, it appears that cultural differences do not play a major role in this sort of environment, save perhaps for the mother tongue. Actually, evidence points towards the conclusion that cultural differences can even be stimulating, and increasing the productivity and creativity of the people that are involved in the project. This definitely advocates the need for further research in this area, involving experts in sociology and psychology.
4. How agile or disciplined is this sort of project, and how is it perceived by the project team members? From the answer to one specific question we infer that there is no agreement on whether the project was agile or disciplined, which could point out to different interpretation of the concepts of agility and discipline. However, there seems to be evidence that points towards a more disciplined approach, as this guarantees better scope management, more effective assignment of management roles to different sites and emphasizes trust and commitment.

What made this project successful? It is hard to draw an itemized list of success factors in this case. Undoubtedly, it was a mixture of project management techniques, tools and the unique personalities of the people involved in project activities.

First of all, the role of the team leader cannot be underestimated. In every team there should be a person who should act as a local team leader, sometimes taking the responsibilities of the project manager for the tasks like work assignment. Due to limited direct contact with the customer (visa restrictions) it was important that the person who was in charge of defining the scope of the project phase (requirements gathering and documentation) was able to work efficiently under time pressure. As it was described above, face-to-face contact was preferred way of communication, and the requirement gathering phase was found the hardest to perform as it was done remotely.

Another important factor was information sharing among team members and tools that were used for it. Team members were up to date with the current project activities as well as with the future plans for the project. It was done by setting up weekly teleconferences gathering all the people involved in the project and allowing to discuss current issues and actions. It was also important to grant access for everybody to necessary resources like corporation intranet, email, product documentation etc.. Lack of these facilities can lead to frustration and lower motivation of the team.

Finally, the most important success factor was unique set of people. It was not mentioned accidentally in the survey that one of the qualities that helped to overcome the distance was trust and commitment. When people can count and rely on each other they can perform very well despite the obstacles.

References

[1] Martin, R.C.: Agile Software Development, Principles, Patterns, and Practices. Prentice Hall, Englewood Cliffs (2002)
[2] Boehm, B., Turner, R.: Balancing agility and discipline. Addison-Wesley, Reading (2004)
[3] http://en.wikipedia.org/wiki/Wiki
[4] Basili, R.V., Shull, F., Lanubile, F.: Building Knowledge through Families of Experiments. IEEE Transactions on Software Engineering 25(4), 456–473 (1999)
[5] Orlikowski, W.J., Baroudi, J.J.: Studying Information Technology in Organizations: Research Approaches and Assumptions. Information Systems Research 2(1), 1–28 (1991)

Appendix: Survey Structure

This appendix reports the questions that composed the survey in the exact form as the survey respondents heard them during the interviews. The questions were divided into three categories.

Method and Tools
1. What methods and techniques (in terms of software development method) that you used in the project were particularly useful in a remote development setting?
2. Which tools you felt contributed to the information sharing (e.g. teleconferences, emails, chat, wiki, configuration management, etc.) and which did not? Please motivate your answer.

Project Management Techniques
3. What project management techniques (e.g. scope management, resource management, task and assignment management, delegation, customer relationship management) facilitated the interaction between the project manager and customer (located in the Middle East) and the team (located in Europe)? what instead did not work?
4. Was this project more agile or more disciplined? Why? How would you improve the approach?
5. Which phase of project management (scope management, time management, resource management, communication management, risk management, quality management, etc.) was made harder by the distance between you and the project manager and what instead was not influenced?

6. What personal qualities of the project manager, architect and developers helped most overcome the distance? To what extent did the roles deviate from the job description? For instance, did the architect sometimes act as project manager?

Cultural Differences

7. What cultural differences did you notice (as far as work approach is concerned) in your multi-national team (please take into account the customer team as well)?
8. Which of the differences did you find stimulating and having good influence on project performance? Which not? Why?
9. Having the current experience in place would you change anything in project management approach, used tools or methodologies in order to diminish the negative / strengthen the positive influence of the cultural difference?

Extending Software Architecting Processes with Decision-Making Activities*

Rafael Capilla and Francisco Nava

Department of Computer Science, Universidad Rey Juan Carlos,
c/ Tulipán s/n, 28933, Madrid, Spain
{rafael.capilla,francisco.nava}@urjc.es

Abstract. The traditional perspective on software architecture has paid much attention to architecting as a development process aimed at creating the architecture of a software system, as well as the documentation used to communicate the architecture to the stakeholders by means of several architectural views. Recently, the software architecture research community has faced the need to record, manage, and document the design decisions and the rationale that lead to such architecture. Because architectures are the result of a set of design decisions, this design rationale must be properly recorded and managed as a complementary process to the modelling activity. In this paper we detail different types of decision-making activities aimed at creating and using design decisions and how these can be supported with tool support.

Keywords: Software architecture, Architecture design decisions, Architectural knowledge, Architecting activity, Maintenance, Evolution.

1 Introduction

Software architectures have been successfully used in the past decades as the central cornerstone for describing the main functional parts of a software system [2], and the interests of different stakeholders are usually represented in the architecture by means of different architectural views [12], [17]. The more traditional perspective on software architecture [2] has paid much attention to modeling and documenting tasks while they have neglected the rationale that led to such designs. Recently, this point of view is changing to include the creation and use of architectural knowledge (AK) as a first class entity that should be recorded. As all architectures are the result of a set of design decisions [3], the impact and benefits of recording this AK seems to be promising for maintenance and evolution activities. Hence, as software systems evolve, the decisions made during the life of the system should evolve accordingly to the changes performed on the system and to new customer needs. Therefore, a continuous decision-making process happens to meet the goals specified in the requirements.

Recently, the software architecture community has recognized the need to record, manage, and document explicitly the rationale that lead to the creation of any software

* This work is partially funded by the PILOH project of the Spanish Ministry of Education and Research programme under grant number URJC-CM-2006-CET-0603.

architecture. Architecture design decisions become now more important as they bridge the gap between requirements and architectural products. Thus, also traceability in maintenance activities can benefit from this approach.

In this paper we focus on those processes needed to deal with design decisions as a complementary product of the architecting activity. Also, we describe how some of these processes are supported by ADDSS, a web-based tool for recording, managing, and documenting design decisions. The structure of the paper is as follows. Section 2 discusses the representation of design decisions in software architecture. Section 3 deals with the processes that affect the creation and use of AK. Section 4 describes which of the processes mentioned in Section 3 are supported in the ADDSS approach. Section 5 provides some conclusions and outlines possible future work.

2 Representing and Creating Architectural Design Decisions

In the early 90s, Perry and Wolf [15] mentioned the rationale and principles that guide the design and evolution of software architectures. This rationale is used in the reasoning activity as the underlying reasons that motivate the selection of a particular architecture. These ideas have been detailed in [6] to state the need for documenting explicitly architectural design decisions, but not the processes that lead to them. Nevertheless, prior to the definition of the activities that should take place in the creation of such architectural knowledge (AK), it seems necessary to know which kind of information we should represent as part of the design rationale. Design rationale is the justification behind decisions, and different authors have addressed the problem to reflect design decisions as part of the architecture documentation [8]. Tyree and Akerman [19] provide a template list of items for characterizing architectural design decisions. In [18] the authors mention the need for documenting design decisions, because documenting architectural descriptions often based on a component & connector view is not enough. One of the reasons to store this AK comes from the need to carry out highly-cost maintenance processes motivated by architecture erosion or from non existing designs because design decisions were never recorded. Others [16] focus on the explicit representation of assumptions as a way to make explicit the tacit knowledge which is often implicit in the architect's mind. In [5], the authors propose a list of attributes which classifies design decisions into mandatory and optional attributes that can be tailored for each particular organization, as well as a set of attributes specific for describing the evolution of architectures. A meta-model combines the characterization of design decisions with the processes used to manage such knowledge. Similarly, the architecture-centric concern analysis (ACCA) method [21] uses a meta-model to capture architectural design decisions and linking them to software requirements and architectural concerns. The approaches mentioned before highlight the relevance for characterizing the architectural knowledge, but the processes that lead to it are only slightly mentioned.

2.1 Lifecycle for AK Creation

In addition to the AK representation, creating and using AK has to be integrated under the "natural" lifecycle of the more traditional architecting and engineering activities. To date, most software architects have seen architectures as a "product" that has to be maintained and evolved as requirements change. According to [3], [14], architects are changing their more traditional perspective by considering *architectural knowledge as a product*, which should be seen as first class co-product of the architecting activity in order to avoid knowledge vaporization. In addition, *architectural knowledge as a process* [14] "deals with the processes that create and use such AK during the software development lifecycle". Use cases, methods for recording and discovering knowledge, tools and services for supporting the usage of AK fall on this category. In this new scenario, the stakeholders involved in the development of any software architecture may act as "producers" and "consumers" of this AK. According to the classification defined in [14], *architecting* and *sharing* activities belong to the producer side while *learning* and *assessment* belong to the consumer side. These activities have been roughly described in [14] but they need some refinement in order to understand the detailed processes concerning to the creation of AK. Our main contribution in this paper focuses on a more detailed list of the processes and sub-processes that happen during the decision-making activity, as a refinement of the main ones described in [14], such as we outline in next section.

3 Activities for Recording and Using Architectural Knowledge

The activities concerning with the creation of AK are described in table 1. AK. Hence, before a decision is made, a reasoning activity may take place [13]. This reasoning process is based on the rationale and the motivation that guides a decision. The rationale often relies on assumptions made as well as on the analysis of the pros and the cons (i.e.: the implications) of each particular decision. Moreover, we have to take into account the existence of constraints for the decisions as well as the dependencies that may appear between current and previous decisions. Once a decision is made, we should give a concrete status (e.g.: pending, approved, rejected, obsolete) and store it in a readable form for subsequent use. Often, before a choice is selected, several alternatives can be considered. The evaluation of these alternatives means to deal with new decisions and sometimes search for codified AK. In addition, evaluation and assessment activities may happen and used to evaluate between different candidate solutions. Also, depending on the specific phase or project milestone, not all the existing AK may be needed at the same during the decision making activity. For instance, we can store a minimum set of attributes to characterize a design decision during the initial development phase, but a subsequent testing or maintenance activity may need extra attributes (e.g.: responsible, status). In practice, as much of these attributes are stored during the creation of AK more comprehensible would be the decisions made. For each main category of the processes defined in [14] (marked with an asterisk in the tables) we have detailed the set of activities and sub-activities that we believe belong to each category.

Table 1. Activities for creating architectural design decisions

ARCHITECTING (*): Creates and stores AK		
Activity	Sub-activities level 1	Sub-activities level 1
Make decision	Reasoning (rationale, motivation) Select the best alternative	Make assumptions Analyze implications Constraint and dependency analysis Evaluate AK Validate before storing
Characterize decision	Assign status and other relevant items	
Store and document decisions		
Evaluate AK	Reuse AK Evaluate alternatives	Search, Discovery Assessment / Learn

Once an amount of decisions has been stored, this AK can be shared with others. The processes that fall in this category are defined in table 2. In many cases, the boundary between producers and consumers for sharing activities is not clear in many cases. Producers share available knowledge to other stakeholders. AK producers may act also as consumers of codified knowledge. Moreover, architects may share AK with other architects, all of them participating in the development process. For instance, during architecting a well-known pattern can be shared to other architects to discuss its applicability as a suitable design solution. In other cases, once a set of design decisions are made and the first version of the architecture is built, a subsequent maintenance process might need to share some of the decisions made with others interested in learning from previous experiences. From our point of view, knowledge sharing can be a more passive task when the stakeholders review existing AK or even when they query a knowledge base. A more pro-active approach can take place if we want to publish knowledge to others that act as subscribers of such AK (e.g.: use of RSS contents for distributed teams). Active publishing-subscribing strategies as well as discussion groups can provide a more dynamic usage of codified knowledge. Moreover, brainstorming meetings can be organized to share and communicate this knowledge. In this case, knowledge sharing requires the participation of at least two or more stakeholders to achieve the communication goal, while a review activity can be done by a single stakeholder that learns from available knowledge.

Table 2. Knowledge sharing activities

SHARING (*): Make AK available to others		
Activity	Sub-activities level 1	Sub-activities level 2
Review AK	Analyze documents or existing AK stored	Search, Discovery
Communicate AK	Subscribe to AK Organize meetings	Pull/ Push (RSS) Discuss / explain

Complementary to AK producers, knowledge consumers include assessing and learning activities, as shown in tables 3 and 4. Assessment provides the guidelines and recommendations for selecting the best or the optimal decisions among several. The expertise of the architects and the results from evaluating different alternatives usually drive these assessment activities. Table 3 shows different assessment activities and sub-activities to assess before or after decisions are made. Sometimes, assessing about decisions needs from a previous learning activity in order to perform the right assessment. In such scenario we could perform assessment during architecting to select the best decision or during a learning activity to teach about future decisions, as architects can learn from right and wrong experiences. Assessing about AK can be used to know the viability of future decisions and provide further recommendations.

Table 3. Assessment activities with architectural knowledge

ASSESSING (*): Recommends the selection of a decision		
Activity	Sub-activities level 1	Sub-activities level 2
Evaluate	Evaluate impact of implications Constraint analysis Evaluate impact of quality attributes	Analysis of alternatives Simulation Impact analysis
Review	Check for completeness and correctness of AK	
Validate	Check decisions against requirements and architectural products Check the integrity of the dependencies between decisions	Traceability
Recommend	Communicate to stakeholders the results of the assessment activity	

The last activity concerns to learning tasks. Architects become more expert consumers of AK as they learn from past experiences. Learning improves also the career of architects from beginners to more expert ones. As a result, future architecting activities are expected to be performed better that initially. As shown in table 4, some learning activities include the evaluation of stored AK as a way to learn which of the decisions made were right or wrong, or to detect inconsistencies in the decision model.

From our point of view, assessment and learning are often intertwined to understand the choices made. The aim of training activities is to teach about past experiences,

Table 4. Learning activities from previous architectural knowledge

LEARNING (*): Understand why decisions were made		
Activity	Sub-activities level 1	Sub-activities level 2
Evaluate stored AK	Compare the decisions to products and requirements Detect wrong decisions or inconsistent AK	Follow trace links Search-Reuse AK
Training	Teaching about past decisions and experiences	Search-Reuse AK Assessment / Learn

but some search could be done to retrieve the decisions made that will be used in learning activities. Some of the sub-activities defined in the tables described before are interrelated or even duplicated because certain tasks in the producer side are enacted in the consumer side and vice-versa. Figure 1 describes the relationships between the activities defined in the tables and different users can participate either as consumers and producers, depending on their specific roles.

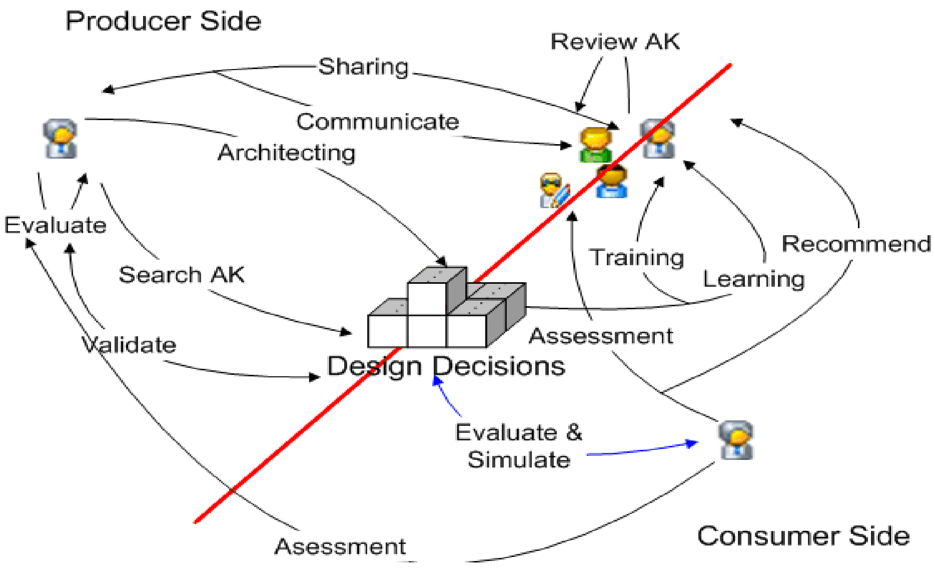


Fig. 1. Activities for producing and consuming architectural knowledge

4 Making AK Explicit with Tool Support

Previous efforts [10] analyzed tool support for design decisions in software architecture. Current technology for supporting such AK is still young and immature, but recent proposals are rapidly gaining popularity to introduce design decisions within the architecting process. Some of the tools that have been recently proposed to store and use design decisions are the following.

Archium (http://www.archium.net) is a research prototype [9] for supporting design decisions as first class entities. Archium defines a meta-model which is composed of three sub-models: *an architectural model, a design decision model, and a composition model* to compose design fragments (an architectural fragment defining a collection of architectural entities). Archium is also a component language which extends Java for describing components, connectors, and design decisions with tool support. Archium integrates an architectural description language (ADL) with Java to describe the elements from a component & connector view but making explicit the architectural design decisions and its rationale [11]. Archium supports the trace from requirements to decisions and is able to check which of these requirements are addressed by one or several decisions. Archium provides visualization facilities for the decisions made using a dependency graph, which can be used to assess about the consequences of the decisions.

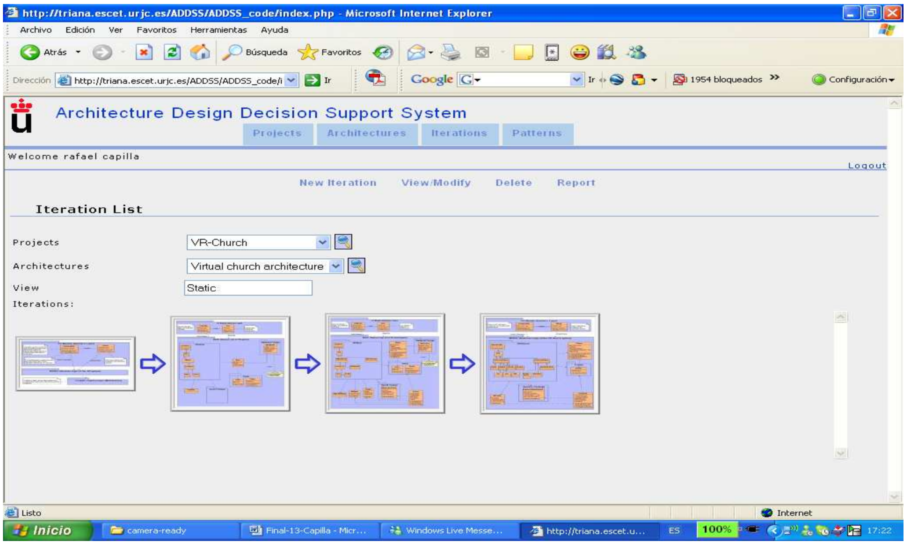
PAKME [1] is a web-based architecture knowledge management tool for providing knowledge management (KM) for software architecture development. PAKME has been built on the top of Hipergate, an open source groupware platform which includes collaborative features, project management facilities and online collaboration tools for decentralized teams. At present, PAKME consists of five components: the *user interface* implemented with JSP and HTML pages, the *KM component* which provides the services necessary to store and update AK, the *search component* which defines three different searching mechanisms (i.e.: keywords, logical operators, and navigation) for retrieving artefacts, the *reporting component* which provides services for representing AK and describing the relationships between different architectural artefacts, and the *repository management* which offers the services needed to maintain the data (currently implemented in PostgreSQL). PAKME uses different templates for capturing and representing the knowledge and the rationale associated to architectural design decisions.

The Architecture Design Decision Support System (ADDSS), available at, (http://triana.escet.urjc.es/ADDSS) [4] is an open web-based tool developed in PHP, HTML and MySQL, and focuses on recording, managing and documenting architectural design decisions under an iterative development process. ADDSS follows the natural way in which architects usually work, that is, creating the architecture under successive iteration for which one or several decisions are made. The design decisions are stored in plain text in MySQL databases. For each set of decisions, an image of the architecture can be uploaded as a thumbnail image. ADDSS does not directly cooperate with other modelling or requirements tools, but it allows uploading images exported with architecture modelling tools. In ADDSS, decisions are motivated by the requirements already stored in the tool. Also, basic dependencies can be established between a decision and previous ones, as a way to create a network of decisions. The result of the decision-making process can be easily visualized and the user can navigate and browse both the resulting architectures and the decisions made. Design decisions in ADDSS decisions can be based on the selection of well-known patterns already stored and a free text description is used to explain the decision made. Finally, PDF documents containing the design rationale of the architecture can be automatically generated using the *fpdf* library for PHP.

4.1 New Features in ADDSS 2.0

The need to count with adequate tool to support new features for characterizing AK, led to evolve the first version of ADDSS. Therefore, we have recently released ADDSS 2.0 with the following additional features respect to the previous version.

- **Visualization capabilities improved:** In ADDSS 2.0, up to 5 architectures are visualized per row showing the thumbnail images of the architectures with the same width, so users can now browse more easily the architectures across the iterations. Figure 2 shows an example of the iterations list.
- **Status of the decisions:** A status can be assigned to each decision (e.g.: pending, rejected, approved, obsolete), so the architect can know which is the current status of that decision in the project.

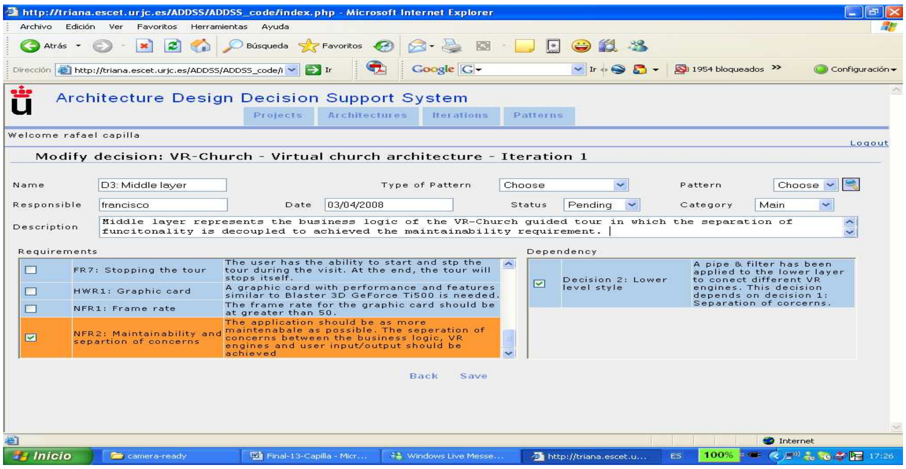

http://triana.escet.urjc.es/ADDSS/ADDSS_code/index.php - Microsoft Internet Explorer
Architecture Design Decision Support System
Projects
Architectures
Iterations
Patterns
Welcome rafael capilla
Logout
New Iteration
View/Modify
Delete
Report

Iteration List

Projects	VR-Church
Architectures	Virtual church architecture
View	Static
Iterations:	

Fig. 2. Iterations list shown the architecture products with ADDSS 2.0

- **Date** of each decision can be added.
- **Support for alternatives decisions:** Decisions can be marked as alternative decisions until the final decision is made (one or more decisions could be the best ones).
- **Tagged requirements** as they have been used by a decision. Therefore, the architect knows at every time the amount of requirements that have been addressed during the architecting activity (see Figure 3).


http://triana.escet.urjc.es/ADDSS/ADDSS_code/index.php - Microsoft Internet Explorer
Architecture Design Decision Support System
Projects
Architectures
Iterations
Patterns
Welcome rafael capilla
Logout
Back
Save

Modify decision: VR-Church - Virtual church architecture - Iteration 1

Name	D3: Middle layer			Type of Pattern	Choose	Pattern	Choose
Responsible	francisco	Date	03/04/2008	Status	Pending	Category	Main

Description: Middle layer represents the business logic of the VR-Church guided tour in which the separation of funcitonality is decoupled to achieved the maintainability requirement.

Requirements

☐	FR7: Stopping the tour	The user has the ability to start and stp the tour during the visit. At the end, the tour will stops itself.
☐	HWR1: Graphic card	A graphic card with performance and features similar to Blaster 3D GeForce Ti500 is needed.
☐	NFR1: Frame rate	The frame rate for the graphic card should be at greater than 50.
✓	NFR2: Maintainability and separtion of concerns	The application should be as more maintenabale as possible. The seperation of concerns between the business logic, VR engines and user input/output should be achieved

Dependency

✓	Decision 2: Lower level style	A pipe & filter has been applied to the lower layer to conect different VR engines. This decision depends on decision 1: Separation of concerns.

Fig. 3. A design decision with its date, status, the requirements that motivated the decision; and a dependency link to a previous decision

- **Category of the decision:** A category attribute discriminate between main, alternative, and derived decisions. A derived decision has a parent decision.
- **PDF documentation improved:** The documentation generated by ADDSS 2.0 details the relationships between requirements, decisions and architectures to follow more easily the trace links. PDF documents describe explicitly the chain of the links between different decisions, so we can easily know which decisions depend from other decisions.
- **User interface improved** (e.g.: menu options, colours).
- **Support for different stakeholder roles**.
- **Pattern classification into different categories:** Pattern search is now more easy and intuitive for the architect.
- **Support for different architectural views:** Now we provide support to define different architectural views and make decisions for each single view.
- **Knowledge search:** In addition to browsing patterns and navigating across the decisions made, a query module extracts relevant information about the decisions made following the links between requirements, decisions, and architectures. For instance, we can extract the requirements and the architectures affected by a particular decision, or we could even know the decisions that affect a particular architecture product.

4.2 Decision-Making Process with ADDSS 2.0

According to the activities described in tables 1 to 4, this section describes which of these are implemented in ADDSS 2.0. Table 5 shows in yellow the activities currently supported by ADDSS 2.0. Those activities marked with "+" can be supported by ADDSS and they have been added with respect to the initial classification of section 3 as a refinement of similar tasks. Also, those processes marked inside a dotted box are not directly supported by ADDSS 2.0 (we don't have an explicit attribute to record such information or process implemented to provide some degree of automatic support), but the result of these activities can be stored as part of the description of the decision as a free text description. The remainder activities are not supported by the tool. The tool provides a semi-automatic support to manage the tacit knowledge and make it explicit to users. The explanation of the activities of table 5 supported by ADDSS 2.0 is as follows. During the architecting process, ADDSS 2.0 records the decisions and assigns to them a status as well as other items like the date and the responsible of the decision. The architect can tag a decision as alternative, derived, or main (the selected decision). This reasoning process implies to consider the pros and the cons of any decision, as well as constraints and dependencies between decisions. The reuse of existing AK is limited by this moment to design patterns previously stored. Reusing previous decisions can be done by examining the documentation generated by the tool. The evaluation of the alternatives is externally done but the results are stored in ADDSS in the form as approved or rejected decisions. Users can navigate through past decisions or even query the database to extract trace information between decisions, requirements and architectural products.

Table 5. Decision-making activities which are automatic or manually supported by ADDSS 2.0 to record and document relevant architectural knowledge

Decision-making activities supported by ADDSS 2.0		
Activity	Sub-activities level 1	Sub-activities level 2
ARCHITECTING (*): Creates and stores AK		
Make decision	Reasoning (rationale, motivation) Select the best alternative	Constraint and dependency analysis Make assumptions Evaluate AK Analyze implications Validate before storing
Characterize decision	Assign status and other relevant items	
Store and document decisions		
Evaluate AK	Reuse AK Evaluate alternatives	Search, Discovery Navigate through DD (+) Query DD (+) Assessment / Learn
SHARING (*): Make AK available to others		
Review AK	Analyze documents or existing AK stored	Search, Discovery Navigate through DD (+) Query DD (+)
Communicate AK	Subscribe to AK Organize meetings	Pull/ Push (RSS) Discuss / explain
ASSESSING (*): Recommends the selection of a decision		
Evaluate	Evaluate impact of implications Constraint analysis Evaluate impact of quality attributes	Analysis of alternatives Simulation Impact analysis
Review	Check for completeness and correctness of AK	
Validate	Check decisions against requirements and architectural products Check the integrity of the dependencies between decisions	Traceability
Recommend	Communicate to stakeholders the results of the assessment activity	
LEARNING (*): Understand why decisions were made		
Evaluate stored AK	Compare the decisions to products and requirements Detect wrong decisions or inconsistent AK	Follow trace links Search-Reuse AK Navigate through DD (+) Query DD (+)
Training	Teaching about past decisions and experiences	Search-Reuse AK Assessment / Learn

Sharing activities could be partially supported in ADDSS 2.0 by the analysis of existing PDF documentation or stored patterns as well as codified architectures and decisions.

Assessment activities can be supported using the traceability mechanism to check requirements against decisions and validate the decisions made. Also, the results of an evaluation of the alternatives can be stored using the status attribute, but no support is provided to carry out the evaluation process in itself. The basic dependency model supported by ADDSS serves to establish links between requirements and architectures which becomes useful for maintenance and evolution activities.

Finally, learning activities can be only carried out through out the evaluation of the decisions that have been recorded. We can compare the decisions made against the requirements to know how many of these have been addressed, and also trace such requirements to the architectural products developed in the process. The documentation generated by the tool shows the chain of the links between decisions as a way to track manually root causes or even known the implications in the architecture when requirements changes.

Otherwise, inconsistencies or wrong decisions may cause to remove a decision or to mark this as wrong. One key aspect not currently supported happens when we remove a decision. ADDSS does not warn about the consequences of removing a decision, which may cause a broken link in the dependency network. Detecting wrong and inconsistent knowledge is still a challenge to face.

4.3 Impact on Traditional Architecting Activities

Software architecting is considered a formal software engineering approach aimed to create and maintain the architecture of a software system over time. Complex and less complex approaches in combination with other software engineering practices are often used to achieve a balance between the more formal activity of well established methods and the agility required to meet the project schedule. In close relationship to this, the introduction of a complementary and concurrent activity like the creation and use of architectural design decisions with specific tool support changes the traditional way in which software architects do their job. By making explicit the process that records the tacit knowledge residing in the architect's mind, we clearly overload the effort spent by architects in the traditional modeling activity. Recording the design decisions introduces an extra effort in architecting, but a significant reduction should be expected during the system maintenance and evolution, as software architects will be able to replay past decisions as well as to avoid other maintenance tasks like architecture recovery or reverse engineering processes. With ADDSS 2.0 we have tried to balance the processes aimed to store and use architectural knowledge with respect to the more traditional architecting activity. Because ADDSS 2.0 is not integrated with other modeling tools like Rationale Rose, decisions can be stored in parallel at the same time the designers use these modeling tools to depict the architecture. In figure 4 we represent the influence of design decisions in the potential overhead and reduction effort in architecture development and maintenance phases.

Initially, architects spend a certain effort in creating the architecture during several project iterations (It), and some additional effort has to be made to create the design decisions (DD) including evaluation, assessment, pattern usage, etc. During any maintenance activity, new decisions have to be made while others can be reused (hexagons in figure 4). For instance, the architecture of iteration It6 is the result of a reused decision and a new one. Hence, the effort spent in re-architecting the system is expected to be lower than if decisions were never recorded. Computing this effort is quite important to estimate how much effort can be saved.

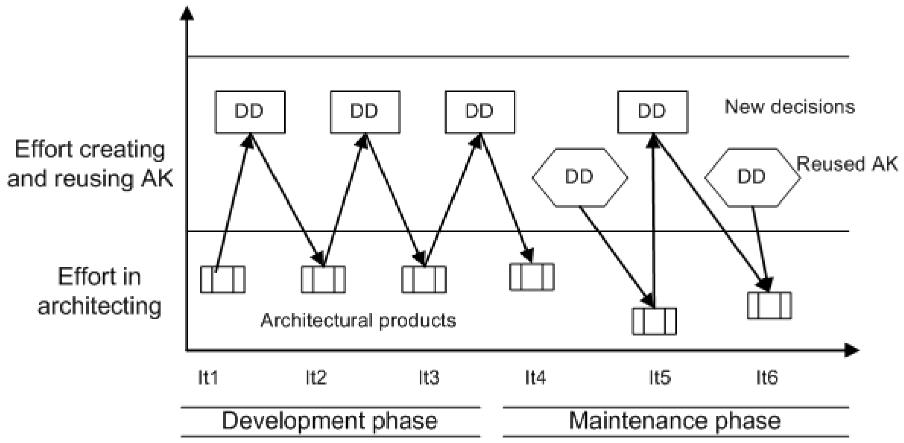


Fig. 4. Effort overview extending the traditional architecting activity with explicit decision-making processes for recording and using architectural design decisions

5 Conclusions and Future Work

As mentioned in [20], "c*reating and maintaining this rationale is very time-consuming*". At present, we have no empirical data concerning the overhead associated with recording and using architectural design decisions. Because ADDSS 2.0 has just been released, we only have the results from a previous evaluation done with ADDSS 1.0, in which 22 master students participated in the evaluation of a small-medium size project. The students were organized in teams of two persons and they spent around 20 hours to record the decisions of a small virtual reality system which has been modeled using Rational Rose and MagicDraw. Because ADDSS 1.0 has limited features (e.g. no support for decision status or alternative decisions) compared to version 2.0, the main results from the evaluation forms and interviews with the team members can be summarized as follows. Most of the teams perceived ADDSS as easy to learn and use, and they have praised ADDSS for understandability. Also, depending on experience of the teams, 4 teams spent around 20 hours while 3 teams spent between 7 and 10 hours, and 4 teams took less than 7 hours using the tool. The average time spent by the teams on recording the design decisions was about 10 hours (it does not comprise the traditional modeling activities). Finally, the average scores of the evaluation of ADDSS by the teams ranged between 5 and 10 points in a scale from 0 to 10, except the learning effort that was around 4 points. With respect to the traditional approach, the teams perceived they needed some extra effort to record and maintain the decisions stored in ADDSS 1.0, but we didn't perform cross-comparison creating the same architecture without using ADDSS.

At present, we have performed just one experiment to estimate the overhead associated with recording design decisions. For the next months we expect to have some additional measurable data using ADDSS 2.0 to evaluate the improvements made and estimate the savings when reusing architectural design decisions. Also, we wan to analyze the barriers and the effort needed as we change the traditional way of architecting when recording decisions in parallel with modeling tasks. Because ADDSS tries to bridge the gap between products and requirements, the maintenance phase can benefit from our approach. Moreover, integration with other popular software engineering tools could reduce the effort in capturing decisions.

Finally, the documentation generated extends the traditional architectural documentation and provides valuable information for different stakeholders who want to learn how the architecture was created. Such information crosscuts the information from other architectural views, such as mentioned in the "decision view" [7], which should be seen as a complementary view to the other traditional ones. ADDSS uses plain text in database fields and PDF documents to store and present the design decisions. However, it is planned to export this information to XML documents in order to facilitate the information exchange with other platforms and tools.

References

1. Babar, M.A., Gorton, I.: A Tool for Managing Software Architecture Knowledge. In: Proceedings of the 2nd Workshop on Sharing and Reusing Architectural Knowledge, ICSE Workshops (2007)
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley, Reading (2003)
3. Bosch, J.: Software Architecture: The Next Step. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 194–199. Springer, Heidelberg (2004)
4. Capilla, R., Nava, F., Pérez, S., Dueñas, J.C.: A Web-based Tool for Managing Architectural Design Decisions. In: Proceedings of the 1st Workshop on Sharing and Reusing Architectural Knowledge. 31 (5). ACM Digital Library, Software Engineering Notes (2006)
5. Capilla, R., Nava, F., Dueñas, J.C.: Modeling and Documenting the Evolution of Architectural Design Decisions. In: Proceedings of the 2nd Workshop on Sharing and Reusing Architectural Knowledge, ICSE Workshops (2007)
6. Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures. Views and Beyond. Addison-Wesley, Reading (2003)
7. Dueñas, J.C., Capilla, R.: The Decision View of Software Architecture. In: EWSA 2005. LNCS, vol. 3047, pp. 222–230. Springer, Heidelberg (2005)
8. Dutoit, A., McCall, R., Mistrik, I., Paech, B. (eds.): Rationale Management in Software Engineering. Springer, Heidelberg (2006)
9. Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: 5th IEEE/IFIP Working Conference on Software Architecture, pp. 109–118 (2005)
10. Jansen, A., Bosch, J.: Evaluation of Tool Support for Architectural Evolution. In: 19th International Conference on Automated Software Engineering (ASE 2004), pp. 375–378 (2004)
11. Jansen, A., van der Ven, J., Avgeriou, P., Hammer, D.K.: Tool Support for Architectural Decisions. In: 6th Working IEEE / IFIP Conference on Software Architecture (WICSA 2007), p. 4 (2007)

12. Kruchten, P.: Architectural Blueprints. The “4+1” View Model of Software Architecture. IEEE Software 12(6), 42–50 (1995)
13. Kruchten, P., Lago, P., van Vliet, H.: T. Building up and Reasoning About Architectural Knowledge. In: Hofmeister, C., Crnković, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, pp. 43–58. Springer, Heidelberg (2006)
14. Lago, P., Avgeriou, P.: ACM SIGSOFT Software Engineering Notes. In: First Workshop on Sharing and Reusing Architectural Knowledge, vol. 3(5), pp. 32–36.
15. Perry, D.E., Wolf, A.L.: Foundations for the Study of Software Architecture, October 1992. Software Engineering Notes, pp. 40–52. ACM SIGSOFT (1992)
16. Roeller, R., Lago, P., van Vliet, H.: Recovering Architectural Assumptions. The Journal of Systems and Software 79, 552–573 (2006)
17. Rozanski, N., Woods., E.: Software Systems Architecture: Working with Stakeholders Using viewpoints and Perspectives. Addison-Wesley, Reading (2005)
18. Tang, A., Babar, M.A., Gorton, I., Han, J.A.: A Survey of the Use and Documentation of Architecture Design Rationale. In: 5th IEEE/IFIP Working Conference on Software Architecture (2005)
19. Tyree, J., Akerman, A.: Architecture Decisions: Demystifying Architecture. IEEE Software 22(2), 19–27 (2005)
20. van der Ven, J.S., Jansen, A.G., Nijhuis, J.A.G., Bosch, J.: Design Decisions: The Bridge between the Rationale and Architecture. In: Rationale Management in Software Engineering, pp. 329–346. Springer, Heidelberg (2006)
21. Wang, A., Sherdil, K., Madhavji, N.H.: ACCA: An Architecture-centric Concern Analysis Method. In: 5th IEEE/IFIP Working Conference on Software Architecture (2005)

A Tool for Supporting Feature-Driven Development

Marek Rychlý and Pavlína Tichá

Department of Information Systems,
Faculty of Information Technology, Brno University of Technology,
Božetěchova 2, 612 66 Brno, Czech Republic
`rychly@fit.vutbr.cz`, `xticha05@stud.fit.vutbr.cz`

Abstract. This paper deals with the Featured Driven Development (FDD), an agile software development method. According to the requirement analysis for the FDD method application, an information system has been created providing all team members with instruments to follow the method. This tool has been implemented as a multi-user web-based application enabling creation of feature lists, planning a project, supporting cooperation among members of a feature-team, and tracking project progress in an illustrative way. To support project management and communication with customer representatives, a wide range of reporting features has been provided.

Keywords: Agile software development, Feature driven development, Feature, Feature-team, Class ownership.

1 Introduction

Traditional software development suffers from slow interaction between the development process and evolving user requirements. It follows from application of traditional software development methods to projects with rapidly changing business requirements. In this context, approaches to software development can be broadly divided into two groups. At one extreme, there are *classical software development methods* where user requirements are obtained in the first phases of the development process and each one of the later phases follows an earlier one (e.g. "the waterfall" model). At the other extreme, there are *agile software development methods* [1], which embrace and promote evolution of user requirements throughout the entire development process (e.g. "eXtreme Programming"). The first extreme produces precisely designed and documented software systems, but those often do not match current user requirements, while the second extreme produces software systems matching the latest user requirements, but often with an inconsistent design and poor documentation.

The *Feature Driven Development* (FDD) [2] is an iterative and incremental software development process. Although the FDD method is one of agile software development methods, it is built around the traditional industry-recognised practices derived from software engineering, including planning, design and documentation phases with fine-grained decomposition of a system's functionality and developers' responsibilities, accurate progress reporting, frequent verification, etc. The application of the FDD method leads to better project management and consistency of a software's design, implementation and documentation.

The paper describes requirement analysis for a software supporting the FDD method [3]. The tool is designed and implemented as an information system providing all team members with instruments to follow the FDD method on real software projects run in a middle-sized software development company. An important feature of the tool is ability of tracking changes in user requirements and map them into modifications in classes and into team members responsible for implementing the changes. Using the feature contributes to an increase in safety of development process.

The remainder of the paper is organised as follows. In Section 2, we introduce the FDD process in more detail. In Section 3, we analyse the FDD process and describe requirements concerning the supporting tool. In Section 4 and Section 5, we describe the design and (briefly) implementation of the tool. In Section 6, we review main approaches that are relevant to the subject and discuss advantages and disadvantages of our system compared with the reviewed approaches. Finally, in Section 7, we summarise our approach, current results and outline the future work.

2 Feature Driven Development (FDD)

The *Feature Driven Development* (FDD) has been published in 1999 [4], after its successful application at an international bank in Singapore in 1997. The FDD is a highly iterative and collaborative agile development method that is composed of *five processes* (see Figure 1). The processes are formally described using the traditional ETVX-based (Entry-Task-Verify-eXit) process descriptions [2]. Informally, the processes can be described as follows:

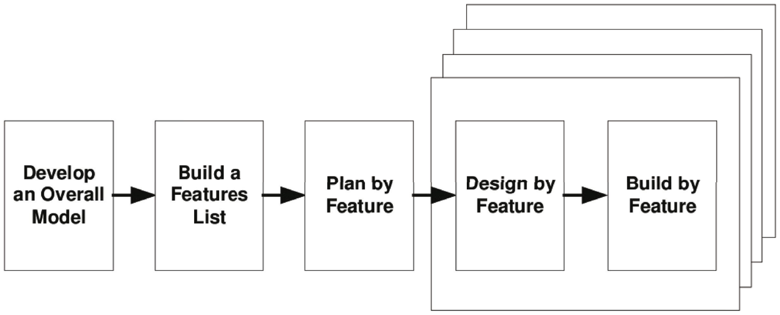


Fig. 1. The five processes of the FDD method (form [2])

Develop an Overall Model – In collaboration with domain experts and developers, an overall *domain object model* is created gradually in series of "walkthroughs" of a software system's scope and context for each area of the problem domain. It captures the key abstractions and their relationships in the system.

Build a Features List – According the initial domain object model, a *list of features* is created where each feature describes an object or its method in the domain model. A "feature" is defined as a small, deliverable client-valued piece of a system's functionality, which can be implemented in no more than 2 weeks. The features are grouped into *feature-sets*, which represent business processes or work-flows, and the feature-sets are grouped into *subject-areas* (or *subject-domains*), which represent business functions or business domains implementing core capabilities of the system. There are recommended *formats for descriptions* of the features, feature-lists and subject-areas, which facilitate its mapping into objects and methods [4]. The recommended format of the descriptions is

- for a feature: *action* the *result* (by, for, of, to, ...) a(n) *object* [(of, for, with, ...) *parameters*], e.g. "verify the password for an user with the login",
- for a feature-list: *action* (for, of, ...) a(n) *object*, e.g. "authentication of an user",
- and for a subject-area: *object* management, e.g. "user management".

Plan By Feature – The feature-sets and features are analysed and their time intensities are estimated. The feature-sets are sorted according to priorities assigned by a customer's representatives, estimated time intensities and technical dependencies. Then they are assigned to individual ad-hoc *feature-teams*. Inside the feature-teams, classes from the domain model are assigned to individual developers. Each *developer is responsible* for creation and maintenance of his classes. The developer is a member of all feature-teams, which have assigned the features related to the developer's classes.

Design By Feature and Build By Feature – Those two processes are iterated for each work package, i.e. for a small group of features from one feature-set. The work package is processed by one feature-team and it is a unit of integration with other feature-sets and feature-teams. Members of the feature-team collaborate to create sequence diagrams and another useful design models for their features, and design interfaces and declarations for corresponding classes and methods. After that, individual developers implement and integrate their classes or parts of the classes into a system.

The FDD processes uses software engineering "best practices" such as domain object modelling (the model is a primary representation of knowledge), developing by feature (iterative and incremental), individual class ownership, ad-hoc set up teams of developers, inspection and reviews (of an overall model, feature-lists, design models and a code), regular builds and verification by a customer's representatives, progress reporting, etc.

2.1 Roles and Responsibilities in the FDD Processes

The FDD method defines more roles than many of other agile methods [1]. The roles can be classified into three categories: key roles, supporting roles and additional roles [2]. One team member can act as multiple roles, and a single role can be shared by several team members.

The six *key roles* in a FDD project are: project manager (the leader of a project), chief architect (overall design of a system), development manager (the coordinator of teams), chief programmer (the leader of a feature-team preparing work packages), class owner (designer, coder, tester and documentarist of its classes) and domain experts (detailed knowledge of user requirements and problem domains).

The five *supporting roles* comprise release manager (controls progress of the process), language lawyer/language guru (has detail knowledge of programming language or technologies), build engineer (responsible for build process and version management), tool-smith and system administrator (technical support of a project).

The three *further roles* that are needed in some projects are: testers (verify that a system fulfils requirements), deployers (maintain data compatibility and prepare new releases) and technical writers (prepare user documentation).

3 Software Support for the FDD Method

To *support of the FDD method*, we need to track the five processes, which are described in Section 2, from points of view of the roles that are listed in Section 2.1. A software support for the FDD method should control application of the FDD processes to real software projects.

The *first FDD process* is focused on the developing of an overall domain model. According to the original presentation of the FDD method [4], it is recommended to use UML visual models [5] with coloured objects. The initial domain model provides a basis for feature-lists, that will be created in the second FDD process. However, in the most projects, many features are arising during the later processes, especially in the processes "design by feature" and "build by feature" as modifications and extensions of the existing features. This issue forbids description of the domain model in details during the first process and requires modifications of the model in later processes. The software support of the FDD method should allow modification of domain models in relation to feature management.

To support the *second FDD process*, which is aimed at building of feature-lists, a software support for the FDD method needs to keep track of all features in a project and their grouping to feature-sets and subject-areas. Generally, lists of features can be created in two ways: top-down and bottom-up, i.e. as decomposition of subject-areas to feature-sets and then to single features and as composition of features into feature-sets and feature-sets into subject-areas, respectively. The software has to support both ways.

In the *third FDD process*, a timetable for feature-sets and features is created. For each feature, it includes the time of its start and its estimated duration. There are two requirements contrary to each other: keep the start-time and duration of a whole project and permit individual planing of features in scope of feature-teams. To balance those requirements, we split the whole project into two-weeks intervals, which are appropriate to maximal time intensities of features [2]. Then, the feature-teams can plan their features within the assigned two-weeks intervals independently and the time-requirements of the whole project are complied. The supporting system should allow to assign features to intervals and their detailed planing, including control of their dependencies and balancing of a workload.

The *last two FDD processes* are focused on designing and building of individual features in compliance with the principle of feature-ownership. Owner of a feature creates a list of activities leading to implementation of the feature. Each activity can require modification of a class or its part related to the feature. The modification will be done solely by the owner of the class (a developer). This implies also ownership of a part of a project's source code. The software support of the FDD method should track those relationships between features, developers and parts of classes.

Finally, a tool for supporting the FDD method has to provide a set of *visual reports*. The reports should continuously show progress for individual features, feature-sets, subject-areas and a whole project, as well as a proper form of workload overviews for individual developers and feature-teams. This information is important for correct planing decisions.

To summarise, the basic requirements to software support of the FDD method according to its five processes are the following: a support of an initial domain model, tracking of its modification and connections of its parts to individual features; a support of features, feature-sets and subject-areas, and their composition and decomposition (top-down and bottom-up approaches); a support of individual planing of features in scope of feature-teams without substantial impact on duration of a whole project; a support of the principle of feature-ownership and class-ownership and collaboration of the owners of features and classes; and a support of wide range of visual reports required for planing decisions.

4 Design of the Tool for Supporting FDD

The Figure 2 shows the basic user roles, which cover important roles from Section 2.1 according to the requirements for a tool for supporting the FDD method (an information system) mentioned in Section 3. Besides those requirements, we define "a domain" as an independent group of projects supported by the tool, e.g. projects of one software developer company using the tool, if it is provided as an outsourced service.

The *SystemAdministrator* maintains technical issues of the whole information system (the same meaning as in Section 2.1), e.g. controls user rights and domains, while the *DomainAdministrator* maintains technical issues of a domain specific part of the system, e.g. controls users in the domain.The *ProjectAdministrator* is

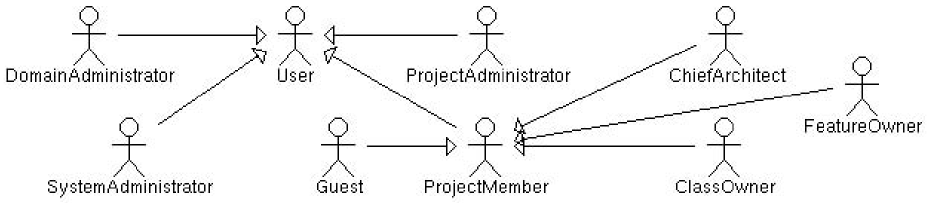


Fig. 2. The hierarchy of the well-established user roles

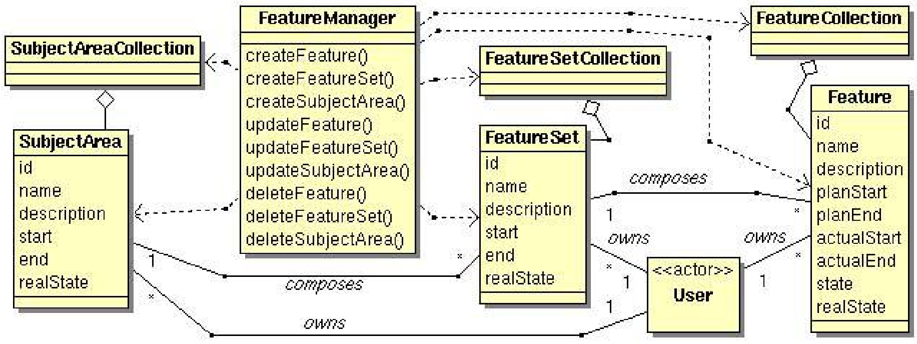


Fig. 3. The class diagram of features and feature-lists

able to create, modify and delete a project, assign and withdraw users and their roles in the project, and obtain reports relevant to the project and its parts. The *ChiefArchitect* can manage a project's domain model and features, feature-sets and subject-areas, and make planing decisions, i.e. assign individual features into two-weeks intervals (see the requirements of the third FDD process in Section 3). The *FeatureOwner* maintains a feature-team, i.e. controls activities leading to implementation of the feature, assigns classes that are modified by the activities and class-owners responsible for realisation of individual activities, watches progress of the activities and verifies finished activities. The *ClassOwner* represents a developer, who implements a part of assigned feature (an individual activity connected to the owned class) and is able to modify information about the progress of the activity. The last role, the *Guest*, represent external supervisor of a project, a customer's representative, who is able to view reports about progress of the project.

In the Figure 3, there is a part of class diagram related to features and feature-lists. The instance of the class `FeatureManager` handles for a project a collection of features (classes `FeatureCollection` and `Feature`) composed into feature-sets (classes `FeatureSetCollection` and `FeatureSet`) composed into subject-areas (classes `SubjectAreaCollection` and `SubjectArea`). Those features, feature-sets and subject-areas have attributes representing scheduled numbers of starting and ending two-weeks intervals. Moreover, the features have attributes indicating actual starting and ending two-weeks intervals and their current

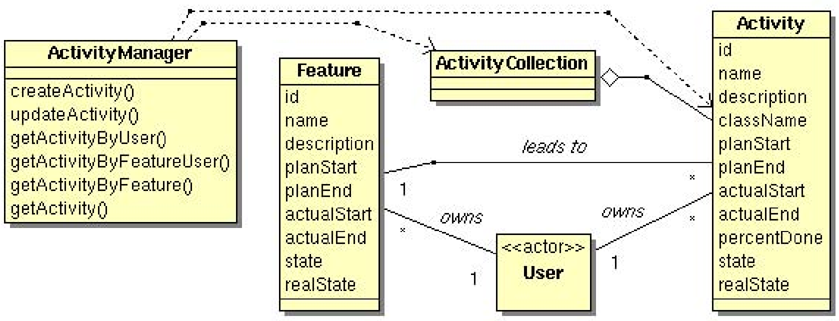


Fig. 4. The class diagram of features and activities

states[1]. Features, feature-sets and subject-areas are connected to users (instances of class `User` with stereotype `Actor`), that act as their owners.

The class diagram in the Figure 4 shows relationship of features and activities and their context. The instance of the class `ActivityManager` handles for a project a collection of activities (classes `ActivityCollection` and `Activity`). Each activity contains attributes that represent its scheduled and actual numbers of starting and ending two-weeks intervals, the name of a class, which is modified by the activity, the progress in percents and current state[1]. Activities are connected to features (instances of class `Feature`) and users (instances of class `User` with stereotype `Actor`), that act as owners of the modified classes during the activities and are responsible for realisation of the activities.

The overall entity-relationship diagram of the system with basic entities is presented in the Figure 5. The diagram connects entities for features and activities to auxiliary entities for domain management (entities `Domain` and `Project`) and user management (entities `User`, `Right`, `Role` and associative entity `UserRoleInProject`). The entities for features and activities (entities `Feature`, `FeatureSet`, `SubjectArea` and `Activity`) have been described before, as the relevant classes in the class diagrams in Figure 3 and Figure 4.

5 Implementation and Practical Results

The system has been implemented in the framework *ASP.NET 2.0* and coded in the C# language as a web-based application for the *Microsoft Internet Information Services* web-server and the relational database management system *Microsoft SQL Server* as a data storage back-end. The external database is accessed via *ADO.NET* and own data abstraction layer, which maps relational data to proper objects and vice versa (see `Manager-` classes in Figures 3 and 4).

The data abstraction layer providing objects based on the relational data also generates some *dynamic attributes* of those objects. A good example is the `realState` attribute of classes `Activity`, `Feature`, `FeatureSet`, `SubjectArea`

[1] The states are: “not started”, “in progress”, “attention” and “completed”.

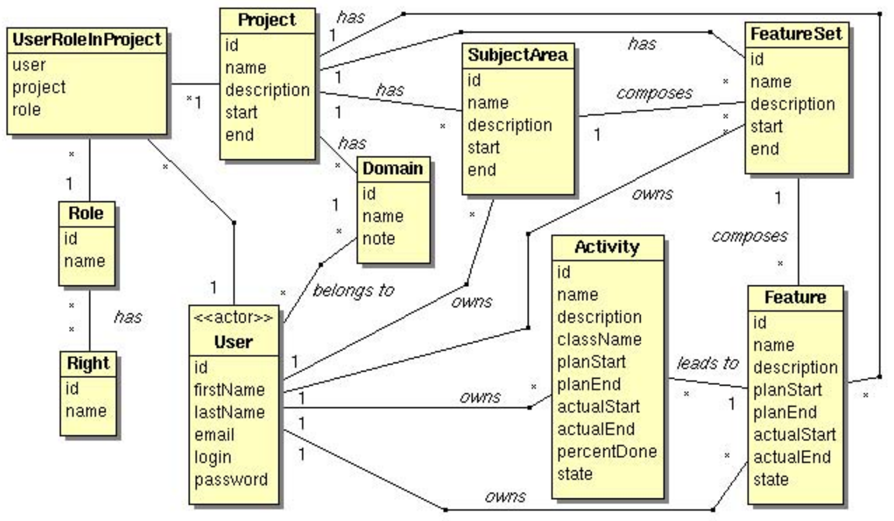


Fig. 5. The entity-relationship diagram with basic entities (the UML notation)

and `Project`. The attribute represents the actual state of an activity, one or more features or a whole project, and can assume values "not started", "in progress", "attention" and "completed", i.e. the same values as `state` attributes. For instances of classes `Activity` and `Feature`, the attribute is computed form attributes `state`, `-Start`/`-End` attributes and the actual two-week interval. The attribute `realState` has value "attention" if the number of the actual two-week interval is greater than `planStart` and `state` is not "in progress" or "completed", or it is greater than `planEnd` and `state` is not "completed", otherwise the attribute `realState` reflects attribute `state`.

Attributes `realState` of instances of classes `FeatureSet`, `SubjectArea` and `Project` are derived in the hierarchy of those objects in the bottom-up direction. For object A (e.g. `FeatureSet`), which composes from objects $B_1, \ldots, B_n$ (e.g. `Feature`-s), attribute `realState` of A has value "attention" or "in progress" if at least one of attributes `realState` of $B_1, \ldots, B_n$ has value "attention" or "in progress"[2], respectively, otherwise the attribute of A reflects values of the attributes of $B_1, \ldots, B_n$.

The result of such "automatic state analysis" of a project's parts can be reported by the system as the project park diagram[3] (see Figure 6). Moreover, values of the `realState` attributes of objects in time can be aggregated through a whole project into the project development roadmap. The roadmap indicates a development plan and real states of completed features in percentages on Y-axis and in time on X-axis of the graph (see Figure 7).

[2] If $B_1, \ldots, B_n$ are "completed" but at least one "not started", A is also "in progress".

[3] The values of progress bars in the boxes of the project park diagram representing `Feature`-s are computed as arithmetical averages of `percentDone` attributes of `Activity`-ies, which are grouped in the boxes at this level.

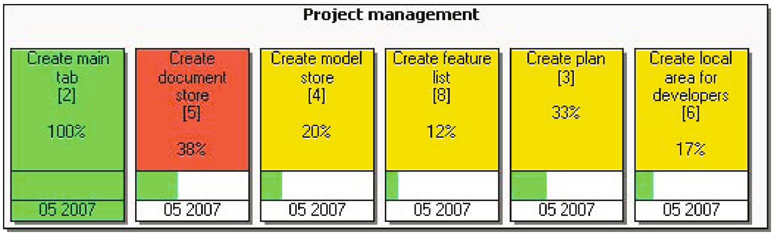

Project management
Create main tab [2] 100% 05 2007
Create document store [5] 38% 05 2007
Create model store [4] 20% 05 2007
Create feature list [8] 12% 05 2007
Create plan [3] 33% 05 2007
Create local area for developers [6] 17% 05 2007

Fig. 6. The project park diagram example where colours of the boxes represent `realState` attributes of a project's parts (a part of real report from the presented tool)

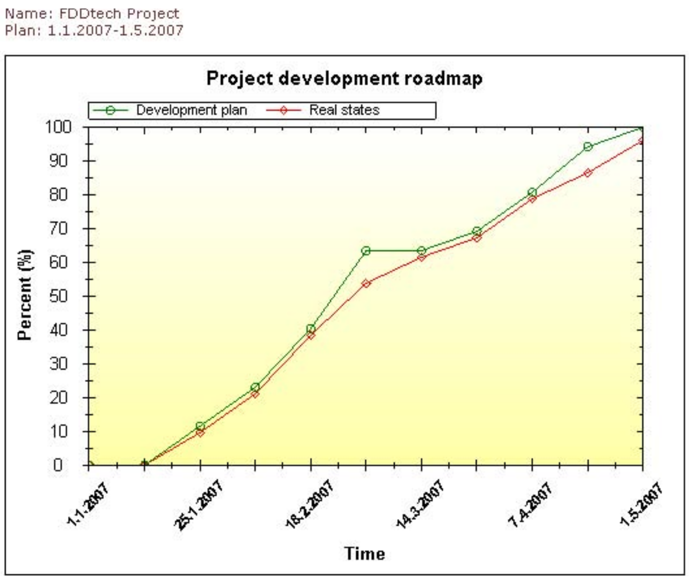


Fig. 7. The project development roadmap (a part of real report from the tool)

The system can export also other reports (e.g. summary progress and trend reports). All reports are available in HTML, PDF and RTF formats, suitable for project managers as well as a customer's representatives.

The source code of the described tool for supporting the FDD method is licensed under the GNU General Public License (GNU GPL) and will be available as an open source project[4].

6 Discussion and Related Work

In this section, we briefly review four projects that support the FDD method of software development and compare them with our approach. Table 1 compares

[4] See `http://www.fit.vutbr.cz/homes/rychly/fdd-tech/`

Table 1. The comparison of the tools that support the FDD method of software development (notes: *the names marked by a star are open source projects)

Name	**Type**	**Users**	**Feature (de)composition**	**Progress reporting**
*FDD Tools Project**	desktop	single user, projects	features, feature-sets, subject-areas (top-down)	project park diagram (interactive)
FDD Tracker	desktops (shared DB)	roles, projects, domains	work-packages, features, feature-sets, subject-areas (top-down)	project park diagram, plan view report, progress summary report, project dev. roadmap (overall and weekly), defect graph
Cognizant FDD	Visual Studio TF Server RTM	roles, projects	modifications of components, features, feature-sets, subject-areas (top-down and bottom-up)	progress report, defect report, prioritised feature list, component ownership matrix
*FDDPMA**	web-based	roles, projects	work-packages, features, feature-sets, subject-areas (top-down)	project park diagram, progress summary report, project development roadmap, plan view report, feature completion trend
*Our tool**	web-based	roles, projects, domains	work-packages, activities (modifications of classes), features, feature-sets, subject-areas (top-down and bottom-up)	project park diagram, progress summary report, project development roadmap, trend report

the basic features of the reviewed projects[5], which are described in more detail bellow.

The *FDD Tools Project* [6] is an open-source Java-based desktop application providing only basic support for the FDD method. It provides one progress report for features (a project park diagram) and there are no means of decomposition of the features into partial tasks. The project seems not to be actively developed anymore.

The *FDD Tracker* [7] is a commercial desktop application executable on Microsoft Windows NT-based operating systems. It provides complex multi-user multi-domain support of the FDD method including support of different roles and views, support of intervals of variable length, inspection management, defect tracking and reporting, etc. The FDD Tracker does not support web-based user interface, does not allow bottom-up composition of features into feature-lists and does not connect parts of features to parts of a source code.

[5] The names of progress reports are not standardised and each project uses its own terminology. In the table, we rename some of the progress reports according their formats and provided information in order to allow direct comparison of the projects.

The *Cognizant Feature-Driven Development* [8] is a commercial tool for supporting the FDD method, which is integrated in Microsoft Visual Studio Team Foundation Server RTM. It allows defining of a software project in Visual Studio as a collection of individual features with connection to the project's source code and tracking of its defects. Cognizant FDD extends five processes of the FDD method with a new process "certify by feature", which follows the fifth FDD process "build by feature" in each iteration, and a new iterative process "release" closing the whole development process [9]. Drawback of Cognizant FDD can be its tight integration with Visual Studio, which prevents managers (non-developers) and a customer's representatives from interaction with products of the FDD method.

The *FDD Project Management Application* (FDDPMA) [10,11] is open-source Java-based application and the only web-based tool among reviewed projects. Unfortunately, the FDDPMA does not allow bottom-up composition of features into feature-lists and does not support relationships between features and classes or their parts (i.e. no class ownership).

In comparison of our tool with the reviewed projects, we can find many similar features that are recommended by the FDD method description (see Section 2). In addition to that, our tool supports top-down and bottom-up approaches to creation of feature-lists, provides decomposition of features into activities, which represent tasks needed to accomplish a feature, connection of the activities and relevant classes or their parts that must be implemented by developers to complete an activity, and detailed hierarchical tracking of the state and progress of features from developers (i.e. activities) to managers (i.e. feature-sets and a project). Usage of the web-based user interface allows easy remote access to project data, especially for domain experts and a customer's representatives that should participate in a project. Drawback of our system can be isolation from other development tools and absence of advanced FDD tools such as tracking of features verification process and defects or support for feature-teams collaboration.

7 Conclusion and Future Work

The paper describes requirements analysis, design, and implementation of a tool supporting the FDD method. The described system covers the five FDD processes, features and feature-lists management, support for decomposition of the features to activities connected to individual classes or their parts, support for feature-owners and class-owners, control of progress at different levels of hierarchy, user management with well-established and user-defined roles, and support of various reports (the park diagram, the project development roadmap, summary progress and trend reports). Some of those features (e.g. support for activities) are novel and have not been used yet.

Incremental development of a software system with strictly defined features and related modifications of parts of classes (i.e. mapping of user requirements into modifications of a source code) allows better tracking of impact of individual increments on quality of a whole software system and contributes to an increase in safety of development process.

Future work is mainly related to integration of human-resource management into the tool (e.g. appointment of feature-team members according to their past experiences and current workload) and support for source code management (i.e. more detailed tracking of source code modifications caused by a feature realisation).

Acknowledgement. This research has been supported by the Research Plan No. MSM 0021630528 “Security-Oriented Research in Information Technology”.

References

1. Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J.: Agile software development methods: Review and analysis. VTT Publications 478, Espoo, Finland: Technical Research Centre of Finland (2002)
2. Palmer, S.R., Felsing, M.: A Practical Guide to Feature-Driven Development. Prentice Hall PTR, Upper Saddle River (2002)
3. Tichá, P.: Information system supporting Feature Driven Development. Master’s thesis, Brno University of Technology, Faculty of Information Technology, Department of Information Systems (May 2007)
4. Coad, P., Lefebvre, E., Luca, J.D.: Feature-Driven Development. In: Java Modeling in Color with UML: Enterprise Components and Process, ch. 6, Prentice Hall PTR, Upper Saddle River (1999)
5. OMG: UML superstructure specification, version 2.0. Document formal/05-07-04, The Object Management Group (August 2005) Also available as ISO/IEC 19501:2005 standard
6. SourceForge.net: FDD tools project (September 2006), `http://fddtools.sf.net/`
7. IT Project Services: FDDTracker (2007), `http://www.fddtracker.com/`
8. Cognizant Technology Solutions: Cognizant Feature-Driven Development (2007), `http://www.cognizant.com/html/content/microsoft/techfddvsts.asp`
9. Cognizant Technology Solutions: Implementing Cognizant Feature-Driven Development using Microsoft Visual Studio Team System. Technology white-paper, Cognizant.NET Center of Excellence (2005)
10. FDDPMA Development: FDD project management application (2007), `http://www.fddpma.net/`
11. Khramthchenko, S.: A project management application for Feature Driven Development (FDDPMA). Master’s thesis, Harvard University (June 2005)

In-Time Role-Specific Notification as Formal Means to Balance Agile Practices in Global Software Development Settings

Dindin Wahyudin[1], Matthias Heindl[2], Benedikt Eckhard[1], Alexander Schatten[1], and Stefan Biffl[1]

[1] Institute of Software Technology
and Interactive Systems,
Vienna University of Technology,
Vienna, Austria
{Dindin,Eckhard,Schatten,Biffl}@ifs.tuwien.ac.at
[2] Support Center Configuration Management Siemens Program and Systems Engineering,
Vienna, Austria
{matthias.a.heindl}@siemens.com

Abstract. In global software development (GSD) projects, distributed teams collaborate to deliver high-quality software. Project managers need to control these development projects, which increasingly adopt agile practices. However, in a distributed project a major challenge is to keep all team members aware of recent changes of requirements and project status without providing too little or too much information for each role. In this paper we introduce a framework to define notification for development team members that allows a) measurement of notification effectiveness, efficiency, and cost; b) formalizing key communication in an agile environment; and c) providing a method and a tool to implement communication support. We illustrate, with an example scenario from an industry background, the concept and report results from an initial empirical evaluation. From the evaluation it follows that the concept allows determining and increasing the effectiveness and efficiency of key communication in a global software development project in a sufficiently formal way without compromising the use of agile practices.

Keywords: Software project management, Software process improvement, Methods and tools of software development, Agile practices in global software development, Context-specific notification.

1 Introduction

Today business competition forces highly distributed and global software development (GSD) players to be more responsive and adaptable to uncertainty during development processes (e.g., changes of requirements, technologies implementation;involvement of partners/subcontractors), especially in novel product development [14].

The Agile Manifesto[1], promised that higher customer satisfaction can be achieved by addressing such uncertainty aspects and delivering working software frequently with shorter timescale. However adoption of agile practices such as daily planning, daily synchronization and daily build [7], [16], requires overall more intensive communication and information exchange among project team members regarding project changes when compared with typical plan driven approach. Especially, in the context of GSD projects, effective communication is an important issue as one has to take into account distant locations and different time zones [9].

Usually, in order to communicate a change in requirements and in other project artifacts, a GSD team member who committed the change notifies other team members in some informal way (e.g., by phone). Such an approach requires extra effort and/or results in loss of information or delay. Another common practice is subscribing by team member to particular tools (e.g., a project manager may subscribe to SVN/CVS to be notified about each check-in performed by his/her developers). Although this approach is a cheaper way of notification, it is often that the target user receives too much information and most of them are out of his current work context or interest.

Hence, to effectively manage such an agile and distributed project one has to address issues specific to agility and distribution, i.e. (a) all team members should be aware of relevant project status (b) information supply should meet the current work context of each role, (now it is hard to measure information supply due to informal way of communication between team members in GSD), and (c) cost and effort of key communications should be reduced. To address these issues, we propose a concept of "in-time role-specific notification". In-time and role-specific means delivering the right information to the right person within his/her current work context (context aware). We define notification in a way that allows measurement of its effectiveness, completeness and correctness. We suggest that such an approach can also be used in the agile context. To address the need for effort and cost reduction we extend the functionality of GSD tools by introducing plug-in integration of the tools to support in-time role-specific notification in GSD settings. Moreover, we present scenarios, based on industrial experience, which illustrate the need for in-time role-specific notification.

The remaining part of the paper is organized as follows: Section 2 describes related work on agile methods adoption in GSD settings and issues important when controlling agile GSD projects. Section 3 introduces the research questions and the concept of "in-time role-specific notification". Section 4 presents scenarios from industry background and later, in Section 5, we provide initial evaluation of the concept. Section 6 discusses the initial evaluation results and compares them with related work. Section 7 concludes and outlines future research on in-time role-specific notification that would be needed to better support collaboration in distributed projects.

2 Related Work

Global software development (GSD) projects can benefit from agile practices to react to changing requirements and project circumstances, however to maintain the overview and control of this project extra care has to be taken to maintain the timely communication

[1] http://agilemanifesto.org/principles.html (accessed on 15 August 2007).

between distributed teams and team members. Formalization of key communication and supported by proper infrastructure can take away the burden of communication "work" from team members while maintaining communication effectiveness and efficiency. The key question is what kind of communication can be automated and how tools can support such automation

2.1 Agile Methods Adoption in GSD Settings

Boehm and Turner [2] describes balancing agility with discipline such as introducing agile practices in plan-driven GSD projects may provide complementary values derived from both approaches. As the usage of plan-driven GSD methodologies promise access to larger competence developer pool with lower development costs, and work effectiveness due to time zone exploitation [9]. While agile software development offers several benefits for GSD such as adaptation to changing requirements, higher customer satisfaction, rapid releases, and lower defect rates [1]. However, Boehm and Turner also suggest that,

The key success is finding the right balance between agility and discipline within the development process, which will vary from different project to project according to circumstances and risk involved.

Several literatures in Distributed and Global Software Development report experiences of agile methods practices in distributed project settings. Schawber [16] reports a case study in scaling Scrum for large project in an outsourcing company. He created multiple small to medium size Scrum teams to perform shorter Sprint cycle and shorter daily Scrum meeting in order to reduce deliverable time of software product. Other study by Martin Fowler [7] reports extreme programming (XP) adoption in large distributed project in USA. The projects successfully used practices such as continuous integration to reduce problems with integrating the work across multi-site teams, short iteration, and multiple communications. To keep communication between teams effective yet relatively intensive as required by XP, he employs a "team ambassadors" as communication buffer or team representative to interface with other distributed teams. Nisar et al. [13] and Xiaohu [18] report their experiences in adopting extreme programming (XP) in offshore teams collaborating with onshore consumers. The development work is done in offshore teams with tightly involvement of the onshore customer. Xiaohu further explained the main issue for implementing XP practices was to reduce the communication delay and improve communication quality between the customers and the offshore development team.

All these experience reports conclude that applied agile methods (such as XP and Scrum) can benefit distributed development; however, research is needed to address issues on communication between project team members which is limited andexpensive in GSD settings [14].

2.2 The Needs for Formalization in GSD and Agile Contexts

To deliver high quality software, in GSD projects typically multiple distributed teams work on the software development. During collaboration, the team members spend more than 50% development time for communication [15], and about 70% of this time accounted for cooperative activities [17]. Other studies in distributed software development suggests that direct /face to face communication is very important in uncertain software development such as to fill in activity details, fix mistakes and inaccurate prediction, counter measures for the effect of project changes [9], to address coordination and interdependency issues [5]. Therefore, direct communication limitation and breakdown regarding the recent changes in requirement and project status make critical situation in software development processes. From GSD project management point of view it is very important to provide information that should meet the roles expectation in order to keep the team member aware to current requirement changes and status of development artifacts, and help to support the multi-sites collaboration activities. However as direct communication and frequent formal reporting of performance status in GSD is luxury and somehow very limited, hence depict the need for an approach that can significantly reduce the effort and cost of communication.

One approach is tool supported notification exchanges between teams and team members by a network of notification server as proposed by deSouza et al. [5] which benefit collaborative development such as in GSD by managing interdependencies of task and artifacts [4]. They suggested that the event data flowing in the project system network and work tools encapsulate critical information necessary to improve coordination of activities, and communication. An event and its attributes (such as requirement changes, automatic build) can represent stakeholder interactions or communication during a software project execution. However although notification server propose automation of some key communication in GSD, however deSouza et al. did not mention how to formalize such notification (e.g. notification specification, rules, and model) which is necessary to bring discipline to the automated notification generation processes. We need to formalize in order to reduce cost, effort, and risk such as delay which is necessary in GSD context. On the other hand we should not formalize everything because it reduces the flexibility which is necessary for certain aspects in the project, too costly and not practical. Therefore, it is necessary to balance formalization and flexibility using cost-benefit analysis.

3 The Concept of In-Time Role-Specific Notification to Balance Agile Practices in GSD Settings

This section motivates the research issues and the proposed concept of in-time role-specific notification to address our research question. We also envision the GSD tool support for collaboration of GSD team members, by introducing the integration of plug-in which allows the information exchanges as part of team member work tools.

3.1 Current Reality of Agile-Global Software Development

To examine the cause and effect logic behind current agile adoption in GSD settings, we employed Current Reality Tree suggested (CRT) in Goldratt's theory of constraints [3] as problem analysis tool. CRT begins with identifying the undesirable effects we see in today practices in GSD and trace back to a few root causes, or a

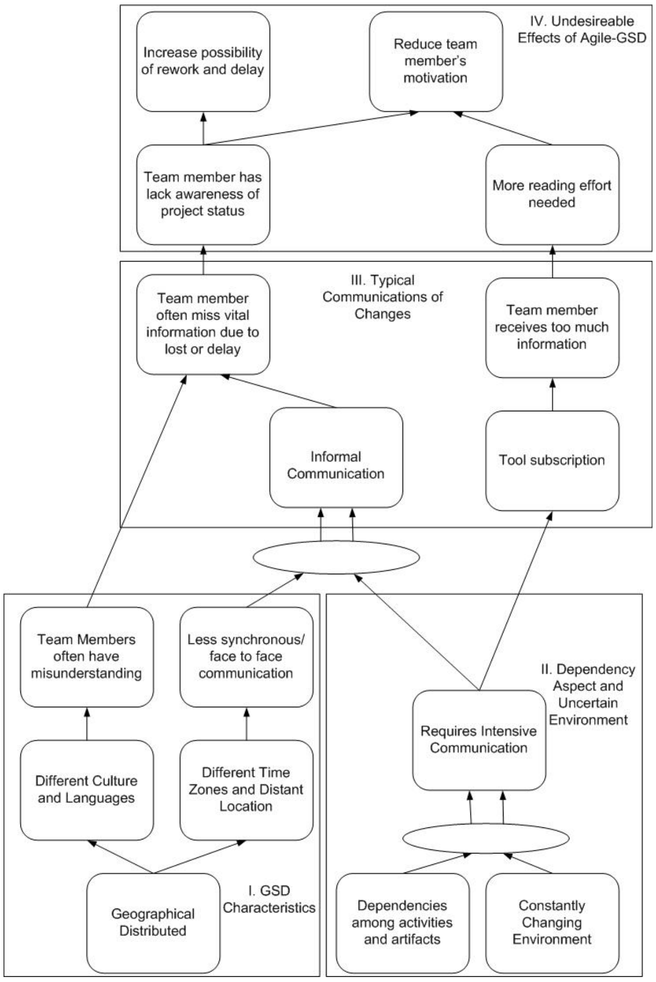

IV. Undesireable Effects of Agile-GSD
Increase possibility of rework and delay
Reduce team member's motivation
Team member has lack awareness of project status
More reading effort needed
III. Typical Communications of Changes
Team member often miss vital information due to lost or delay
Team member receives too much information
Informal Communication
Tool subscription
Team Members often have misunderstanding
Less synchronous/ face to face communication
Different Culture and Languages
Different Time Zones and Distant Location
Geographical Distributed
I. GSD Characteristics
II. Dependency Aspect and Uncertain Environment
Requires Intensive Communication
Dependencies among activities and artifacts
Constantly Changing Environment

Fig. 1. Current Reality Tree of Agile-GSD Project, undesirables' effects such as delay and motivation degradation of developer can be derived from (a) higher effort and higher cost to retrieve information of project status and (b) the poor quality of conveyed information.

single core problem. Later we can select what to improve that will have the greatest positive effect to agile-GSD development. Figure 1 illustrates the current reality of typical Agile-GSD project, the lower level represent the root cause, while the upper level signify undesirable effects.

The rectangles represent entities such as core problem, root cause, effect and undesirable effect, while an ellipse represents AND operator and arrow signify the impact direction.

We grouped the entities into 4 groups to avoid confusion of reader due to number of entity represented in this model. The first group (box I) represents typical characteristics of global software development process as suggested by many literatures in distributed and global software engineering domain such as in [9] [10] and [12]. The distributed participants with different culture, different language may have impact in the content of information being exchanged, as the result team members sometimes have misinterpretation or misunderstanding of the conveyed message, on the other hand the distant location and different time zone, make face-to face communication such as daily synchronization more expensive, worth more effort and hard to coordinate.

The second group (box II), express the need for more intensive communication among team members due to their work dependencies and changing in project environment (e.g. requirement and artifact changes), however as direct communication is infrequent in GSD context, in group 3 (box III) reveals that the communication of changes are committed either in informal way or by subscribing to work tools as described in introduction.

The fourth group (box IV) illustrates the undesirable effects due to current communication methods in Agile-GSD project. As the team member missed vital information this will cause lack of awareness of important project status concerning his work context. This information deficiency may lead team member to perform a task with flaw direction, and increase the possibility of versioning problem, rework and delay. The tool subscribed method, often shower a team member with information spam; consequently he needs more effort to select which information is relevant for his current work context, which sometimes can be a frustrating task. Both of these undesirable effects (lack of awareness and more reading effort) will decrease the developer motivation, and eventually will have larger impact to overall development process.

3.2 Research Issues

Based on Current Reality Tree in section 3.1, direct communications between team members are extensively required by agile methods but missing in GSD due to cost and effort allocation as the result of geographical distribution. Hence, the agile practices adoption in GSD settings will face greater challenge to traditional GSD project.

This hybrid Agile-GSD projects requires a novel method which promise cost and effort reduction in information exchanges between GSD team members. One solution is to automate the communication supported by tools as described in related work, however the challenge is how much formalization of communication is enough, as in agile context, we still need to maintain some aspect of flexibility due to project uncertainty. Therefore in this paper we propose two research questions which are:

(a) What kind of communication can be automated during development processes?
(b) How can tools support such automation?

To address these research issues, we introduce a framework to define notification for development team member which allows:

- Measurement of notification effectiveness, and effort. To determine the effectiveness and effort of key communication and the value of notification in global software development project in formal way without compromising the use of agile practices.
- Formalizing key communication in an agile environment. We provide example scenarios from industry background to explain the concept of formalization of key communication in form of notification exchanges between GSD team members.

- o To provide method and tool support to implement communication support. Tool support to increase the effectiveness and efficiency of key communication in global software development project in formal way without compromising the use of agile practices. We also perform initial empirical evaluation from one of the scenarios as the proof of concept.

3.3 In-Time Role-Specific Notification: Definition and Concept

In global software development setting, collaboration between team members from multiple sites is essential. Figure 2 illustrates the typical work and collaboration in GSD, here a team member first assigned a role within specified work context, e.g. project manager, developer, and tester, in certain location. In agile practices, role assignment may not be a static position, for example a team member can be assigned as software architect at the beginning of the project, later he can act as a developer once the designs and specifications completed.

Based on current assigned role, a team member should perform some activities or task typically supported by a set of work tools to deliver software artifacts. Every change of software artifacts can be considered as an event which is also typically recorded in the work tool where the event happened. Typically works in GSD environment are not stand alone; a team member may have dependencies of artifact developed by other team members. Based on these dependencies, a team member needs to be notified for certain events represent changes of the artifact. Hence he should specify a notification and retrieve the correct notification in time. To receive information which is delayed, partial or not relevant will reduce a team member work performance and also may affect other development tasks performed by other team members who depend on his deliverables, as the consequences the project may face some risky condition such as version conflict, release delay, and quality reduction of to-be delivered software.

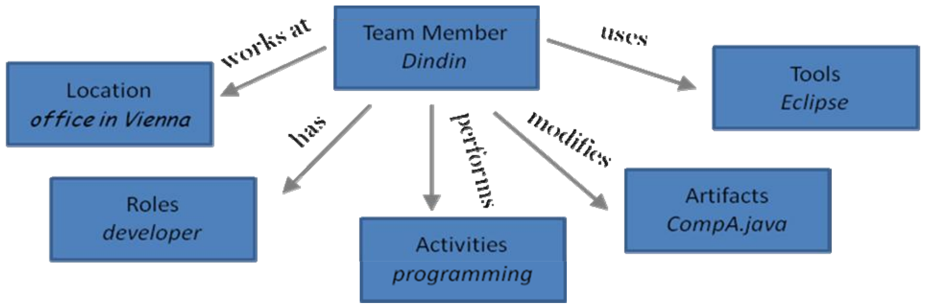


Fig. 2. GSD Team Member role, and the need for notification based on his current work dependencies

3.3.1 In-Time Role-Specific Notification Definition

We define a notification as an object that collects information about status changes, errors, early warnings and other time-relevant project status information to be presented to target roles. A notification can be triggered by events, correlation of events or measurement data passing pre-defined threshold during project execution. For example a tester needs to be notified when a developer closed a development ticket (ticket closing events), which required to be tested before adding the new code-set to current body of code of to-be delivered software.

The meaning of in-time aims to localized notification to meet the user expectation of particular timely effective information awareness, as he may only concern to be notified for relevant project status changes in particular time of deliverable (immediately, or summarized) and within his current work context (e.g. what I'm doing now, with whom/what my work connected with) and consider other out of context and delayed notification as information waste or noise. Meanwhile the role-specific term means to deliver the notification to the right notification user.

3.3.2 Notification Specification: How Much Formalization Is Enough?

The intention to specify a notification is to provide correct notification for target user in formal way. In our context a notification derived from selected key communication between team members. We use three selection criteria to select which key communications are worth enough for formalization and automation by tool support, such as: (a) the key communication is significantly important to support collaboration of GSD team members according to circumstances in development processes; (b) repetitive or frequently occurrences in larger part of development life cycle(e.g. hours and daily occurrence); and (c) data transmitted has significant probability of risks, such as to become lost, error, impartial or delayed in manual way of transmission. Table 1 provides some examples of key communication selection for formalization and automation, these key communications passed the first selection criteria as considered important to support collaboration in GSD.

Based on our Industry background we assumed the values of the selection criteria for each key communication as described in table 1, communication of changes of requirements and components are feasible for formalization and automation. After selection of key communication, the next step is to specify what kind of notification should be provided for target user. The specification also needed to localize the scope of formalization as we only need to formalize several relevant aspect of key communication, and leave the rest to stay flexible.

Table 1. Examples of Key Communication Selection in Agile-GSD settings

Key Communctions	**Roles Involved**	**Frequency of Occurrence**	**Risk of loss and delay**	**Need for Formalization**
Changes in requirements	Project manager, developer, tester	Medium	High	Yes
Requirement traces	Project manager, developer	High	High	Yes
Component changes	Developer	High	High	Yes
Fix defects in code	Developer	High	Low	No
Fill in plan	Project manager, Technical leader, QA	Low	Low	No

There are several elements of key communication that should be formalized to specify a notification such as: processes performed during communication task (e.g. impact analysis of requirement change, decision approval for requirement change), all roles involved in information exchange (e.g. project manager as target user, and developer as events provider in changing requirement scenario, see section 4), data transmitted during communication (e.g. traceability information of requirement), distributed events to publish-subscribed the notification (e.g. source code element changes published by the developer to trigger notification consumed by the project manager), and delay allowance of notification represents the time between artifact/requirement changes and capturing of notification by target user.

The next step of formalization is to model the work-flow to trigger the notification from abovementioned elements. We can use a process centric model such as IDEF0 or extension of UML proposed by Penker and Eriksson [6]. In this paper we use Penker and Eriksson extension to illustrate notification for proposed scenario in section 4, as this extension offers more capability in expressing and formalization of notification by mitigating the ambiguity often associated with narrative specifications or scenarios.

3.3.3 Rules Definition and Notification Escalation

In order to deliver and present notification in-time and within context of particular roles, those we need to formulate the notification rule. The syntax to formulate notification rules consists of the following parts: Notify <whom> in <what way> (e.g., e-mail, SMS, entry in change log) by <when> (e.g., immediately; batch every hour/day) concerning <in which context> (e.g. implement particular task, managing certain project) due to <change event> (e.g. requirement changes, component changes).

Whom: list of persons, roles, or groups. Change can be any observable or derived event or state change regarding an artifact or project state, e.g., some expected event did not happen during the given time window. While context can be any task that assigned to the user, and selected as his current work focus or need to be notified when certain changes occur. For example in requirement changes scenario as described in section 4, a notification for John a developer if particular requirement changed, can be described as: *Notify* John in his Eclipse workspace, *immediately* concerning his task T1 to implement requirement R1 *due to* changes of Requirement R1.

If a condition can not be handled by the system based on the rule set, and then the issue should be escalate to a sufficiently competent role that can provide a reasonable decision. For example in continuous integration build scenario as applied in XP adoption in distributed off-shore project by [13], in this scenario typically a developer will automatic build his code before send it to the repository. For each build he will get notification of build status either success or broken build, however in certain situation such as in an approaching deadline, if a developer experiences too many build failures which is risky situation as there is possibility of he may not deliverer his task on-time. This issue should be escalated to the project manager, so then he can take some appropriate counter measures to address such risk. This example reveals the benefit of notification as early warning sign that may be used to complement information from developer, and to reduce delay for information dissemination.

3.3.4 Derived Measurement

The value of in-time role-specific notification influenced by several factors that can be measured such as:

- **Effort** (*E*) is an accumulation of work hours to prepare (*Tpr*), to process (*Tpc*) and to create notification of changes (*Tcr*). Integrated tools' plug-ins supported notification should be able to reduce significantly the overall effort allocated by the GSD team members.

  Here we can formulate effort as

$$E= Tpr+Tpc+Tc \tag{1}$$

- **Correct Notification** (*CN*) is number of notifications created and transmitted to target user within the scope of pre-defined specification.
- **False Positives** (*FP*) is number of notifications determined not in the scope of correct specified notifications for a target user.
- **False Negatives** (*FN*) indicates number of notifications determined in the scope of correct specified notifications but do not reach target users
- **Effectiveness** (*EF*) is number of correct notifications (*CN*) in proportion to all generated notifications (*GN*) for a specified notification set (*SN*). We expect that tool support increase notification transmission effectiveness as expected in agile context.

  Here we can formulate:

$$EF= CN/GN \tag{2}$$

$$GN = CN + FP+ EF \tag{3}$$

These factors are considered as general measurement of value of notification and should be applicable to almost every scenario in GSD and Agile context. We can use this measurement for balancing agility and formalism in notification, by comparing the results of several alternatives of delivering the notification. For example in scenario described in section 5 we can compare the effort needed by two traditional alternatives of requirement tracing (with Excel and Req.Pro) with our proposed plug-in alternatives, if the results reveal that plug-in offers significant effort reduction with respect to cost to develop such plug-in, then GSD project manager may need to consider to apply such alternatives, on the other hand if the effort reduction is considered not worth enough compare to plug-in development's cost and other set-up effort, then PM may just discard the idea of the plug-in approach.

3.4 Tool Integration and Support

In this work we propose the presentation of notifications in the user interface of a tool routinely used by the target role in order to reduce team member refusal due to "another-tool-syndrome". Tool support mostly consists of tool sets (requirements, development, configuration, tracking and test tools) that can interact in principle providing the basis for redundancy-free, consistent storage of data and exchange of data between tools (via tools interfaces). Tool-based notification also promise cost-reduction which make information exchange can be much more affordable in GSD context, moreover a comprehensive tool support is needed to enable consistent, error-free, and up-to-date information exchange in a GSD context. The interfacing between tools using plug-ins can support information exchange of events recorded by tools during project execution.

Tool support allows to implement notifications using a rule engine, which can be captured and processed into meaningful information or notification using complex events processing techniques [11] e.g., a correlated events processor (CEP). Figure 2 illustrates how GSD work tools can be connected to an enterprise service BUS (ESB) using plug-ins (plug-ins integration). These plug-ins captured particular events occur in the tools, and publish the events to the ESB in XML format. These events later captured and processed by the CEP, and if a measurement threshold or certain rules apposite with an event or correlated events then a notification (also in XML format) is triggered and published to the ESB. Some subscribed tools' plug-ins consumes the notification and presents it to the user as part of their work tools. In summary these plug-ins act as notification or event publisher and as notification subscriber/consumer, and can be configured dynamically by the user (GUI-based configuration for a general user and an event selection pattern language for more sophisticated user).

In continuous integration practices, some activity triggering automation (e.g., automatic build and automatic test) benefit agile software development by reducing effort and time for certain tasks, these activities also may trigger events that considerable worth noting for roles involved in development process such as build status, build error. Correlating these events (e.g. correlating build failures for particular task in certain period of time) can derive time-relevant status information such as quality degradation and quality prediction of software product.

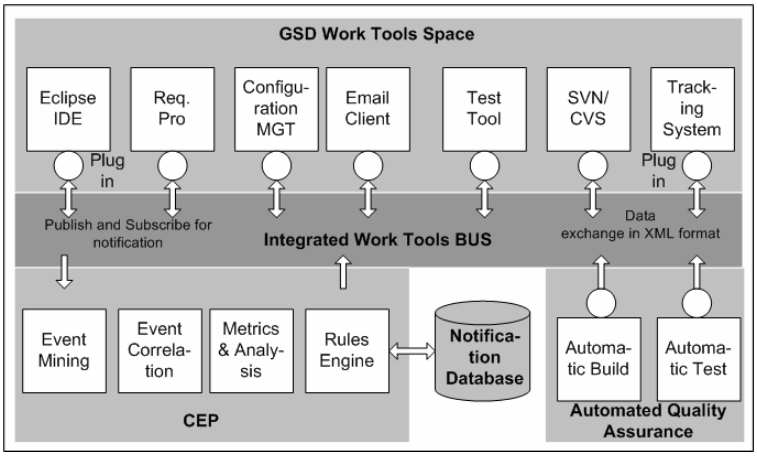


Fig. 3. Integrated tool support for In-time Role-Specific Notification in Agile-GSD settings

4 Example Scenario

The following scenario illustrates how in-time role specific notification provides support to current global software development especially when agile practices introduced to the development processes. We provide initial empirical evaluation based on the result of implementation of the scenario. In this scenario several distributed team members such as a project manager on site A who has responsibility in requirement management which later implemented by the developer from site B. The project manager manages the requirement in requirement management tool such as Requisite Pro, while the developers use IDE tool such as Eclipse as their development platform. If a change of requirement X arrives from the customer, accordingly the project leader performs impact analysis, in order to decide whether such change should be implemented or not (see figure 4), he needs to know the current status from developer who assigned to implement the requirement X, and what kind of impact may derived by this change e.g. risks and cost.

Typically developers in GSD create some *Excel* matrices to store traceability information of implementation status which can be considered as ad-hoc approach or systematically draw a license for the project's requirements management tool (e.g. Req.Pro). Project manager then manually assesses this information, performs the analysis and creates an impact report as the basis of decision approval whether a change should be implemented or not. Based on this scenario, we can define the *impact analysis* as the processes, *project manager* and *developer* as roles involved in this process, and *traceability information* transmitted by the developer as key communication to be automated. Let's assume that we extended the functionality of tools used by developer (Eclipse) and project manager (Req.Pro) with plug-ins to provide interface between the two tools. In this extended scenario whenever a developer committed some changes in his code set, the Eclipse plug-in will store this event and correlate these changes to relevant requirement (Req.X), and automatically publish a requirement traces notification (N) consists of developer ID, source code elements that have been changed, date of changes and its correlation with Req.X. The Req.Pro plug-in which subscribed for this type of notification then captures notification N from the integrated work tools BUS (see figure 3), and present this notification in the project manager's Req.Pro interface.

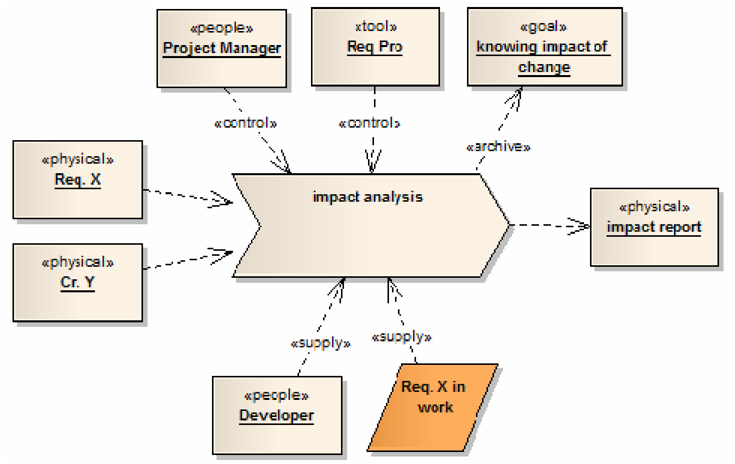


Fig. 4. Impact Analysis is performed by project manager based on requirement traceability information from the developer

Despite of cost and effort reduction, as result of automation, this approach can benefit distributed project controlling as a project managers can decide if a requirement should be changed although development has already been started. They can also easily get in contact with the developer that is working on it to ask him about the current progress or potential implications. As the consequences notification may enhance the impact analysis processes in order to avoid potentially dangerous situation such as to put barrier to the developer against risky or unnecessary changes (as in Scrum before a Sprint release).

5 Initial Empirical Evaluations

We performed an initial feasibility study of the integrated tool plug-in support for scenario of requirement traces to support impact analysis as described. In order to compare the plug-in-based approach with other alternatives, we observed a set of projects at Siemens PSE to evaluate the tracing efforts, correctness and completeness of each alternative. The projects were different in domain (transportation systems, telecommunication, etc.), but similar in size: medium size projects, with 2 to 4 sites (e.g., Austria, Romania, Slovakia), and between 10 and 60 team members.

The number of requirements of each project is between 150 and 300; number of source code methods to be traced range from 6000 to 13000, while number or traces per requirements is between 150 and 300. Based on these project setting factors we compared the effort to trace requirements to source code methods, the completeness and correctness of traces for the tracing alternatives described in section 4. For more detail information and scenario of evaluation can be found in Heindl et al. [8].

Comparison of the 3 alternatives of requirement tracing, reveal that using plug-in alternatives for tracing requirement may significantly reduce the effort of developer teams and increasing completeness and correctness of tracing. Heindl et al., also reported such improvement lead to higher developer motivation, as developer will have more awareness of changes in requirement, lower effort to trace the requirement and more confidence of correctness of trace information, which also reduce possibility of delay or rework.

Table 2. Comparison of Tracing Effort and Tracing Qualities, *source* Heindl et al. [8]

	Effort for tracing (in working hours)		Tracing Qualities	
	For 150 requirements	For 300 requirements	Correctness (%)	False Positives (%)
Ad-hoc	450 to 1350	900 to 2700	5% to 30%	5-10%
Systematic	50 to 167	99 to 334	20% to 40%	10%
Continuous/ Plug in	8 to 26	16 to 53	60% to 75%	5%

In this paper we compare three alternatives of requirement tracing, and to investigate the continuous tracing approach using in-time role-specific notification concept on the effort and quality of traces. However as reported by Heindl et. al, this approach will have greater benefit for medium to large projects, as for smaller projects, the tracing effort might be too high compare to traditional ad-hoc tracing. Automation of notification in this scenario also has to consider the amount of investment needed especially in project with a low number of requirements and requirement changes.

We use requirement tracing scenario for our initial evaluation because we believe that changing of requirements is the most prominent factor in current software development which need more attentions from the development teams.

6 Discussion

From related work, we can conclude that agile practices adoption in GSD settings may provide several benefits needed by current industry. However one challenge is to provide a means of communication and information exchanges between team members concerning occurrence of changes. Referring to our initial research questions, distributed project needs to define some key communications which is feasible for automation in order to reduce cost and effort. In our initial feasibility study we selected traceability of requirement changes as the key communication that can be automated. The integration of plug-in for developer's tool (Eclipse) and project manager's tool (Req.Pro), provide an interface between the tools, which allows automating this key communication.

The framework also allows measurement of the value of notification, as in our initial empirical study we found that integration of tool support significantly reduces the effort for requirement tracing compared to more expensive time consuming alternatives (e.g. using Excel and Requisite Pro) which are commonly used in current GSD projects. However the evaluation of the concept from other context ofagile-distributed development such as different development process scenarios and measuring it's the impact to overall development performance will be further work.

7 Conclusions

In the paper we proposed a concept of role-specific and context-aware notification supported by integrated tools and oriented towards distributed projects. The goal was to complement current distributed project controlling mechanisms and to address communication issues associated with application of agile practices in GSD settings.

Formalization and automation of some key communications between team members in a form of notification may provide benefits such as cost and effort reduction but seems limited to GSD settings. Moreover, we believe that such notification will provide GSD team members with more timely and context-aware information on project status changes. Our initial empirical evaluation provided promising results. However, we would like to perform similar evaluation in the industrial setting with larger size of development team.

Acknowledgments. We would like to thank Franz Reinisch from Siemens PSE Austria and Prof. A Min Tjoa from IFS TU Wien for their contributions to the paper. The paper has been partly supported by The Technology-Grant-South-East-Asia No. 1242/BAMO/2005 Financed by ASIA-Uninet. More details on In-time Role-Specific Notification can be found in our technical report, available at http://qse.ifs.tuwien.ac.at/publications.htm.

References

1. Boehm, B.: Get ready for agile methods, with care. Computer 35(1), 64–69 (2002)
2. Boehm, B., Turner, R.: Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
3. Dettmer, H.: Goldratt's Theory of Constraints: A System Approach to Continuous Improvement. Quality Press (1997)
4. de Souza, C., Redmiles, D., Mark, G., Penix, J., Sierhuis, M.: Management of interdependencies in collaborative software development. In: International Symposium on Empirical Software Engineering, 2003. ISESE 2003, pp. 294–303 (2003)
5. de Souza, C., Basaveswara, S., Redmiles, D.: Supporting global software development with event notification servers. In: The ICSE 2002 International Workshop on Global Software Development (2002)
6. Eriksson, H.E., Penker, M.: Business Modeling With UML: Business Patterns at Work. John Wiley & Sons, Inc., New York (1998)
7. Fowler, M.: Using agile process with offshore development (June 2007), `http://www.martinfowler.com/articles/agileOffshore.html`
8. Heindl, M., Reisnich, F., Biffl, S.: Integrated Developer Tool Support for More Efficient Requirements Tracing and Change Impact Analysis, Technical Report. Institute f. Software Technology and Interactive System, Vienna University of Technology (2007)
9. Herbsleb, J., Moitra, D.: Global software development. Software, IEEE 18(2), 16–20 (2001)
10. Herbsleb, J.D., Paulish, D.J., Bass, M.: Global software development at Siemens: experience from nine projects. In: Inverardi, P., Jazayeri, M. (eds.) ICSE 2005. LNCS, vol. 4309, pp. 524–533. Springer, Heidelberg (2006)
11. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, Reading (2002)
12. Mockus, A., Herbsleb, J.: Challenges of global software development. In: Seventh International Software Metrics Symposium, 2001. METRICS 2001, pp. 182–184 (2001)
13. Nisar, M., Hameed, T.: Agile methods handlfing offshore software development issues. In: 8th International Multitopic Conference, 2004. Proceedings of INMIC 2004, pp. 417–422 (2004)
14. Paasivaara, M., Lassenius, C.: Could global software development benefit from agile methods? In: International Conference on Global Software Development, pp. 109–113 (2006)
15. Perry, D.E., Staudenmayer, N., Votta, L.G.: People, organizations, and process improvement. IEEE Softw. 11(4), 36–45 (1994)
16. Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Prentice Hall PTR, Upper Saddle River (2001)
17. Vessey, I., Sravanapudi, A.P.: Case tools as collaborative support technologies. Communication of ACM 38(1), 83–95 (1995)
18. Xiaohu, Y., Bin, X., Zhijun, H., Maddineni, S.: Extreme programming in global software development. In: Canadian Conference on Electrical and Computer Engineering, 2004, vol. 4, pp. 1845–1848 (2004)

An Integrated Approach for Identifying Relevant Factors Influencing Software Development Productivity

Adam Trendowicz[1], Michael Ochs[1], Axel Wickenkamp[1], Jürgen Münch[1], Yasushi Ishigai[2,3], and Takashi Kawaguchi[4]

[1] Fraunhofer IESE, Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
{trend,ochs,wicken,muench}@iese.fraunhofer.de
[2] IPA-SEC, 2-28-8 Honkomagome, Bunkyo-Ku, Tokyo, 113-6591, Japan
ishigai@ipa.go.jp
[3] Research Center for Information Technology Mitsubishi Research Institute, Inc.
3-6, Otemachi 2-Chome, Chiyoda-Ku, Tokyo, 100-8141, Japan
ishigai@mri.co.jp
[4] Toshiba Information Systems (Japan) Corporation, 7-1 Nissin-Cho,
Kawasaki-City 210-8540, Japan
kawa@tjsys.co.jp

Abstract. Managing software development productivity and effort are key issues in software organizations. Identifying the most relevant factors influencing project performance is essential for implementing business strategies by selecting and adjusting proper improvement activities. There is, however, a large number of potential influencing factors. This paper proposes a novel approach for identifying the most relevant factors influencing software development productivity. The method elicits relevant factors by integrating data analysis and expert judgment approaches by means of a multi-criteria decision support technique. Empirical evaluation of the method in an industrial context has indicated that it delivers a different set of factors compared to individual data- and expert-based factor selection methods. Moreover, application of the integrated method significantly improves the performance of effort estimation in terms of accuracy and precision. Finally, the study did not replicate the observation of similar investigations regarding improved estimation performance on the factor sets reduced by a data-based selection method.

Keywords: Software, development productivity, influencing factors, factor selection, effort estimation.

1 Introduction

Many software organizations are still proposing unrealistic software costs, work within tight schedules, and finish their projects behind schedule and budget, or do not complete them at all [23]. This illustrates that reliable methods to manage software development effort and productivity are a key issue in software organizations. One essential aspect, when managing development effort and productivity, is the large number of associated and unknown influencing factors (so-called *productivity factors*) [27].

Identifying the right productivity factors increases the effectiveness of productivity improvement strategies by concentrating management activities directly on those development processes that have the greatest impact on productivity. On the other hand, focusing measurement activities on a limited number of the most relevant factors (goal-oriented measurement) reduces the cost of quantitative project management (collecting, analyzing, and maintaining the data). The computational complexity of numerous quantitative methods grows exponentially with the number of input factors [4], which significantly restricts their industrial acceptance.

In practice, two strategies to identify relevant productivity factors, promoted in the related literature, are widely applied. In *expert-based* approaches, one or more software experts decide about a factor's relevancy [24]. In *data-based* approaches, existing measurement data, covering a certain initial set of factors, are analyzed to identify a subset of factors relevant with respect to a certain criterion [6, 10]. These factor selection strategies have, however, significant practical limitations when applied individually. Experts usually base their decisions on subjective preferences and experiences. In consequence, they tend to disagree largely and omit relevant factors while selecting the irrelevant ones [24]. The effectiveness of data-based methods largely depends on the quantity and quality of available data. They cannot identify a relevant factor if it is not present in the initial (input) set of factors. Moreover, data analysis techniques are usually sensitive to messy (incomplete and inconsistent) data. Yet, assuring that all relevant factors are covered by a sufficient quantity of high-quality (i.e., valid, complete, and consistent) measurement data is simply not feasible in practice.

In this paper, we propose an integrated approach to selecting relevant productivity factors for the purpose of software effort estimation. We combine expert- with data-based factor selection methods, using a novel multi-criteria decision aid method called *AvalOn*. The presented approach is then evaluated in the context of a large software organization.

The remainder of the paper is organized as follows. Section 2 provides an overview of factor selection methods. Next, in Section 3, we present the integrated factor selection method, followed by the design of its empirical evaluation (Section 4) and an analysis of the results (Section 5). The paper ends with conclusions and further work perspectives (Section 6).

2 Related Work

Factor selection can be defined as the process of choosing a subset of M factors from the original space of N factors (M≤N), so that the factor space is optimally reduced according to a certain *criterion*. In principle, the selection process may be based on data analysis, expert assessments, or on both, experts and data. In *expert-based factor selection*, the factor space N is practically infinite and not known *a priori*. One or more experts may simply select a subset of relevant factors and/or weight factors with respect to their relevancy (e.g., using the *Likert scale* [21]). Factor ranking is equal to assigning discrete weights to them. In *data-based factor selection*, the factor space N is known and determined by available measurement data. Most of the factor selection methods originate from the data mining domain and belong to the so-called *dimensionality reduction* methods [8]. In principle, data-based selection methods assign weights to available factors. Weight may be dichotomous (factor selection), discrete (factor raking), or continuous (factor weighting).

The purpose of factor selection methods in software effort estimation is to reduce a large number of potential productivity factors (cost drivers) in order to improve estimation performance while maintaining (or reducing) estimation costs. Moreover, information on the most relevant influence factors may be used to guide measurement and improvement initiatives. In practice (authors' observation), relevant cost drivers are usually selected by experts and the selection process is often limited to uncritically adopting factors published in the related literature. Software practitioners adopt the complete effort model along with the integrated factors set (e.g., COCOMO [3]) or build their own model on factors adapted from an existing model. In both situations, they risk collecting a significant amount of potentially irrelevant data and getting limited performance of the resulting model.

In the last two decades, several data-based approaches have been proposed to support software organizations that are already collecting data on arbitrarily selected factors in selecting relevant factors. Various data analysis techniques were proposed to simply reduce the factor space by excluding potentially irrelevant ones (factor selection). The original version of the ANGEL tool [19] addressed the problem of optimal factor selection by exhaustive search. However, for larger factor spaces (>15-20), analysis becomes computationally intractable due to its exponential complexity. Alternative, less computationally expensive factor selection methods proposed in the literature include Principal Component Analysis (PCA) [22], Monte Carlo simulation (MC) [12], general linear models (GLM) [13], and wrapper factor selection [10, 6]. The latter approach was investigated using various evaluation models (e.g., regression [6], case-based reasoning [10]), and different search strategies (forward selection [6, 10], as well as random selection and sequential hill climbing [10]). In all studies, significant reduction (by 50%-75%) of an initial factors set and improved estimation accuracy (by 15%-379%) were reported. Chen et al. [6] conclude, however, that despite substantial improvements in estimation accuracy, removing more than half of the factors might not be wise in practice, because it is not the only decision criterion.

An alternative strategy to removing irrelevant factors would be assigning weights according to a factor's relevancy (*factor weighting*). The advantage of such an approach is that factors are not automatically discarded and software practitioners obtain information on the relative importance of each factor, which they may use to decide about the selection/exclusion of certain factors. Auer et al. [1] propose an optimal weighting method in the context of the k-Nearest Neighbor (k-NN) effort estimator; however, exponential computational complexity limits its practical applicability for large factor spaces. Weighting in higher-dimensionality environments can be, for instance, performed using one of the heuristics based on rough set analysis, proposed recently in [11]. Yet, their application requires additional overhead to discretize continuous variables.

A first trial towards an integrated factor selection approach was presented in [2], where Bayesian analysis was used to combine the weights of COCOMO II factors based on human judgment and regression analysis. Yet, both methods were applied on sets of factors previously limited (arbitrarily) by an expert. Moreover, experts weighted factor relevancy on a continuous scale, which proved to be difficult in practice and may lead to unreliable results [24]. Most recently, Trendowicz et al. proposed an informal, integrated approach to selecting relevant productivity factors [24]. They used an analysis of existing project data in an interactive manner to support experts in identifying relevant factors for the purpose of effort estimation. Besides increased estimation performance, the factor selection contributed to increased understanding and improvement of software processes related to development productivity and cost.

3 An Integrated Factor Selection Method

In this paper, we propose an integrated method for selecting relevant productivity factors. The method employs a novel multi-criteria decision aid (MCDA) technique called *AvalOn* to combine the results of data- and expert-based factor selection.

3.1 Expert-Based Factor Selection

Expert-based selection of relevant productivity factors is a two-stage process [24]. First, a set of candidate factors is proposed during a group meeting (brainstorming session). Next, factor relevancy criteria are identified and quantified on the Likert scale. Example criteria may include a factor's impact, difficulty, or controllability. *Impact* reflects the strength of a given factor's influence on productivity. *Difficulty* represents the cost of collecting factor-related project data. Finally, *controllability* represents the extent to which a software organization has an impact on the factor's value (e.g., a customer's characteristics are hardly controllable). Experts are then asked to individually evaluate the identified factors according to specified criteria.

3.2 Data-Based Factor Selection

Data-based selection of relevant productivity factors employs one of the available factor weighting techniques. As compared to simple factor selection or ranking techniques, weighting provides experts with the relative distance between subsequent factors regarding their relevance. Selected weighting should be applicable to regression problems, i.e., to the continuous dependent variable (here: development productivity). Given the size of the factor space, optimal weighting [1] (small size) or weighting heuristics [11] (large size) should be considered. In this paper, we employ the *Regression ReliefF* (RRF) technique [16]. RRF is well suited for software engineering data due to its robustness against sparse and noisy data. The output of RRF (weighting) reflects the ratio of change to productivity explained by the input factors.

3.3 An Integrated Factor Selection Method

Integrated factor selection combines the results of data- and expert-based selections by means of the *AvalOn* MCDA method. It is the hierarchically (tree) structured model that was originally used in COTS (Commercial-of-the-shelf) software selection [15]. AvalOn incorporates the benefits of a tree-structured MCDA model such as the Analytic Hierarchy Process (AHP) [17] and leverages the drawbacks of pair-wise (subjective) comparisons. It comprises, at the same time, subjective and objective measurement as well as the incorporation of uncertainty under statistical and simulation aspects. In contrast to the AHP model, which only knows one node type, it distinguishes several node types representing different types of information and offering a variety of possibilities to process data. Furthermore, AvalOn offers a weight rebalancing algorithm mitigating typical hierarchy-based difficulties originating from the respective tree structure. Finally, it allows for any modification (add, delete) of the set of alternatives while maintaining consistency in the preference ranking of the alternatives.

3.3.1 Mathematical Background of the AvalOn Method

As in many MCDA settings [25], a preference among the alternatives is processed by summing up *weight x preference* of an alternative. In AvalOn (Fig. 1), this is accomplished for the node types *root*, *directory*, and *criterion* by deploying the following abstract model in line with the meta-model structure:

$$pref_i(a) = \sum_{j \in subnodes(i)} w_j \cdot pref_j(a),$$

where i is a node in the hierarchy, a the alternative under analysis, *subnodes(i)* the set of child/sub-nodes of node i, $pref_j(a) \in [0..1]$ the preference of a in subnode j, and $w_j \in [0..1]$ the weight of subnode j. Hence $pref_i(a) \in [0..1]$.

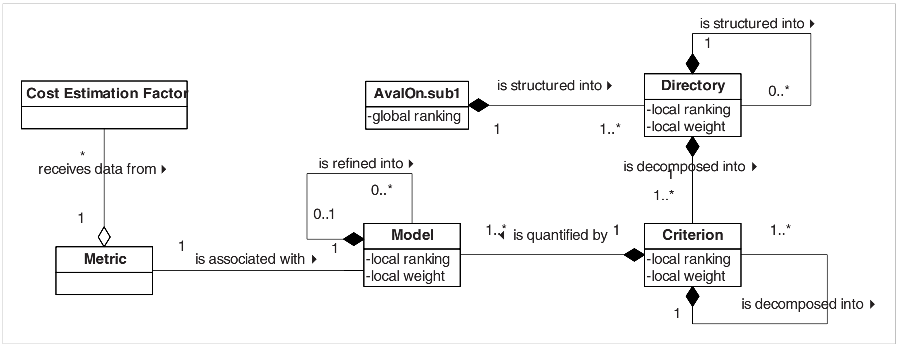


Fig. 1. Meta-model for factor selection

In each model, a value function (*val*) is defined, building the relation between data from {*metrics x alternatives*} and the assigned preference values. *val* may be defined almost in an arbitrary way, i.e., it allows for preference mappings of metric scaled data as well as categorical data.

In this way, *val* can model the whole range of scales from semantic differential, via Likert to agreement scales. Please note that when calculating $pref_i(a)$ on the lowest criterion level, the direct outputs of the function *val* in the subnodes, which are *models* in this case, are weighted and aggregated. The full details of the general model definition for *val* is described in [18]. In this context, two examples for *val*, one metric scaled (figure on the left), and one categorical (figure on the right), are given in Fig. 2. On the x-axis, there are the input values of the respective metric, while the y-axis shows the individual preference output *val.* A full description of the AvalOn method can be found in [15, 18].

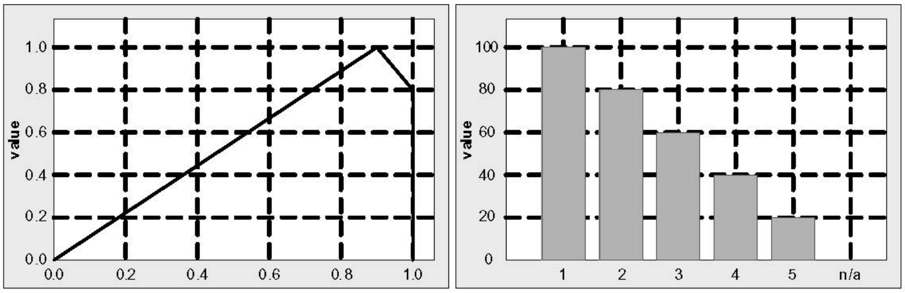


Fig. 2. Example *val* models

3.3.2 Application of the AvalOn Method

AvalOn allows for structuring complex information into groups (element *directory* in the meta-model) and criteria (element *criterion* in the meta-model). Each directory as well as each criterion may be refined into sub-directories and sub-criteria. Each (sub)-criterion may then be refined into individual model(s) and sub-models. The models transform the measurement data coming from each alternative into initial preference values. The models providing the preferences based on each measurement by alternative are associated with a set of previously defined metrics. Bottom-up, the data coming from each alternative to be potentially selected (here: *productivity factors*) are then processed through the models and aggregated from there to the criteria and directory level(s). Finally, in the root node (here: *AvalOn.sub1*), the overall preference of the productivity factors based on their data about individual metrics is aggregated using a weighting scheme that is also spread hierarchically across the tree of decision (selection) criteria. The hybrid character of the setting in this paper can be modeled by combining expert opinion and objective data from, e.g., preliminary data analyses, into criteria and models within different directories, and defining an adequate weighting scheme.

4 Empirical Study

The integrated factor selection method proposed in this paper was evaluated in an industrial context. We applied the method for the purpose of software effort estimation and compared it with isolated expert- and data-based selection methods. Data-based factor selection employed the RRF technique [16] implemented in the WEKA data mining software [26]. Expert-based factor selection was performed as a multiple-expert ranking (see Section 4.2 regarding the ranking process).

4.1 Study Objectives and Hypotheses

The objective of the study was to evaluate in a comparative study expert- and data-based approaches and the integrated approach for selecting the most relevant productivity factors in the context of software effort estimation. For that purpose, we defined two research questions and related hypotheses:

Q1. Do different selection methods provide different sets of productivity factors?

H1. Expert-based, data-based and integrated methods select different (probably partially overlapping) sets of factors.

Q2. Which method (including not reducing factors at all) provides the better set of factors for the purpose of effort estimation?

H2. The integrated approach provides a set of factors that ensure higher performance of effort estimation than factors provided by expert- and data-based selection approaches when applied individually.

Some effort estimation methods such as stepwise regression [5] or OSR [4] already include embedded mechanisms to select relevant productivity factors. In our study, we wanted to evaluate in addition how preliminary factor selection done by an independent method influences the performance of such estimation methods. This leads us to a general research question:

Q3. Does application of an independent factor selection method increase the prediction performance of an estimation method that already has an embedded factor selection mechanism?

Answering such a generic question would require evaluating all possible estimation methods. This, however, is beyond the scope of this study. We limit our investigation to the OSR estimation method [4] and define a corresponding research hypothesis:

H3. Application of an independent factor selection method does not increase the prediction performance of the OSR method.

Finally, in order to validate replicate the results of the most recent research regarding the application of data-based factor selection to analogy-based effort estimation (e.g., [6, 10]) we define the following question:

Q4. Does application of a data-based factor selection method increase the prediction performance of an analogy estimation method?

H4. Application of a data-based factor selection method increases the prediction performance of a k-NN estimation method.

4.2 Study Context and Empirical Data

The empirical evaluation was performed in the context of Toshiba Information Systems (Japan) Corporation (TJSYS). The project measurement data repository contained a total of 76 projects from the information systems domain. Fig. 3 illustrates the variance of development productivity measured as function points (unadjusted, IFPUG) per man-month.

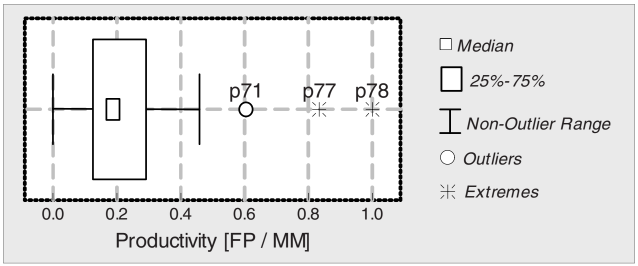


Fig. 3. Development productivity variance (data presented in a normalized form)

Expert assessments regarding the most relevant factors were obtained from three experts (see Table 1). During the group meeting (brainstorming session) an initial set of factors was identified. It was then grouped into project-, process, personnel-, and product-related factors as well as context factors. The first four groups refer to the characteristics of the respective entities (software project, development process, products, and stakeholders). The latter group covers factors commonly used to limit the context of software effort estimation or productivity modeling. The application domain, for instance, is often regarded as a context factor, i.e., an effort model is built for a specific application domain. Finally, experts were asked to select the 5 most important factors from each category and rank them from the most relevant (rank = 1) to least relevant (rank = 5).

Table 1. Experts participated in the study

	Expert 1	**Expert 2**	**Expert 3**
Position/Role	Project manager	Developer	Quality manager
Experience [#working years]	8	15	3
Experience [#performed projects]	30	15	40

4.3 Study Limitations

Unfortunately, the measurement repository available did not cover all relevant factors selected by the experts. It was also not possible to collect the data ex post facto. This prevented us from doing a full comparative evaluation of the three factor selection methods considered here for the purpose of software effort estimation. In order to at least get an indication of the methods' performance, we decided to compare them (instead of all identified factors) on the factors identified by experts for which measurement data were available. This would represent the situation where those factors cover all factors available in the repository and identified by experts.

4.4 Study Design and Execution

4.4.1 Data Preprocessing

Measurement data available in the study suffered from incompleteness (44.3% missing data). An initial preprocessing was thus required in order to apply the data analysis techniques selected in the study. We wanted to avoid using simple approaches to handling missing data such as list-wise deletion or mean imputation, which significantly reduce data quantity and increase noise. Therefore, we decided to apply the k-Nearest Neighbor (k-NN) imputation method. It is a common *hot deck* method, in which k nearest projects minimizing a certain similarity measure (calculated on non-missing factors) are selected to impute missing data. It also proved to provide relatively good results when applied to sparse data in the context of software effort prediction [14]. Moreover, other more sophisticated (and potentially more effective) imputation methods required removing factor collinearities beforehand. Such a pre-processing step would, however, already be a kind of factor selection and might thus bias the results of the actual factor selection experiment. We adopted the k-NN imputation approach presented in [9]. In order to assure maximal performance of the imputation, before applying it, we removed factors and projects with large missing data ratio so that the total ratio of missing data was reduced to around one third, however, with minimal loss of non-missing data. We applied the following procedure: We first removed factors where 90% of the data were missing and next, projects where more than 55% of the data were still missing. As a result, we reduced the total rate of missing data to 28.8%, while losing a minimal quantity of information (removed 19 out of 82 factors and 3 out of 78 projects). The remaining 28.8% of missing data were imputed using the k-NN imputation technique.

4.4.2 Empirical Evaluation

Let us first define the following abbreviations for the factor sets used in the study:

***FM*:**	factors covered by measurement data.
***FM_R*:**	relevant FM factors selected by the RReliefF method (factors with weight > 0)
***FM_{R10}*:**	the 10% most relevant FM_R factors
***FE*:**	factors selected by experts
***FI*:**	factors selected by the integrated method
***FT*:**	all identified factors ($FM \cup FE$)
***FC*:**	factors selected by experts for which measurement data are available ($FM \cap FE$)
***FC_{E25}*:**	the 25% most relevant FC factors selected by experts
***FC_{R25}*:**	the 25% most relevant FC factors selected by the RRF method
***FC_{I25}*:**	the 25% most relevant FC factors selected by the integrated method

Hypothesis H1. In order to evaluate H1, we compared factor sets selected by the data-based, expert-based, and integrated method (FM_R, FE, and FI). For the 10 most relevant factors shared by all three factor sets, we compared the ranking agreement using Kendall's coefficient of concordance [20].

Hypothesis H2. In order to evaluate H2, we evaluated the estimation performance of two data-based estimation methods: k-Nearest Neighbor (k-NN) [19] and Optimized Set Reduction (OSR) [4]. We applied them in a leave-one-out cross validation on the following factor sets: FM, FC, FC_{E25}, FC_{R25}, and FC_{I25}.

Hypothesis H3. In order to evaluate H3, we compared the estimation performance of OSR (which includes an embedded, data-based factor selection mechanism) when applied on FM and FM_{R10} factor sets.

Hypothesis H4. In order to evaluate H4, we compared the estimation performance of the k-NN method when applied on FM and FM_{R10} factor sets.

To quantify the estimation performance in H2, H3, and H4, we applied the common accuracy and precision measures defined in [7]: magnitude of relative estimation error (MRE), mean and median of MRE (MMRE and MdMRE), as well as prediction at level 25% (Pred.25). We also performed an analysis of variance (ANOVA) [20] of MRE to see if the error for one approach was statistically different from another. We interpret the results as statistically significant if the results could be due to chance less than 2% of the time ($p < 0.02$).

5 Results of the Empirical Study

Hypothesis H1: *Expert-based, data-based and integrated methods select different (probably partially overlapping) sets of factors.*

After excluding the dependent variable (development productivity) and project ID, the measurement repository contained data on 61 factors. Experts identified a total of 34 relevant factors, with only 18 of them being already measured (FC). The RRF method selected 40 factors (FM_R), 14 of which were also selected by experts. The integrated approach selected 59 factors in total, with only 14 being shared with the former two selection methods. Among the FC factors, as many as 8 were ranked by each method within the top 10 factors (Table 2). Among the top 25% FC factors selected by each method, only one factor was in common, namely *customer commitment and participation*. There was no significant agreement (Kendall = 0.65 at p=0.185) between data- and expert-based rankings on the FC factors. The integrated method introduced significant agreement on ranks produced by all three methods (Kendall = 0.72 at p = 004).

Interpretation (H1): Data- and expert-based selection methods provided different (partially overlapping) sets of relevant factors. Subjective evaluation of the shared factors suggests that both methods vary regarding the assigned factor's importance; yet this could not be confirmed by statistically significant results. The integrated method introduced a consensus between individual selections (significant agreement) and as such might be considered as a way to combine the knowledge gathered in experts' heads and in measurement data repositories.

Table 2. Comparison of the ranks on FC factors (top 25% marked in bold)

Productivity factor	FC_E	FC_R	FC_I
Customer commitment and participation	**3**	**3**	**3**
System configuration (e.g., client-server)	5	**2**	5
Application domain (e.g., telecommunication)	**1**	6	**1**
Development type (e.g., enhancement)	7	**1**	**4**
Application type (e.g., embedded)	**2**	7	**2**
Level of reuse	9	**4**	9
Required product quality	6	10	7
Peak team size	8	9	8

Hypothesis H2: *The integrated approach provides a set of factors that ensure higher performance of effort estimation than factors provided by expert- and data-based selection approaches when applied individually.*

A subjective analysis of the estimates in Table 3 suggests that the k-NN provided improved estimates when applied on a reduced FC factors set (FC_{E25}, FC_{R25}, and FC_{I25}), whereas OSR does not consistently benefit from independent factor reduction (by improved estimates). The analysis of the MRE variance, however, showed that the only significant ($p = 0.016$) improvement in estimation performance of the k-NN predictor was caused by the integrated factor selection method. The OSR predictor improved its estimates significantly ($p < 0.02$) only on the FC_{E25} factors set.

Interpretation (H2): The results obtained indicate that a factors set reduced through an integrated selection contributes to improved effort estimates. Yet, this does not seem to depend on any specific way of integration. The k-NN predictor, which uses all input factors, improved on factors reduced by the AvalOn method. The OSR method, however, improved slightly on the factors reduced by experts. This interesting observation might be explained by the fact that OSR, which includes an embedded, data-based factor selection mechanism, combined this with prior expert-based factor selection. Still, the effectiveness of such an approach largely depends on the experts who determine (pre-select) input factors for OSR (expert-based selection is practically always granted higher priority).

Table 3. Comparison of various factor selection methods

Predictor	*Factors Set*	MMRE	MdMRE	Pred.25
k-NN	*FM*	73.7%	43.8%	21.3%
	FC	52.6%	40.0%	26.7%
	FC_{E25}	46.3%	38.5%	33.3%
	FC_{R25}	48.3%	36.9%	29.3%
	$\mathbf{FC_{I25}}$	**47.5%**	**33.3%**	**30.7%**
OSR	*FM*	59.7%	50.8%	17.3%
	FC	65.9%	59.2%	18.7%
	$\mathbf{FC_{E25}}$	**30.7%**	**57.9%**	**24.0%**
	FC_{R25}	66.2%	52.1%	14.7%
	FC_{I25}	65.1%	57.9%	14.7%

Hypothesis H3: *Application of an independent factor selection method does not increase the prediction performance of the OSR method.*

A subjective analysis of OSR's estimation error (Table 3 and Table 4) suggests that it performs generally worse when applied on the factors chosen by an independent selection method. This observation was, however, not supported by the analysis of the

Table 4. Results of data-based factor selection

Predictor	*Factors Set*	MMRE	MdMRE	Pred.25	ANOVA
k-NN	*FM*	73.7%	43.8%	21.3%	p = 0.39
	FM_{R10}	56.8%	40.7%	22.7%	
OSR	*FM*	59.7%	50.8%	17.3%	p = 0.90
	FM_{R10}	68.1%	59.1%	16.0%	

MRE variance. The exception was the FC set reduced by experts (FC_{E25}), on which a slight, statistically significant improvement of the OSR's predictions was observed.

Interpretation (H3): The results obtained indicate that no general conclusion regarding the impact of independent factor selection on the prediction performance of OSR can be drawn. Since no significant deterioration of estimation performance was observed, application of OSR on the reduced set of factors can be considered useful due to the reduced cost of measurement. Yet, improving OSR's estimates might require a selection method that is more effective than the selection mechanism embedded in OSR.

Hypothesis H4: *Application of a data-based factor selection method increases the prediction performance of a k-NN estimation method.*

A subjective impression of improved estimates provided by the k-NN predictor (Table 4) when applied on the reduced factors set (FM_{R10}) was, however, not significant in the sense of different variances of MRE ($p = 0.39$). Yet, estimates provided by the k-NN predictor improved significantly when used on the FC data set reduced by the integrated selection method ($p = 0.016$). The two individual selection methods did not significantly improve performance of the k-NN predictor.

Interpretation (H4): Although a subjective analysis of the results (Table 3 and Table 4) suggests improved estimates provided by the k-NN predictor when applied on reduced factors sets, no unambiguous conclusion can be drawn. The performance of k-NN improved significantly only when applied on factors identified from the FC set by the integrated selection method (the FC_{I25} set). This might indicate that k-NN's performance improvement depends on the applied factor selection method (here, the integrated method was the best one).

5.1 Threats to Validity

We have identified two major threats to validity that may limit the generalizability of the study results. First, the estimation performance results of the factor selection methods investigated, compared on the FC set, may not reflect their true characteristics, i.e., as compared on the complete set of identified factors (threat to hypothesis H2). Yet, a lack of measurement data prevented us from checking on this. Second, the RRF method includes the k-NN strategy to search through the factor space and iteratively modify factor weights. This might bias the results of k-NN-based estimation by contributing to better performance of k-NN (as compared to OSR) on factors selected by RRF (threat to hypotheses H3 and H4).

6 Summary and Further Work

In the paper, we proposed an integrated approach for selecting relevant factors influencing software development productivity. We compared the approach in an empirical study against selected expert- and data-based factor selection approaches.

The investigation performed showed that expert- and data-based selection methods identified different (only partially overlapping) sets of relevant factors. The study indicated that the *AvalOn* method finds a consensus between factors identified by individual selection methods. It combines not only the sets of relevant factors, but the individual relevancy levels of selected factors. We showed that in contrast to data- and expert-based factor selection methods, the integrated approach may significantly improve the estimation performance of estimation methods that do not include an embedded factor selection mechanism. Estimation methods that include such a mechanism may, however, benefit from integrating their capabilities with expert-based factor selection.

The study did not replicate the observation of similar investigations regarding improved estimation performance on the factor sets reduced by a data-based selection method. Neither of the estimation methods employed in the study (k-NN and OSR) improved significantly when applied on factor sets reduced by the RReliefF method. Although k-NN improved in terms of aggregated error measures (e.g., MMRE) the difference in the MRE variance was insignificant. The results obtained for the OSR method may indicate that the change of its prediction performance when applied on a reduced set of factors depends on the selection method used.

Finally, we also observed that the function point adjustment factor (FPAF) was not considered among the most relevant factors, although factor selection was driven by a variance on development productivity calculated from unadjusted function point size. Moreover, some of the factors considered as relevant (e.g., performance requirements) belong to components of the FPAF. This might indicate that less relevant sub-factors of the FPAF and/or the adjustment procedure itself may hide the impact of relevant factors. Considering sub-factors of FPAF individually might therefore be more beneficial.

In conclusion, factor selection shall be considered as an important aspect of software development management. Since individual selection strategies seem to provide inconsistent results, integrated approaches should be investigated to support software practitioners in limiting the cost of management (data collection and analysis) and increasing the benefits (understanding and improvement of development processes).

Further work will focus on a full evaluation of the three selection strategies presented on a complete set of measurement data (including data on all factors identified by experts). Finally, methods to identify and explicitly consider factor dependencies require investigation. Such information may not only improve the performance of effort estimation methods, but also the understanding of interactions among organizational processes influencing development productivity.

Acknowledgments. We would like to thank Toshiba Information Systems Corporation (Japan) and all involved experts who greatly contributed to the study. We would also like to thank the Japanese Information-technology Promotion Agency for supporting the study. Finally, we would like to thank Sonnhild Namingha and Marcus Ciolkowski for reviewing the paper.

References

1. Auer, M., Trendowicz, A., Graser, B., Haunschmid, E., Biffl, S.: Optimal project feature weights in analogy-based cost estimation: improvement and limitations. IEEE Transactions on Software Engineering 32(2), 83–92 (2006)
2. Boehm, B.W., Abts, C., Brown, A.W., Chulani, S., Clark, B.K., Horowitz, E., Madachy, R., Refer, D., Steece, B.: Software Cost Estimation with COCOMO II. Prentice-Hall, Englewood Cliffs (2000)

3. Boehm, B.W.: Software Engineering Economics. Prentice-Hall, Englewood Cliffs (1981)
4. Briand, L., Basili, V., Thomas, W.: A Pattern Recognition Approach for Software Engineering Data Analysis. IEEE Transactions on Software Engineering 18(11), 931–942 (1992)
5. Chatterjee, S., Hadi, A.S., Price, B.: Regression Analysis by Example, 3rd edn. Wiley, Chichester (1999)
6. Chen, Z., Menzies, T., Port, D., Boehm, B.: Finding the right data for software cost modeling. IEEE Software 22(6), 38–46 (2005)
7. Conte, S., Dunsmore, H., Shen, V.Y.: Software Engineering Metrics and Models. Benjamin Cummings, CA (1986)
8. Guyon, I., Elisseeff, A.: An Introduction to Variable and Feature Selection. Journal of Machine Learning Research 3, 1157–1182 (2003)
9. Jönsson, P., Wohlin, C.: An Evaluation of k-Nearest Neighbour Imputation Using Likert Data. In: 10th Int'l Symposium on Software Metrics, pp. 108–118 (2005)
10. Kirsopp, C., Shepperd, M., Hart, J.: Search Heuristics, Case-based Reasoning and Software Project Effort Prediction. In: Genetic and Evolutionary Computation Conference, pp. 1367–1374 (2002)
11. Li, J., Ruhe, G.: A comparative study of attribute weighting heuristics for effort estimation by analogy. In: International Symposium on Empirical Software Engineering, pp. 66–74 (2006)
12. Liang, T., Noore, A.: Multistage software estimation. In: 35th Southeastern Symposium on System Theory, pp. 232–236 (2003)
13. Maxwell, K.D., Van Wassenhove, L., Dutta, S.: Software development productivity of European space, military, and industrial applications. IEEE Transactions on Software Engineering 22(10), 706–718 (1996)
14. Myrtveit, I., Stensrud, E., Olsson, U.H.: Analyzing data sets with missing data: An empirical evaluation of imputation methods and likelihood-based methods. IEEE Transactions Software Engineering 27, 999–1013 (2001)
15. Ochs, M., Pfahl, D., Chrobok-Diening, G., Nothhelfer-Kolb, B.: A Method for efficient measurement-based COTS Assessment & Selection – method description and evaluation results. In: 7th International Software Metrics Symposium (2001)
16. Robnik-Sikonja, M., Kononenko, I.: Theoretical and Empirical Analysis of ReliefF and RReliefF. The Machine Learning Journal 53, 23–69 (2003)
17. Saaty, T.L.: The Analytic Hierarchy Process. McGraw-Hill, New York (1990)
18. Schillinger, D.: Entwicklung eines simulationsfähigen COTS Assessment und Selection Tools auf Basis eines für Software adequaten hierarchischen MCDM Meta Modells. Ms Thesis, Dept. of Computer Science, TU Kaiserslautern (Supervisors): Prof. Dr. D. Rombach, Michael Ochs, Kaiserslautern, Germany (2006)
19. Shepperd, M., Schofield, C.: Estimating Software Project Effort Using Analogies. IEEE Transactions on Software Engineering 23(12), 736–743 (1997)
20. Sheskin, D.J., Sheskin, D.: Handbook of Parametric and Nonparametric Statistical Procedures, 2nd edn. Chapman & Hall/CRC (2000)
21. Spector, P.: Summated Rating Scale Construction. Sage Publications, Thousand Oaks (1992)
22. Subramanian, G.H., Breslawski, S.: Dimensionality reduction in software development effort estimation. Journal of Systems and Software 21(2), 187–196 (1993)
23. The Standish Group. CHAOS Chronicles. West Yarmouth, MA (2003)

24. Trendowicz, A., Heidrich, J., Münch, J., Ishigai, Y., Yokoyama, K., Kikuchi, N.: Development of a Hybrid Cost Estimation Model in an Iterative Manner. In: 28th International Conference on Software Engineering, Shanghai, China, pp. 331–340 (2006)
25. Vincke, P.: Multicriteria Decision-aid. John Wiley & Sons, Chichester (1992)
26. Witten, I.H., Frank, E.: Data Mining: Practical machine learning tools and techniques, 2nd edn. Morgan Kaufmann, San Francisco (2005)
27. Trendowicz, A.: Factors Influencing Software Development Productivity - State of the Art and Industrial Experiences. Report no. 008.07/E. Fraunhofer IESE, Kaiserslautern, Germany (2007)

A Framework for QoS Contract Negotiation in Component-Based Applications

Mesfin Mulugeta and Alexander Schill

Institute for System Architecture,
Dresden University of Technology, Germany
{mulugeta,schill}@rn.inf.tu-dresden.de

Abstract. The support of QoS properties in component-based software requires the run-time selection of appropriate concrete QoS contracts at the ports of the interacting components. Such a selection process is called QoS contract negotiation. This paper discusses the architecture of a QoS contract negotiation framework and how it is implemented in our prototype. The framework can be integrated in a component container and act as a run-time support environment when QoS contracts are negotiated under different application scenarios. Our approach is based on: (i) the notion that the required and provided QoS properties as well as resource demands are specified at the component level; and (ii) that QoS contract negotiation is modeled as a constraint solving problem.

1 Introduction

Component-Based Software Engineering (CBSE) allows the composition of complex systems and applications out of well defined parts (components). In today's mature component models (e.g. EJB and Microsoft's .NET), components are specified with syntactic contracts that provide information about which methods are available and limited non-functional attributes like transaction properties. This underspecifies the components and limits their suitability and reuse to a specific area of application and environment. In [2], component contracts have been identified in four different levels: syntactic, behavioral, synchronization, and QoS. The explicit consideration of component QoS contracts aims at simplifying the development of component-based software with non-functional requirements like QoS, but it is also a challenging task.

For applications in which the consideration of non-functional properties (NFPs) is essential (e.g. Video-on-Demand), a component-based solution demands the appropriate composition of the QoS contracts specified at the different ports of the collaborating components. The ports must be properly connected so that the QoS level required by one must be matched by the QoS level provided by the other. This matching requires the selection of appropriate QoS contracts at each port. Generally, QoS contracts of components depend on run-time resources (e.g. network bandwidth, CPU time) or quality attributes to be established dynamically. QoS contract negotiation involves the run-time selection of appropriate concrete QoS contracts specified at the ports of the interacting components.

In [9], we presented how QoS contract negotiation can be formulated as a constraint solving problem and proposed two-phased heuristic algorithms in a single-client - single-server and multiple-clients scenarios. This paper focuses on a conceptual negotiation framework that can be integrated in a component container to act as a run-time support environment when QoS contracts are negotiated under different application scenarios. We also discuss details of our prototype implementation of the framework. The advantages of having the framework are the following. Firstly, it can be directly used for various types of applications as long as similar QoS contract specification schemes are used. Secondly, the framework can be extended to handle different scenarios either by incorporating new negotiation algorithms or by including more features with respect to the basic components of the framework.

The rest of the paper is organized as follows. In section 2 we examine related work. Section 3 details our QoS contract negotiation framework. Section 4 is devoted to the discussion of how we have implemented the proposed framework by demonstrating the ideas based on an example application scenario. The paper closes with a summary and outlook to future work.

2 Related Work

The work in [4] offers basic QoS negotiation mechanisms in only a single container. It hasn't pursued the case of distributed applications where components are deployed in multiple containers. In [11] QoS contract negotiation is applied when two components are explicitly connected via their ports. In the negotiation, the client component contacts the server component by providing its requirement; the server responds with a list of concrete contract offers; and the client finally decides and chooses one of the offers. This approach covers only the protocol aspect of the negotiation process. It hasn't pursued the decision making aspects of the negotiation.

In [8] a model is described where a component provides a set of interrelated services to other components. These components are QoS-aware and are capable of engaging in QoS negotiations with other components of a distributed application. The paper attempts to create a framework for software components that are capable of negotiating QoS goals in a dynamic fashion using analytic performance models. The QoS negotiation between two components occurs by taking performance as a QoS requirement and concurrency level as a means of negotiation element. Our treatment of QoS negotiation is more generic and general, which may be applied for a larger set of problems. Moreover, the container handles the negotiation between components in our case, which enhances the reusability of the components. QuA [13] aims at defining an abstract component architecture, including the semantics for general QoS specifications. QuA's QoS-driven Service Planning has similarities to our concept of QoS contract negotiation. Complexity issues, however, haven't been accounted for in the service planning.

The Quality Objects (QuO) framework offers one of the most advanced concepts and tools to integrate QoS into distributed applications based on CORBA

[7]. In QuO, a QoS developer specifies a QoS contract between the client and object. This contract specifies the QoS that the client desires from the object, the QoS that the object expects to provide, operating regions indicating possible measured QoS, and actions to take when the level of QoS changes. Having to provide the adaptive behavior explicitly in the QoS contract is a burden on the QoS developer. In our work, we have taken the approach that instead of specifying the pre-determined adaptive behavior in the QoS contract, we leave the reasoning on adaptation (or negotiation) to the component containers, which they would perform based on the specification of the component's QoS profiles. One advantage of this approach is that it makes the application development process easier.

3 Framework Architecture and Interaction

3.1 Architecture

Our framework can be seen as a reusable design that consists of the representation of important active components, data entities, and the interaction of different instances of these to enable QoS contract negotiation. Fig. 1 shows the conceptual architecture of our framework represented as a UML class diagram.

`Negotiator` coordinates and performs the contract negotiation on behalf of the interacting components. In order for `Negotiator` to decide on the solution (i.e. selection of appropriate concrete QoS contracts at the ports of components), it has to make reference to: (i) the QoS specification of all the cooperating components, which is assumed to be available declaratively in the form of one or more QoS profiles, (ii) user's QoS requirement and preferences, (iii) available resource conditions, (iv) network and container properties, and (v) policy constraints. Finally, `Negotiator` establishes contracts, which will have to be monitored and enforced by the container. Next, we describe the different building blocks of our framework.

QoS Profile. Component's QoS Contracts are specified with one or more QoS profiles. A component's QoS contract is distinguished into *offered QoS contract* and *required QoS contract* [10]. We use CQML$^+$ [12][5], an extension of CQML

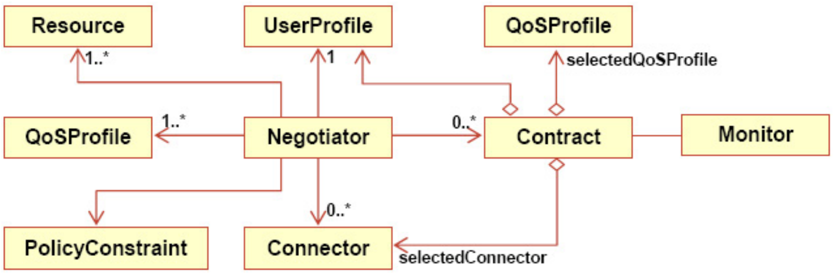


Fig. 1. Architecture of a QoS Contract Negotiation Framework

[1], to specify the offered- and required-QoS contract of a component. CQML$^+$ uses the *QoS-Profile* construct to specify the NFPs (provided and required QoS contracts) of a component's implementation in terms of what qualities a component requires (through a *uses* clause) from other components and what qualities it provides (through a *provides* clause) to other interacting components, and the resource demand by the component from the underlying platform (through a *resource* clause). The *uses* and *provides* clauses are described by a *QoS statement* that constrain a certain quality characteristic in its value range. A simplified example shown below depicts these elements for a `VideoPlayer` component that may be used in video streaming scenarios.

```
QoSProfile aProfile for VideoPlayer {
    provides frameRate=15, resolution=352x288;
    uses frameRate=15, resolution=352x288;
    resources cpu=8.9%; networkBandwidth=2.1kbps; memory=30KB;
}
```

It is assumed that the component developer specifies the QoS-Profiles after conducting experiments and measuring the provided quality, required quality, and the resource demands at the component level.

Connector. `Connector` is an abstraction of the network and the containers that exist between interacting components deployed on multiple nodes. A communication channel may have a number of QoS properties. For example, it introduces a delay. The connector properties are used when matching conformance between provided- and required-QoS contracts of components interacting across containers.

It is assumed that the values of the connector properties are available to `Negotiator` before negotiation starts. Two possible approaches for estimating the values are: (i) Off-line measurement - the required properties are measured off-line by applying different input conditions (e.g. throughput) and load conditions in the network and end-systems; and (ii) On-line measurement - the properties are measured during the application launch and/or at run-time.

User Profile. `UserProfile` is used to specify the user's QoS requirements and preferences. The user's requirement may be specified for one or more QoS-dimensions. Additional parameters such as *user class* need to be defined when considering, for example, a multiple-clients scenario. `UserProfile` is assumed to be constructed by the run-time system after obtaining the user's request for a given service. The user might be given the chance to select values of attributes from one of many templates supplied for the application or specify the attributes himself.

Resource. `Resource` is used to store information about the available resources at the nodes and the end-to-end bandwidth between nodes in which components are deployed. Monitoring functions are used to supply data about a node's load conditions on CPU, memory, etc. It is assumed that the available resources are monitored at run-time. Changes in available resources might initiate re-negotiation.

Negotiator. A user's request to get a service is first intercepted by `Negotiator` on the client node. The `Negotiator` at the client and server side exchange information about the required service and the user's profile before the negotiation begins. `Negotiator` is responsible for selecting appropriate QoS-Profiles of the interacting components that should satisfy a number of constraints (e.g. user's, resource, etc.). It is also responsible for finding a good solution from a set of possible solutions. `Negotiator` creates `Contract` after successfully performing the negotiation. For an unsuccessful negotiation, the selection process is repeated after systematically relaxing the user's QoS requirement.

In order to accomplish the stated responsibilities, `Negotiator` relies on our modeling of the QoS contract negotiation as a Constraint Satisfaction Optimization Problem (CSOP) [14]. A CSOP consists of variables whose values are taken from finite, discrete domains, a set of constraints on their values, and an objective function. The task in a CSOP is to assign a value to each variable so that all the constraints are satisfied and a solution that has an optimal value with regard to the objective function is found. The objective function maps every solution to a numerical value.

In the above modeling, we take the variables to be the QoS-Profiles to be used for the collaborating components. The domain of each variable is the set of all QoS-Profiles specified for a component. The constraints identified are classified as *conformance*, *user's*, and *resource*. As an objective function, we use an application utility function [6], which is represented by mapping quality points to real numbers in the range [0, 1] where 0 represents the lowest and 1 the highest quality.

Contract. The creation of contracts proceeds after the selection of appropriate concrete QoS-profiles of the interacting components. Contracts may exist between components deployed in the same or different containers. In the case of a front-end component, a contract exists between this component and the user. A simplified abstraction of `Contract` is given below.

```
public class Contract {
     QoSProfile selectedQoSProfileClient;
     QoSProfile selectedQoSProfileServer;
     Connector selectedConnector
     UserProfile userProfile;
     double contractValidityPeriod;
     //...
};
```

If a contract is established between two components deployed in the same container, the clauses of the contract contains the QoS offers and needs as well as the resource demands of the components. That means, the selected QoS-profiles of the client and server components would be clauses in the contract (in this case, `selectedConnector` and `userProfile` are `null`). If a contract is established between components across containers, a selected connector is also part of the contract. For a contract between a user and the front-end component, a user's profile would become part of the contract (`selectedQoSProfileClient` and `selectedConnector` are `null` in this case). Note that resources required from the underlying platform are included in the contract through the QoS-profiles. Additional parameters such as contract dependencies, etc. also need to be defined in the contract in order to facilitate contract monitoring and enforcement.

Monitor. After contracts are established, they can be violated for a number of reasons like a shortage of available resources. `Monitor` constantly monitors contracts to assure that no contract violations would occur and in case one occurs, some corrective measures should be taken through contract re-negotiations.

Policy Constraints. As described previously, `Negotiator` uses a CSOP framework to find good solution. The CSOP framework in turn relies on the specification of constraints and a utility function in order to find appropriate solutions. There are, however, certain behaviors that cannot be captured in utility functions. Such behaviors are modeled as policy constraint, which can be defined as an explicit representation of the desired behavior of the system during contract negotiation and re-negotiation. `Negotiator` can achieve, for instance, different optimization goals based on varying specifications in the policy constraints. For e.g., the service provider might want to allocate different percentages of resources to different user classes (e.g. premium and normal users).

3.2 Interaction

The interaction diagram in Fig. 2 depicts an overall view of the negotiation process. It is assumed that the application's components are deployed in client and server containers. The diagram shows a successful negotiation scenario performed using a centralized approach. The following is demonstrated in Fig. 2.

- A user requests the application for a service by providing the service's name (e.g. playing a given movie or performing payment for usage of a particular operation) together with his/her QoS and preference needs (step 1).
- Intercepting the user's request, `Negotiator` at the client's container constructs `UserProfile` and sends a message to the server container, which will identify the components that participate to provide the required service (step 2).
- In steps 3 to 5, the container that is responsible for the negotiation collects QoS contracts specified for the collaborating components, resource conditions at each node and the network, and policy constraints that may be imposed by service providers.
- The responsible container performs the negotiation (step 6) in two phases [9]. In the first phase, negotiation is made on coarse-grained properties (step 7). When this is successful, negotiation on fine-grained properties continues (step 8).
- The responsible container creates all contracts. A contract is established between any two interacting components and between a user and the front-end component (step 9).
- The client container retrieves relevant contracts from the server container (step 10). These are contracts between components deployed in the client container or between components connected across containers.

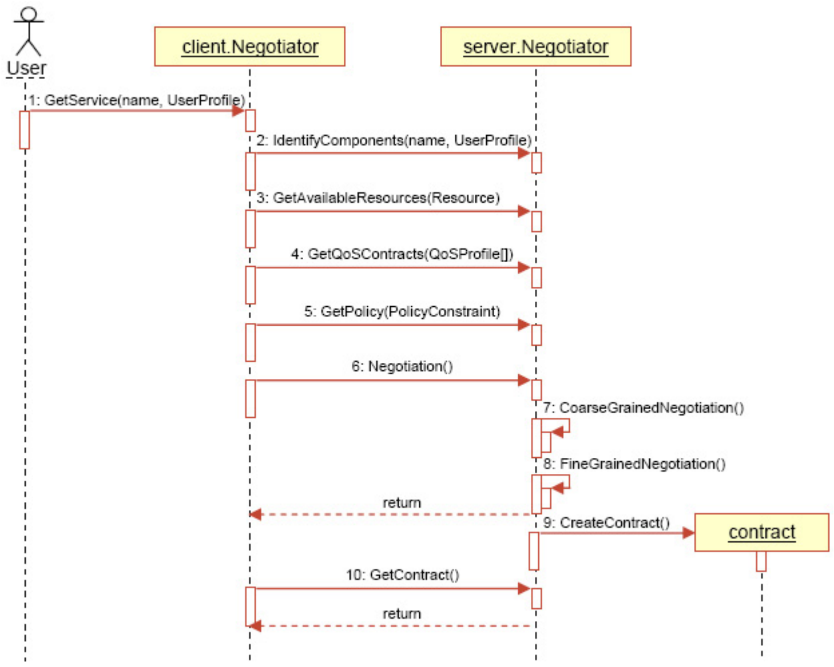

User
client.Negotiator
server.Negotiator
1: GetService(name, UserProfile)
2: IdentifyComponents(name, UserProfile)
3: GetAvailableResources(Resource)
4: GetQoSContracts(QoSProfile[])
5: GetPolicy(PolicyConstraint)
6: Negotiation()
7: CoarseGrainedNegotiation()
8: FineGrainedNegotiation()
return
9: CreateContract()
contract
10: GetContract()
return

Fig. 2. Interaction between client and server containers (centralized approach)

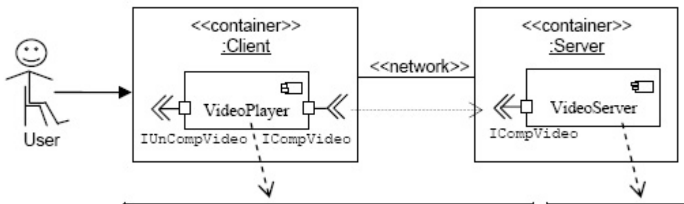
4 Implementation and Example

4.1 Example

As a proof-of-concept, we have developed a prototype of the framework proposed in Fig. 1. Our prototype implements all the elements of the framework with the exception of `PolicyConstraints` and `Monitor`. The framework has been used to negotiate QoS contracts at run-time for a video streaming application scenario. In the future we plan to extend our implementation to include contract monitoring and the consideration of policy constraints. The prototype has been implemented in Java as also demonstrated by the code snippets shown below.

The video streaming application scenario that we used involves a `VideoServer` component deployed in a server container and a `VideoPlayer` component deployed in a client container (Fig. 3). We use the COMQUAD component model [5] that supports streams as special interface types and allows to specify non-functional properties for them. `VideoPlayer` implements two interfaces: a *uses* interface `ICompVideo` and a *provides* interface `IUnCompVideo` while `VideoServer` implements a provides interface `ICompVideo`. `VideoPlayer`'s `ICompVideo` is connected to `VideoServer`'s `ICompVideo` to receive video streams for a playback at the client's node.

We conducted an experiment to specify the QoS-Profiles of `VideoPlayer` and `VideoServer`. The `VideoPlayer` component was implemented using Sun's JMF



	VideoPlayer		
	uses ICompVideo (resolution, frame rate in s^{-1})	provides IUnCompVideo (resolution, frame rate in s^{-1})	Resource (CPU in %, bandwidth in kbps, memory KB)
1	352x288, 30	352x288, 30	13.2, 2165, 32
2	352x288, 15	352x288, 15	8.9, 2146, 30
3	352x288, 5	352x288, 5	5.9, 1852, 30
4	176x144, 30	176x144, 30	1.0, 321, 26
5	176x144, 15	176x144, 15	0.9, 252, 19
6	176x144, 5	176x144, 5	0.4, 135, 24
7	128x96, 30	128x96, 30	0.5, 152, 26
8	128x96, 15	128x96, 15	0.4, 120, 25
9	128x96, 5	128x96, 5	0.2, 70, 24

VideoServer	
provides ICompVideo (resolution, frame rate in s^{-1})	Resource (bandwidth in kbps)
352x288, 30	2165
352x288, 15	2146
352x288, 5	1852
176x144, 30	321
176x144, 15	252
176x144, 5	135
128x96, 30	152
128x96, 15	120
128x96, 5	70

Fig. 3. A video streaming scenario and QoS-profiles of VideoPlayer and VideoServer implementations

framework and the `VideoServer` component abstracts the video media file that has been pre-encoded into many files with differing frame rates, resolutions, protocols, and coding algorithm. Fig. 3 depicts some of the measured QoS-Profiles of `VideoPlayer` and `VideoServer`, with UDP protocol and mp42 coding. Note that these QoS-Profiles depend on the content of the video. During the measurements, average bandwidth and CPU percentage time have been considered. The bandwidth requirement of `VideoServer` is taken to be the same as that of `VideoPlayer`. The measured CPU requirements of `VideoServer` are too small (in the range of 0.1%) and hence have been left out from Fig. 3.

4.2 Implementation

Next, we will see how the framework elements are instantiated and how they interact during the QoS contract negotiation, which is initiated when a user sends his request to get a service. Our subsequent discussion roughly follows the sequence diagram in Fig. 2.

For the example in Fig. 3, a user requests to watch a video clip that is streamed from the video provider to the user. The involved components are `VideoPlayer` and `VideoServer`. A user also sends his QoS requirements from which `UserProfile` is constructed. `UserProfile` is initialized with the user's QoS requirement, i.e. $frameRate \geq 12s^{-1}$ and $resolution = 176 \times 144$.

```
  public class UserProfile {
       List<QoSStatement> uses;
       //...
  };
5 userProfile = new UserProfile({frameRate=12, resolution=176x144});
```

The other element from the framework that needs to be instantiated before the negotiation is started is `Resource`. For the example in Fig. 3, three instances of `Resource` are used as illustrated below. Let's assume that the available resources at the client and server nodes as well as the end-to-end network bandwidth are as indicated below.

```
   public class Resource {
        double cpu; // in percentage
        double memory; // in KB
        double networkBandwidth; // in kbps
10      //...
   };
   clientResources = new Resource(80, 150, null);
   serverResources = new Resource(50, 200, null);
   endToEndBandwidth = new Resource(null, null, 1000);
```

For the video streaming example, the QoS contracts are specified with multiple QoS profiles as shown in Fig. 3. A QoS-Profile for `VideoPlayer` is, for example, represented as:

```
15      QoSProfile aProfile for VideoPlayer {
             provides frameRate=15, resolution=352x288;
             uses frameRate=15, resolution=352x288;
             resources cpu=8.9%; networkBandwidth=2.1kbps; memory=30KB;
        }
```

We used a data structure called `ComponentMeta` to store the QoS-Profiles of each component (i.e. a component's meta data) as shown below. Additional variables are also needed to be defined. `tempSelectedProfile` holds temporarily selected profiles during the negotiation process while `selectedProfile` stores the selected profile after the negotiation is concluded. `currentProfilePos` indicates the position of the temporarily selected profile in the array of QoS profiles, which are stored from low to high quality. Although not shown in the code snippet below, `ComponentMeta` defines, among others, the `Set` and `Get` functions for the variables it defines.

```
20 public class ComponentMeta {
        List<QoSProfile> profiles = null;
        QoSProfile selectedProfile = null;
        QoSProfile tempSelectedProfile = null;
        int currentProfilePos=0;
25      //...
   };
   ComponentMeta c[] = new ComponentMeta[N];
   enum CG {On_Client, On_Server, Across_Containers} // component's group
```

`ComponentMeta` is instantiated for each cooperating component. It is assumed that `profiles` in `ComponentMeta` are populated with values after parsing the QoS specification that is declaratively available as an XML file to the run-time system. CG (Line 28) is used to identify whether a component is deployed on the client, server, or connected across containers.

An instance of `Connector` is required during the negotiation when components are deployed in distributed nodes. The following code snippet assumes that only a delay property is specified.

```
   public class Connector {
30     List<QoSStatement> properties = new ArrayList<QoSStatement>();
       //...
   };
   connector = new Connector({delay=0.001});
```

`Negotiator` uses instances of `c[i]`, `userProfile`, `connector`, `client Resource`, `serverResource`, and `endToEndBandwidth` described above when performing the negotiation. Additional variables are also required for the particular algorithms used in the negotiation.

```
   public class Negotiator {
35     ComponentMeta c[] = null;
       Resource clientResources = null;
       Resource serverResources = null;
       Resource endToEndBandwidth = null;
       UserProfile userProfile = null;
40     Connector connector = null;
       UserProfile boundProfile = null;//initialized to user's QoS
       requirement
       Contract contracts[] = null;
       //...
45     void GetUserProfile() {// gets data for userProfile }
       void GetAvailableResources() {// gets data for the various
       resources}
       void GetQoSContracts() {// gets data for c[i].profiles}
       boolean FineGrainedNegotiation() {//performs fine grained
50     negotiation}
   };
```

`FineGrainedNegotiation()` uses the standard branch and bound (B&B) [14] technique to find a solution for a problem modeled with a CSOP. To apply B&B to our problem, we need to define policies concerning selection of the next variable and selection of the next value. We must also specify the objective and heuristic functions. In our implementation, the variables (QoS-profiles to be used by each component) are ordered for assignment by topologically sorting the network of cooperating components. The assignment starts from the minimal element (i.e. the front-end component, for e.g. `VideoPlayer` in Fig. 3) and from there continues to the connected components, and so on. The possible values of each variable, i.e. the QoS-profiles specified for each component, must be ordered from lower to higher quality.

The heuristic function, `hValue`, maps every partial labeling (assignment) to a numerical value and this value is used to decide whether extending a partial labeling to include a new label would result in a "better" solution. At any point during the assignment of values to variables, the QoS property of the partially completed solution can be taken as the provided QoS contract of the front-end component. Hence, `hValue` (Line 53) can be calculated based on the utility function by taking the QoS points in the provided-QoS contract of the front-end component. Because of the ordering strategy of variables we followed, `hValue` needs to be computed only at the beginning of each iteration, that is, when the front-end component is assigned a new value (Line 52). If the new assignment to the front-end component violates the user's constraint, the choice is retracted and the sub-tree under the particular assignment will be pruned. The process will then re-start with a new assignment.

```
   boolean FineGrainedNegotiation()
50 {
     if (ConformanceCheck() == false) return false;
     for(int i=c[0].GetCurrentProfilePos();i<c[0].profiles.size(); i++) {
       if(hValue(c[0].profiles.get(i).provides)>hValue(boundProfile.uses)){
         c[0].SetTempSelectedQoSProfile(components[0].profiles.get(i));
55       c[0].SetCurrentProfilePos(i+1);
         if(FindAppropriateProfiles()) {
           for(int k=0; k<components.length; k++) {
             c[k].SetSelectedQoSProfile(c[k].GetTempSelectedQoSProfile());
           // change bound with the new value
60         boundProfile = new UserProfile();
           boundProfile.uses = c[0].GetSelectedQoSProfile().provides;
         } else break; // break is a termination condition
       }
     }
65 }
```

`ConformanceCheck()` (Line 51) performs conformance consistency check to every connected pair of components: (C_i, C_j) where C_i is the parent of C_j. It removes QoS-Profiles from the domain of C_i for which no conformant profiles have been specified in C_j. It returns false if there cannot be conformance between at least two connected components.

```
   int FindAppropriateProfiles()
   {
        FindConformantProfiles(CG.On_Client);
        if(CheckResourceConstraints(CG.On_Client)) {
70          FindConformantProfiles(CG.Across_Contaners\CG.On_Client);
            if(CheckResourceConstraints(CG.Across_Contaners)) {
                FindConformantProfiles(CG.On_Server\CG.Across_Contaners);
                if(CheckResourceConstraints(CG.On_Server))
                    return 1; // successful
75              else return 0;
            } else return 0;
        } else return 0;
   }
```

`FindConformantProfiles()` (Line 79) finds QoS-profiles, which are conformant to one another for all the components specified in the input argument. At each iteration this function improves the solution by one step based on the specified QoS-profiles. `IsMatching()` (Line 92) checks the conformance between two interacting components. Conformance [3] exists between two QoS-profiles of interacting components when the server's provided-QoS contract conforms to the client's required-QoS contract. If the interacting components are on different containers (as identified by `AreOnDifferentNodes()` (Line 90)), the connector properties are required during conformance check. A component may belong to two groups in CG (Line 28). For example, a component deployed on the client container and that also communicates across containers belongs to On_Client and Across_Containers. The notation \ in (Lines 70,72) is read as "less". `CheckResourceConstraint()` (Lines 69,71,73) returns true when there are enough resources for the current selection.

```
void FindConformantProfiles(CG componentGroup)
{
  int lowerIndex = GetLowerIndex(componentGroup);
  int higherIndex = GetHigherIndex(componentGroup);
  if(lowerIndex==0)
    int startingIndex = lowerIndex+1; // as the profile of the
          // front-end component has been already selected
  else
    int startingIndex = lowerIndex;
  for(int j=startingIndex; j<=higherIndex;j++) {
    Connector tempConn = null;
    if(AreOnDifferentNodes(c[j-1],c[j])) tempConn = connector;
    for(i=c[j].GetCurrentProfilePos(); i<c[j].profiles.size(); i++) {
      if(IsMatching(c[j-1].GetTempSelectedQoSProfile().uses,
                    c[j].profiles.get(i).provides, tempConn)) {
        c[j].SetTempSelectedQoSProfile(c[j].profiles.get(i));
        c[j].SetCurrentProfilePos(i); break;
      }
    }
  }
}
```

Let's next see the outcome of a negotiation for the example depicted in Fig. 3. Suppose the various input conditions are:

– user's QoS requirement: $frameRate > 12fps, resolution = 176 \times 144$; resolution is preferred over frame rate,
– resource availability: at the client's node, CPU=80%, memory=150KB; at the server's node, CPU=50%, memory=200KB; and the end-to-end bandwidth is 1Mbps,
– QoS contracts of the components are as given in Fig. 3.

As the first solution, `FineGrainedNegotiation()` selects the 5^{th} QoS profiles of `VideoPlayer` and `VideoServer`, i.e. the ones with the offered QoS contract of $176 \times 144, 15fps$. Further iterations improve the solution and ultimately the 6^{th} QoS-profiles of `VideoPlayer` and `VideoServer` are selected. The next step is to establish contracts between `VideoPlayer` and `VideoServer` and between `VideoPlayer` and User. These contracts will then be monitored and enforced by the run-time system.

4.3 Experiences

In the various application scenarios we studied, we had to conduct an experiment to specify the QoS contracts of components. We used these data to check the validity of our approach and its prototype implementation. In our prototype we simulated different behaviors concerning: (i) user's QoS requirements and preferences, (ii) resource availability conditions concerning the client, server, and network bandwidth, and (iii) the specified QoS-Profiles of the collaborating components. Under various conditions, the outcome of the negotiation gives a solution that has the highest utility as far as the most preferred QoS dimension is concerned. The run-time complexity of the negotiation algorithm is $O(nd^2)$ where n is the total number of cooperating components and d is the number of QoS-Profiles specified for each component. Such a complexity is achieved by assuming that the cooperating components form a tree so as to achieve a non-backtracking solution (Lines 68-77).

It is to be noted that our entire approach extensively depends on the QoS-Profiles of the collaborating components. The component developer specifies the QoS-Profiles after conducting experiments and measuring the provided quality, required quality and the resource demand at the component level. Given the same application, constituting components, and same environment, the outcome of the QoS contract negotiation can depend on the specified QoS-Profiles. One of the drawbacks of this is that the solution obtained might not be the optimal one. In order to overcome such a discrepancy, there must be some standard way of specifying QoS contracts, which might be done by either using measurements or analytical means.

5 Conclusions and Outlook

We presented a QoS contract negotiation framework that can be integrated in a component container using the interceptor pattern, which enables adding cross-cutting concerns like contract negotiation. The framework acts as a run-time support environment when QoS contracts are negotiated in various applications. As a proof-of-concept we have developed a prototype of the proposed framework. This paper discussed the implementation details of the prototype. We also illustrated how the framework can be applied to perform negotiation in a componentized video streaming example application. In the future we plan to extend our implementation to include contract monitoring and the consideration of policy constraints.

References

1. Aagedal, J.Ø.: Quality of Service Support in Development of Distributed Systems. PhD thesis, University of Oslo (2001)
2. Beugnard, A., Jézéquel, J.-M., Plouzeau, N., Watkins, D.: Making components contract aware. IEEE Computer 32(7), 38–45 (1999)
3. Frolund, S., Koistinen, J.: Quality-of-Service specification in distributed object systems. IOP/BCS Distributed Systems Engineering Journal (December 1998)
4. Göbel, S., Pohl, C., Aigner, R., Pohlack, M., Röttger, S., Zschaler, S.: The COMQUAD component container architecture and contract negotiation. Technical Report TUD-FI04-04, Technische Universität Dresden (April 2004)
5. Göbel, S., Pohl, C., Röttger, S., Zschaler, S.: The COMQUAD Component Model—Enabling Dynamic Selection of Implementations by Weaving Non-functional Aspects. In: 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004), Lancaster, UK, March 22–26 (2004)
6. Lee, C., Lehoczky, J., Rajkumar, R., Siewiorek, D.P.: On quality of service optimization with discrete qos options. In: IEEE Real Time Technology and Applications Symposium, p. 276 (1999)

7. Loyall, J., Schantz, R., Zinky, J., Bakken, D.: Specifying and measuring quality of service in distributed object systems. In: Proc. 1st Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 1998), Kyoto, Japan (April 1998)
8. Menascé, D.A., Ruan, H., Gomaa, H.: A framework for qos-aware software components. In: The fourth international workshop on Software and performance, Redwood Shores, CA, USA, pp. 186–196 (2004)
9. Mulugeta, M., Schill, A.: An approach for QoS contract negotiation in distributed component-based software. In: Schmidt, H.W., Crnković, I., Heineman, G.T., Stafford, J.A. (eds.) CBSE 2007. LNCS, vol. 4608. Springer, Heidelberg (2007)
10. Object Management Group. UML profile for modeling quality of service and fault tolerance characteristics and mechanisms, v1.0. OMG Document (May 2006), `http://www.omg.org/docs/formal/06-05-02.pdf`
11. Ritter, T., Born, M., Unterschutz, T., Weis, T.: A QoS metamodel and its realization in a CORBA component infrastructure. In: Proceedings of the Hawaii International Conference on System Sciences (2003)
12. Röttger, S., Zschaler, S.: CQML$^+$: Enhancements to CQML. In: Bruel, J.-M. (ed.) Proc. 1st Int'l Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France, pp. 43–56. Cépaduès-Éditions (June 2003)
13. Staehli, R., Eliassen, F., Amundsen, S.: Designing adaptive middleware for reuse. In: ARM 2004: Proceedings of the 3rd workshop on Adaptive and reflective middleware, pp. 189–194. ACM Press, New York (2004)
14. Tsang, E.P.K.: Foundations of Constraint Satisfaction. Academic Press, London (1993)

A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team

Raimund Moser[1], Pekka Abrahamsson[2], Witold Pedrycz[3], Alberto Sillitti[1], and Giancarlo Succi[1]

[1] Free University of Bolzano-Bozen, Italy
{rmoser,asillitti,gsucci}@unibz.it
[2] VTT Electronics, Oulu, Finland
pekka.abrahamsson@vtt.fi
[3] University of Alberta, Canada
pedrycz@ee.ualberta.ca

Abstract. Refactoring is a hot and controversial issue. Supporters claim that it helps increasing the quality of the code, making it easier to understand, modify and maintain. Moreover, there are also claims that refactoring yields higher development productivity – however, there is only limited empirical evidence of such assumption. A case study has been conducted to assess the impact of refactoring in a close-to industrial environment. Results indicate that refactoring not only increases aspects of software quality, but also improves productivity. Our findings are applicable to small teams working in similar, highly volatile domains (ours is application development for mobile devices). However, additional research is needed to ensure that this is indeed true and to generalize it to other contexts.

Keywords: Refactoring, Software process, Methodologies, Software metrics.

1 Introduction

Fowler defines refactoring as “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior” [17]. In Agile Methods refactoring is an integral part of the development process; it is adopted to improve continuously the structure and understandability of source code during development. In the agile community it is widely accepted that refactoring contributes to confine the complexity of source code and has a positive impact on the understandability and maintainability of a software system: frequently refactored code is believed to be easier to understand, correct and adjust to new requirements.

A growing number of studies address the relationship between refactoring and the internal structure of source code and its impact on program understanding, software quality, and the evolution of a software design: an excellent overview is given in [25]. Most of these studies focus on the following issues:

- Impact of refactoring on the structure of the source code [6]
- Identification of code smells to locate possible refactorings [7], [15], [28], [32]

- In reverse engineering, how refactoring can reconstruct the overall design of existing systems [12] and improve the quality of legacy code [26]

Mens *et al.* [25] define and discuss different refactoring activities related to the issues mentioned above: In this research we focus on one particular topic, namely the assessment of the effect of refactoring on some quality characteristics that depend or have an impact on software maintainability both from the point of view of the software product and the software process [14]. Only few empirical studies analyze the impact of refactoring on code quality: Demeyer [13] analyzes whether refactoring has a negative impact on program performance; Bois and Mens [6] develop a framework for analyzing the impact of refactoring on internal quality metrics, but they do not provide any experimental validation in an industrial environment. Stroulia and Kapoor [33] perform a case study in an academic environment, where it is shown that size and coupling metrics of a software system decrease after refactoring. Bois *et al.* [7] propose refactoring guidelines for enhancing cohesion and coupling metrics and obtain promising results by applying them on an open source project. Simon *et al.* [32] follow a similar strategy. Sahraoui *et al.* [29] use quality estimation models for analyzing whether some object-oriented metrics can be used for detecting situations where a particular transformation of source code (refactoring) can be applied to improve the quality of a software system. Again, they do not validate their approach within an industrial case study or experiment. Yu *et al.* [35] use a modeling framework for non-functional requirements and relate refactorings to soft goals. They perform a case study, which shows that refactoring can be measured as the transformation on the state of program in the quality space. Tahvildari and Kontogiannis [34] investigate the use of metrics for detecting potential design flaws and for suggesting potentially useful transformations for correcting them. Finally, Kataoka *et al.* [20] provide a quantitative evaluation of maintainability enhancement by refactoring. For the purpose of validation they analyze a project developed by a single developer, but do not provide any information on the development environment. Thus, it is questionable if their findings are valid in a different context where development teams follow a structured process and use common software engineering practices for knowledge sharing.

In the context of Agile Methods there are several claims that refactoring provides four significant advantages [17]:

- Refactoring helps developers to program faster
- Refactoring improves the design of the software
- Refactoring makes software easier to understand
- Refactoring helps developers to find bugs

The first advantage relates to productivity and is probably the most important for managers who are mainly concerned with time to market. Nevertheless, there is almost no solid, empirical, and quantitative evidence of such claim, apart from a small case study, where it appeared that refactoring decreased the long-term productivity [1]. Recently Schofield *et al.* [30] performed a return on investment analysis on an open source project in order to estimate savings in effort, given a specific (beneficial) code change. They found that, most of the time, refactorings have beneficial impacts on maintenance activities, and thus are motivated from an economic perspective.

The last three advantages of refactoring refer to software quality attributes. We have previously mentioned some studies that analyze the impact of code restructuring induced by refactorings on internal product metrics, which are typically used to measure quality attributes, such as complexity, coupling and cohesion. Such early results are promising, still there is a need for (a) additional empirical validation to better understand and generalize the findings, and (b) a clear linkage to external quality attributes, such as number of defects.

Altogether, the real advantages of refactoring are still to be fully assessed [24]. In particular, it is not yet clear whether refactoring increases developer productivity and the extent to which refactoring improves software quality. As regards quality, it appears to be a convergence of positive remarks, still, without solid quantification. Needless to say, a major impediment for a deeper understanding of these issues is a lack of empirical investigation, based on hard data coming from industry.

The paper is organized as follows. In Section 2, we describe our research methodology and the experimental set-up. In Section 3, we present a case study and discuss the results obtained from it; in Section 4, we discuss the limitations of our approach. Finally, conclusions and implications of the investigation are drawn in Section 5.

2 Research Methodology and Experimental Set-Up

In order to investigate a research problem we have to define (a) the objectives and hypotheses of the study, (b) the variables along with the metrics used to measure them, (c) the instruments used in the experiment and the data collection procedure, and (d) the data analysis method. We will discuss each of these points below. The results of the case study and threats to the validity of the experiment are presented in subsequent sections.

2.1 Research Hypotheses

Software is naturally subjected to continuing change, increasing size and complexity and therefore declining maintainability. In particular, in the one-way traditional development process, internal code measures tend to show a continuous increase in complexity and coupling and a decrease in cohesion as new features are added to a software system. This natural process of code corrosion is even more manifest as time goes by [21]. More complex and intertwined code is more difficult to manage and maintain; therefore, we expect that also development productivity will show a decreasing trend over time. In contrast, in XP-like processes, thanks to its agile practices (in particular constant refactoring, unit testing, frequent releases), the complexity of the code and the effort for adding new functionalities is claimed to remain about constant or to grow very slowly [3]. Unfortunately, due to high costs of industrial software development we are not able to run a formal experiment with an industrial partner where we could analyze two similar projects, one developed using XP practices and one without, and compare directly the evolution of respective quality and productivity metrics.

We have to content ourselves with a simpler approach: We focus only on one XP practice, namely refactoring, and compare changes of productivity before and after explicit refactorings and use such comparison as criteria for assessing the impact of refactoring on it. As regards quality and maintainability, we determine the changes of several design metrics after an *explicit refactoring* has been applied and compare changes with the average daily changes per iteration. If they are significantly different (improved) we then conclude that refactoring has a positive effect on code quality and, as a consequence, on software maintainability. We define in section 2.2 what we intend by *explicit refactorings* in the context of our study.

Framed in terms of research questions, we aim at presenting evidence that will allow us to reject (or accept) the following two null hypotheses:

- H^0_A: after an *explicit refactoring* the average productivity for the consecutive development iteration is the same as for the previous iteration.
- H^0_B: the considered internal quality metrics (complexity, coupling, and cohesion) do not show any improvement after an *explicit refactoring* with respect to their average daily changes.

In order to obtain more reliable and smoother results we do not simply compare the changes of productivity and quality metrics before and after the application of a refactoring. Such changes could happen by chance or because of some other factors we do not control within this case study (for example mood of developers, work on particular part of the code, problems with tools, other XP practices). To minimize the influence of random and uncontrolled changes we compare average productivities between development iterations (the one in which an explicit refactoring has been applied with the following iteration). Also for the quality metrics we compute their average daily changes and compare them with the changes induced by an *explicit refactoring*.

2.2 Explicit Refactorings, Productivity, and Quality

In more traditional development processes, refactoring is present in ordinary maintenance tasks or extraordinary maintenance projects, in order to improve software maintainability [20]. The context of our analysis however is an agile development process, namely a tailored version of Extreme Programming [2]; in such environment refactoring is an integral part of software development. Kent Beck illustrates the principle of agile development with the two hats metaphor: One is adding new functionality (coding) and the other is refactoring. The developer should swap frequently between these two hats but wear only one at a time. Therefore, we assume that developers apply small refactorings like Extract Method, Rename, Simplify Conditional, Move Method/Field, and so on [17] throughout development – without even documenting it. We believe that all these small refactorings improve slightly the quality of the code and increase overall development productivity compared to a development process, which does not use the practice of refactoring.

However, due to the lack of empirical data (of two comparable software projects, one developed using an agile and one using a traditional method) such comparison is out of scope of this research. Instead, we analyze the effect of *explicit refactorings* on productivity and quality within the same project. *Explicit refactoring* means that developers wrote *explicitly* a user story for refactoring tasks and that the implementation of such user story took a considerable amount of time – in our case even several hours.

For the time being, we do not identify different kinds of refactorings and analyze separately the impact of each type of refactoring on productivity or quality. After having defined what we intend by *explicit refactoring* we have to define the other variables of interest for this research, namely development productivity and metrics for software quality and in particular maintainability.

Lots of work has been done on how to measure developers' productivity [16]. However, no definitive measure has been found and perhaps such definite measure does not exist. A very simple measure of productivity is the ratio of lines of code (LOC) produced and effort in hours spent in producing them:

$$productivity = \frac{LOC}{Effort}$$

In this research we use this equation because of its simplicity and expressiveness. In addition, programmers are all working in good faith – they volunteered for this experiment, the effort spent in activities other than coding has been closely monitored and evenly distributed, code reuse has been closely scrutinized also via the CVS repository, and no code generators have been used.

Software quality is a composite property of many internal and external software attributes. There has been a lot of discussion on the meaning of software quality [23], [5]. It is now commonly agreed [16] that software quality is a property defined by several small-scaled and directly measurable attributes. In this research we use complexity, coupling, and cohesion metrics, as defined by Chidamber and Kemerer (CK) [10]; such measures are widely accepted both by practitioners and researchers and validated by several previous studies [4], [9]. In addition, such measures are easy to collect and to understand, a precondition for their effective use [19].

Software maintainability is related both to software quality (it is considered as a quality factor) and cost, as good maintainability of software reduces significantly maintainance effort [11]. An XP project is constantly in the state of maintainance [3], therefore, besides quality measures also evolution of development productivity is a good indicator for its maintainability. The CK metrics include measures for complexity (WMC) and coupling (CBO) of object-oriented systems: Both of them are related to software maintainability as an increase of software complexity and coupling deteriorates its understandability [18].

2.3 Data Collection

The software project we analyze was developed using an agile, XP-like methodology tailored by Abrahamsson *et al.* [2]. Therefore, data collection had to be **(a)** non-invasive to preserve the agile nature of the project itself [27], and **(b)** accurate and reliable for doing meaningful statistics.

In order to achieve these two goals we use the PROM tool [31] for collecting product and effort metrics. PROM is a fully automated measurement framework for software engineering processes and products. Source code metrics are extracted daily from the source code management system employed by the company. PROM enables the automatic collection of the effort associated with different tasks such as reading documents, browsing the web and coding. In particular, a plug-in for the IDE in place collects the time spent by developers for coding activities for individual methods and classes. Effort data for coding is collected as soon as the developer enters the cursor in the source code editor of the IDE and ends if the editor is off focus, the IDE is closed or the screensaver is activated. Moreover, PROM allows the user to specify if one or two programmers are sitting in front of a machine.

The notion of effort adopted in this context is strongly related to only coding activities and does not include the time spent discussing about the design/code on a whiteboard; however in an XP-like process, which itself assigns to coding activities the highest importance, this measure is a reasonable measure for development effort. Both source code metrics and effort data are integrated and stored automatically in a central database, from which we access the data for our analysis.

To collect the product and process metrics listed in Table 1 with the PROM tool, we adopt the following data collection procedure:

- Every day at midnight the source code metrics are extracted from a CVS repository.
- A plug-in for Eclipse (the IDE used by developers) collects automatically the time spent for coding on individual classes and methods.
- We identify the days on which explicit refactorings are applied from the user stories (described in the project plan).

Table 1 provides an overview of the information that come from PROM and is used in this research.

Table 1. Sample data collected by PROM and aggregated at a system level. All metrics are per day.

Day	**LOC**	**CK metrics**	**Effort (hour)**	**Productivity (LOC/hour)**
22	150	30, 7, 5, 3	4.24 h	35.3
...	…	...	...	…

We aggregate metrics at a system level (we add up all single classes) and compute their overall changes per day in the case of product metrics and the total time spent for coding per day in the case of effort. The way we aggregate metrics is a first approach and could be refined: We could for example identify the classes affected by refactoring using a technique presented in [12], [30] and use only them for analyzing changes in quality and productivity. Whether this would change our findings, has to be assessed in a future analysis.

2.4 Data Analysis Method

Our research design is to some extent a one-factor, repeated-measures design: The treatment (in our case refactoring) is applied twice to the same subjects. We use box-plots for comparing the means of different populations (before-after refactoring productivity). In addition we perform a Wilcoxon rank sum test [22], as we cannot assume a normal distribution and homogeneity of variance of data.

As regards the quality metrics we proceed in the following way: first, we compute their changes at the end of a day when developers applied an explicit refactoring with respect to the previous day. Then, we use a Wilcoxon Signed-Rank [22] test to conclude whether these changes are lower than the average daily changes per iteration or not. Our final goal is to disprove the null hypotheses by using the Wilcoxon Signed-Rank tests to determine (a) if the development productivity is higher after refactoring than before, and (b) if quality metrics are significantly improved by refactoring with respect to their average changes.

3 Case Study

In the following section, first we describe the context of the case study; afterwards, we present and discuss the results of our analysis.

3.1 Context of the Case Study

The object under study is a software project in an agile, close-to-industrial development environment ("close-to-industrial" refers to an environment where the development team is composed of both professional software engineers and students [2]). The result is a commercial software product developed at VTT in Oulu, Finland, to monitor applications for mobile, Java enabled devices. The programming language was Java (version 1.4) and the IDE was Eclipse 3.0. The project was a full business success in the sense that it delivered on time and on budget the required product.

Four developers formed the development team. Three developers had an education equivalent to a BSc and limited industrial experience. The fourth developer was an experienced industrial software engineer.

The development process followed a tailored version of the Extreme Programming practices [2], which included all the practices of XP but the "System Metaphor" and the "On-site Customer"; there was instead a local, on-site manager that met daily with the group and had daily conversations with the off-site customer. In particular, the team worked in a collocated environment and used the practice of pair programming. The project lasted eight weeks and was divided into five iterations, starting with a 1-week iteration, which was followed by three 2-weeks iterations, with the project concluding in a final 1-week iteration. Throughout the project, mentoring was provided on XP and other programming issues according to the XP approach. Since the team was exposed for the first time to an XP-like process, a brief training of target XP practices was given before the start of the project.

The total development effort per developer was about 192 hours (6 hours per day for 32 days). Since with PROM we monitored all the interactions of the developer with different applications, we are able to differentiate between coding and other activities: About 75% of the total development effort was spent for pure coding activities inside the IDE while the remaining 25% was spent for other assignments like working on text documents, reading and writing emails, browsing the web and similar tasks. The developed software consists of 30 Java classes and a total of about 1770 Java source code statements (LOC counted as number of semicolons in a Java program).

During development two user stories have been explicitly written for refactoring activities: One at the end of iteration two with the title "Refactor Static Classes to Object Classes" and one at the end of iteration four with the title "Refactor Architecture". We refer to the implementation of these two user stories as explicit refactorings; we analyze changes of productivity and quality measures before and after their completion.

3.2 H^0_A – Does Productivity Increase After "*Explicit Refactorings*"?

Figure 1 shows the evolution of the average productivity per iteration over the whole development period.

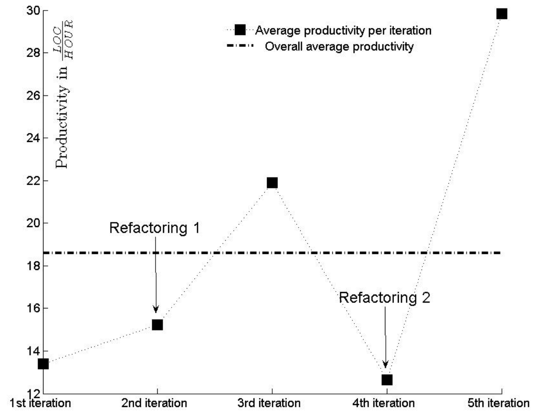


Fig. 1. Average development productivity per iteration

The productivity is almost the same in iterations 1, 2, and 4, (about 15 LOC/HOUR) while it is significantly higher in iterations 3 and 5 (more than 22 LOC/HOUR). This distribution is interesting for two reasons: First, productivity does not show a decreasing trend during software development as we were expecting due to higher effort for adding new functionality as the system's complexity and coupling is growing. Second, whenever developers perform an *explicit refactoring* – i.e. at the end of iteration 2 and at the end of iteration 4 – productivity of the following iteration is significantly higher than average productivity of the remaining iterations. For the data under scrutiny we can only measure changes of productivity after two *explicit refactorings* (treatments). With such small sample size statistical tests are hardly applicable as significance values are rather meaningless. Instead we prefer to use a box-plot for visualizing the difference in productivity in the before-after refactoring situations.

Figure 2 shows a box-plot of the average productivity per iteration throughout development. Moreover, we draw two dashed lines, one indicating the average productivity for the iteration following the first *explicit refactoring*, the other for the iteration following the second *explicit refactoring*. We can observe a clear improvement of productivity for both cases with respect to average productivity.

For the sake of completeness we perform a Wilcoxon rank sum test to compare productivity after the 2 refactorings with average productivity of the remaining iterations. As expected, given our small sample size, we obtain a p value of 0.2 meaning that we cannot reject H^0_A neither for refactoring 1 nor for refactoring 2. Overall, we can conclude that the productivity data sustain the claim that refactoring raises development productivity in the short-term, thus nullifying to some extent the complexity naturally added during development. However, this conclusion is more a confirmation of a suspicion and not a clear affirmation based on statistical inference from experimental data.

In order to consider the overall evolution of productivity throughout development, we compare the medians of the daily productivity of each iteration using a non-parametric Kruskal-Wallis test [22]. The result is that they are not statistically different from each other. In fact Figure 1 emphasizes that productivity is rather increasing than declining towards the end of the project.

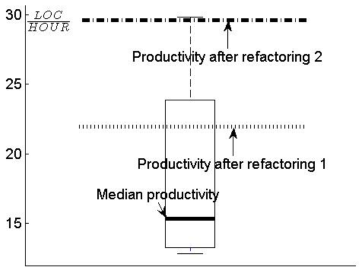


Fig. 2. Box-plot of average productivity per iteration

Altogether, our findings strongly advocate that refactoring of a software system raises subsequent development productivity and prevents in a long-term its deterioration.

3.3 H^0_B — Cohesion, Coupling and Complexity: Does Refactoring Improve Code Quality?

Findings of prior studies claim that refactoring improves some low-level quality metrics like coupling and cohesion measures [7]. In this research we look at the temporal evolution of the CBO, WMC, RFC, and LCOM metrics and how it is related to refactoring. A visual inspection of the evolution of these metrics (Figure 3) evidences that their changes, from one iteration to the next, tend to decrease starting from the second iteration (1st *explicit refactoring*) for the CBO and RFC metrics, and from the third for the LCOM and WMC metrics. This is a first indication that refactoring could limit

Table 2. p-values for the one-sided Wilcoxon Signed-Rank test for testing if the population mean of the median of the daily changes per iteration of CK metrics is higher than the changes after refactoring

	WMC	LCOM	CBO	RFC
Refactoring1	0.72	0.5	**0.03**	0.18
Refactoring2	**0.03**	**0.02**	**0.02**	**0.02**

the overall decrease of cohesion and increase of coupling and complexity metrics that we expect to occur during software development.

Table 2 gives the p-values (significant values at the 0.05 level are set in bold face) of the Wilcoxon Signed-Rank test for assessing whether or not the changes of the 4 CK metrics after the two *explicit refactorings* are the same with respect to their average changes: We can see that all of them improve after the second refactoring, since their changes are significantly lower (they are in fact negative) than the average of their daily changes. For the first refactoring this is only true for the coupling metric CBO. The results are not strong enough to reject H^0_B for both refactorings, but only for the second and in part for the first. Still, they provide confidence that with more comprehensive experimentation on larger projects it will be possible to significantly prove it.

Visually inspecting the plot (Figure 3) of the changes of LCOM, CBO, RFC and WMC per iteration, we also notice an interesting phenomenon: After an initial phase of remarkable growth of these metrics, they start to decrease, most likely thanks to refactoring. We interpret this as the people gathering a more comprehensive view of the application to develop, and thus being able to better refactor the system, creating simpler, less coupled, and more cohesive code. Moreover, by refactoring the system they acquire a better understanding of the program being developed, which could explain a boost in productivity (this observation is consistent with the findings of other researchers [8]). Yet, this is an interpretation based on a visual inspection rather than on a statistical test: only future research involving larger data samples will be able to assess its statistical significance and validity.

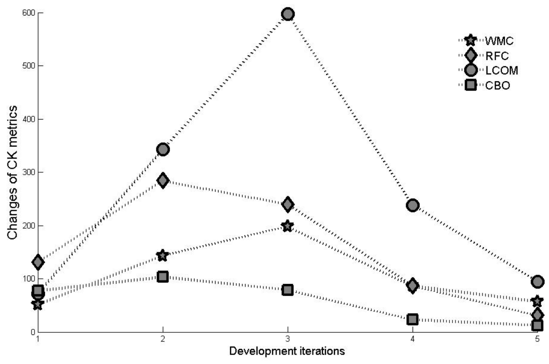


Fig. 3. Evolution of the average changes of LCOM, CBO, RFC, and WMC per iteration

Altogether, this research evidences that there are visual indicators (in part supported by statistical tests) that refactoring prevents an explosion of complexity and coupling metrics by driving developers to simpler design and as a consequence less complex and coupled and easier to maintain code.

4 Threats to Validity

This research aims at assessing the impact of refactoring on development productivity and software quality measures. The results of this research are particularly interesting as they come from a case study in a close-to-industry context: A situation, which is quite rare in software engineering. However, there are a number of threats to construct, internal and external validity of the study that have to be addressed properly. Those are in particular:

(a) First, the conclusions we draw depend strongly on the definition we give of productivity and software quality and its validity in industry. We define productivity as the ratio between source code statements and time spent for coding. Several objections have been raised against this measure: A malicious developer could artificially inflate the number of lines of code; only coding is considered ignoring all the other phases of development – analysis, design, etc; code reuse and automatically generated code are not taken properly into account; and other. Despite all the criticism, this equation is by far the most used in industry, as it is very easy to understand and gives clear and absolute numbers, which are easy to compare and to use in statistical calculations. Moreover, in the context of XP much emphasis is put on coding activities; thus, development effort coincides mostly with coding effort. In order to support the validity of the productivity measure used in this study it would be interesting to run similar studies using this definition but also a definition based on other parameters, for instance function points or user stories.

(b) As regarding to internal validity we have to be aware that with a single case study it is not possible to infer whether or not the observed relation is a causal one. We are not able to control and manipulate the influence of other confounding factors such as short release cycles or pair programming. For example, the observed increase in productivity after explicit refactorings could be explained differently: Maybe in the iterations following an *explicit refactoring* developers implemented "easy" user stories or did not do any refactoring at all (refactoring itself decreases to some extent productivity as measured in this study). Moreover, even if we were sure that refactoring is the cause for the observed improvements in productivity due to the small sample size such relation is of low statistical significance. However, confounding factors, which we identified in the context of this study, are averaged over iterations and should impact productivity and quality measures equally (i.e. independent of specific iterations). Therefore, we are confident that the observed effects are due to the explicit refactorings and that a larger study would provide necessary statistical significance, which is only suggested by our results.

(c) We do not consider different kinds of refactorings. Such coarse grained analysis could bias our results: Developers may for example apply only a limited subset of refactorings – due to their inexperience or other reasons – and in such case we can probably not generalize the implications for all other types of refactorings. We plan to take into account different categories of refactorings in a more refined, future study.

(d) We sum averaged quality metrics and productivity over all classes, whereas probably only a few of them have been affected by the two explicit refactorings. In doing so we could misinterpret the real impact of refactoring; we plan in a future work with a larger sample size to analyze the changes of productivity and code quality only for the classes that have been involved directly in a refactoring activity.

(e) The subjects of the case study are heterogeneous (three students and one professional engineer) and use for the first time an XP-like methodology. This could confound our findings, as for example students may behave very different from industrial developers. Moreover, also a learning effect could be visible and for example be the cause for the evolution of the productivity and quality metrics as shown, respectively, in Figure 1 and Figure 3. Developers were aware that they are monitored, but did not know that we measured in particular productivity before and after refactorings; we did not communicate them the objectives of the study as such knowledge could influence their behavior leading to higher productivities after refactorings.

(f) As with every case study, it is hard to generalize to other, larger contexts. We think that our findings are applicable to small teams working in similar, highly volatile domains (ours is application development for mobile devices). However, additional research is needed to ensure that this is indeed true and to generalize it to other contexts.

Furthermore, it would be interesting to analyze how much refactoring is "good enough" to keep productivity high and what kinds of refactorings are important to improve both productivity and quality.

5 Conclusions

Although agile processes and practices are gaining more importance in the software industry there is limited solid empirical evidence of their effectiveness. This research focuses in particular on the practice of refactoring, which is one of the key practices of Extreme Programming and other Agile Methods.

While the majority of software developers and researchers agree that refactoring has long-term benefits on the quality of a software product (in particular on program understanding) there is no such consensus regarding the development productivity. Available empirical results regarding this issue are very limited and not clear [1]. This might refrain managers from adopting refactoring, as they might be scared of loosing resources.

This work contributes to a better understanding of the effects of refactoring both on code quality – in particular on software maintainability - and development productivity in a close-to industrial, agile development environment. It provides new empirical, industrially based evidence that refactoring rather increases than decreases development productivity and improves quality factors, as measured using common internal quality attributes – reduces code complexity and coupling; increases cohesion. The implications on defects are not discussed, as such data are not available. Moreover, we do not contribute in exploring the linkage of refactoring to other external quality attributes. Clearly, this question has to be addressed in a future study.

As regards productivity, these results are in contradiction with the previous work of Abrahamsson and Koskela [1]. However, such older work addressed a case that was too limited to be taken as a reference. For internal quality metrics, our results are in accordance with the existing literature. Altogether, we believe that our findings are particularly relevant, as this work is a case study in a close-to-industry environment, a kind of empirical investigation that is rare for the research problem we discuss here. Clearly, this is a first work in the area. A real, generalizable assessment of the implications of refactoring requires several repetitions of studies like this, possibly also including data on defects.

The findings of this research have major implications for a widespread use of refactoring, as already mentioned by Beck in his first work on XP [3]. Of course, refactoring as any other technique is something a developer has to learn. First, managers have to be convinced that refactoring is very valuable for their business; this research should help them in doing so as it sustains that refactoring – if applied properly – intrinsically improves code maintainability and increases development productivity. Afterwards, they have to provide training and support to change their development process into a new one that includes continuous refactoring.

Case studies in close-to-industry contexts are very rare in software engineering and this gives us a remarkable confidence on the results that we have obtained. However, it is important to remember that, formally, such results are only valid in the specific context of the study. To achieve a high level of confidence of them, it is essential to replicate such case studies, also in other contexts and using different measures.

Acknowledgments. Special thanks go to development team at VTT, Oulu, Finland, which has been disposed to install and use the PROM tool for data collection and in the end, enabled this case study. The authors would also like to acknowledge the support by the Italian Ministry of Education, University and Research via the FIRB Project MAPS (http://www.agilexp.org) and the autonomous province of South Tyrol via the Interreg Project Software District (http://www.caso-synergies.org).

References

1. Abrahamsson, P., Koskela, J.: Extreme programming: Empirical results from a controlled case study. In: ACM-IEEE International Symposium on Empirical Software Engineering (ISESE 2004), Redondo Beach CA, USA (2004)
2. Abrahamsson, P., Hanhineva, A., Hulkko, H., Ihme, T., Jäälinoja, J., Korkala, M., Koskela, J., Kyllönen, P., Salo, O.: Mobile-D: An Agile Approach for Mobile Application Development. In: Proceedings 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, Vancouver, British Columbia, Canada (2004)
3. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, Reading (2000)
4. Basili, V.R., Briand, L.C., Melo, W.L.A.: Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering 22(10), 267–271 (1996)

5. Boehm, B.W., Brown, K.J.R., et al.: Characteristics of Software Quality. TRW Series of Software Technology. North-Holland, Amsterdam (1978)
6. Bois, B.D., Mens, T.: Describing the impact of refactoring on internal program quality. In: Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA), Amsterdam, The Netherlands (2003)
7. Bois, B.D., Demeyer, S., Verelst, J.: Refactoring – Improving Coupling and Cohesion of Existing Code. In: Belgian Symposium on Software Restructuring, Gent, Belgium (2005)
8. Bois, B.D., Demeyer, S., Verelst, J.: Does the "Refactor to Understand" Reverse Engineering Pattern Improve Program Comprehension? In: Proceedings 9th European Conference on Software Maintenance and Reengineering (CSMR 2005), Manchester, UK, March 21-23 (2005)
9. Briand, L.C., Wüst, J.: Modeling Development Effort in Object-Oriented Systems Using Design Properties. IEEE Transactions on Software Engineering 27(11), 963–986 (2001)
10. Chidamber, S., Kemerer, C.F.: A metrics suite for object-oriented design. IEEE Transactions on Software Engineering 20(6), 476–493 (1994)
11. Corbi, T.A.: Program Understanding: Challenge for the 1990s. IBM Systems Journal 28(2), 294–306 (1989)
12. Demeyer, S., Ducasse, S., Nierstrasz, O.: Finding Refactorings via Change Metrics. In: Proceedings of the 15th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2000, Minneapolis, USA (2000)
13. Demeyer, S.: Maintainability versus Performance: What's the Effect of Introducing Polymorphism? Technical report, Lab. on Reeng. Universiteit Antwerpen, Belgium (2002)
14. Van Deursen, A.: Program Comprehension Risks and Opportunities in Extreme Programming. In: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE 2001), Stuttgart, Germany, October 2-5 (2001)
15. Van Emden, E., Moonen, L.: Java Quality Assurance by Detecting Code Smells. In: Proceedings of the 9th Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos (2002)
16. Fenton, N., Pfleeger, S.L.: Software Metrics A Rigorous & Practical Approach. PWS Publishing Company, Boston (1997)
17. Fowler, M.: Refactoring Improving the Design of Existing Code. Addison-Wesley, Reading (2000)
18. Henderson-Sellers, B.: Object-Oriented Metrics: Measures of Complexity, p. 62. Prentice-Hall, Upper Saddle River (1996)
19. Johnson, P.M., Disney, A.M.: Investigating Data Quality Problems in the PSP. In: Proceedings of Sixth International Symposium on the Foundations of Software Engineering (SIGSOFT 1998) (1998)
20. Kataoka, Y., Imai, T., Andou, H., Fukaya, T.: A Quantitative Evaluation of Maintainability Enhancement by Refactoring. In: Proc. Int'l Conf. Software Maintenance, October 2002, pp. 576–585 (2002)
21. Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, P.E., Turski, W.M.: Metrics and laws of software evolution-the nineties view. In: Proceedings of the Fourth International Software Metrics Symposium, November 5-7 (1997)
22. Lehmann, E.L.: Testing Statistical Hypotheses. Springer, New York (1986)
23. McCall, J.A., Richards, P.K., Walters, G.F.: Factors in Software Quality. RADC TR-77-369, Vols I, II, III, US Rome Air Development Center Reports NTIS AD/A-049 014, 015, 055 (1977)
24. Mens, T., Demeyer, S., Bois, B.D., Stenten, H., van Gorp, P.: Refactoring: Current Research and Future Trends. Electronic Notes in Theoretical Computer Science 82(3) (2003)

25. Mens, T., Tourwé, T.A.: Survey of Software Refactoring. IEEE Transactions on Software Engineering 30(2), 126–139 (2004)
26. Pizka, M.: Straightening spaghetti-code with refactoring? In: Proceedings of the Int. Conf. on Software Engineering Research and Practice - SERP, Las Vegas, NV, pp. 846–852 (2004)
27. Poppendieck, T., Poppendieck, M.: Lean Software Development: An Agile Toolkit for Software Development Managers. Addison-Wesley, Reading (2003)
28. Ratzinger, J., Fischer, M., Gall, H.: Improving Evolvability through Refactoring. In: Proceedings 2nd International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA (2005)
29. Sahraoui, H.A., Godin, R., Miceli, T.: Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In: Proc. International Conference on Software Maintenance, pp. 154–162 (October 2000)
30. Schofield, C., Tansey, B., Xing, Z., Stroulia, E.: Digging the Development Dust for Refactorings. In: Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006), Athens, Greece (2006)
31. Sillitti, A., Janes, A., Succi, G., Vernazza, T.: Collecting, Integrating and Analyzing Software Metrics and Personal Software Process Data. In: Proceedings of the EUROMICRO 2003, Belek-Antalya, Turkey (2003)
32. Simon, F., Steinbruckner, F., Lewerentz, C.: Metrics based refactoring. In: Proc. European Conf. Software Maintenance and Reengineering, pp. 30–38. IEEE Computer Society Press, Los Alamitos (2001)
33. Stroulia, E., Kapoor, R.V.: Metrics of Refactoring-based Development: An Experience Report. In: The 7th International Conference on Object-Oriented Information Systems, Calgary, AB, Canada, pp. 113–122. Springer, Heidelberg (2001)
34. Tahvildari, L., Kontogiannis, K.A.: Metric-Based Approach to Enhance Design Quality through Meta-Pattern Transformations. In: Proc. European Conf. Software Maintenance and Reeng., pp. 183–192 (2003)
35. Yu, Y., Mylopoulos, J., Yu, E., Leite, J.C., Liu, L., D'Hollander, E.H.: Software refactoring guided by multiple soft-goals. In: Proceedings of the 1st workshop on Refactoring: Achievements, Challenges, and Effects, in conjunction with the 10th WCRE conference 2003, Victoria, Canada, November 13-16, pp. 7–11 (2003)

Modeling of Requirements Tracing

Matthias Heindl[1] and Stefan Biffl[2]

[1] Support Center Configuration Management, Siemens Program and Systems Engineering, Siemens AG Austria, Gudrunstrasse 11, A-1100 Vienna, Austria
`Matthias.a.Heindl@siemens.com`
[2] Institute of Software Technology and Interactive Systems, Vienna University of Technology, Favoritenstrasse 9/188, A-1040 Vienna, Austria
`Stefan.Biffl@tuwien.ac.at`

Abstract. Software customers want both sufficient product quality and agile response to requirements changes. Formal software requirements tracing helps to systematically determine the impact of changes and to keep track of development artifacts that need to be re-tested when requirements change. However, full tracing of all requirements on the most detailed level can be very expensive and time consuming. In the paper an initial "tracing activity model" is introduced along with a framework that allows measuring the expected cost and benefit of tracing approaches. In a feasibility study a subset of the activities belonging to the model has been applied to compare three tracing strategies: agile, "just in time" tracing, and fully formal tracing. The study focused on re-testing and it has been performed in the context of an industry project where the customer was a large financial service provider.In the study a) the model was found useful to capture costs and benefits of the tracing activities and to compare different strategies; b) a combination of tracing approaches proved helpful in balancing agility and formalism.

Keywords: Software Requirements Tracing, Re-Test, Tracing Activity Model, Feasibility study.

1 Introduction

The main goal of software development projects is to develop software that fulfills the requirements of most important stakeholders, i.e., customers and users. However, in typical projects requirements tend to change throughout the project, e.g. due to revised customer needs or modifications in the target environments. These changes of requirements may introduce significant extra effort and risk, which need to be assessed realistically when change requests come up, e.g. test cases have to be adapted in order to test the implementation against the revised requirements. Thus, software test managers need to understand the likely impact of requirement changes on product quality and needs for re-testing (regression testing) to continuously balance agile reaction to requirements changes with systematic quality assurance activities.

An approach to support the assessment of the impact of requirements changes is formal requirements tracing, which helps to determine necessary changes in the design and implementation as well as needs for re-testing existing code more quickly and accurately. Requirements tracing is formally defined as the ability to follow the life of a requirement in a forward and backward direction [11], e.g. by explicitly capturing relationships between requirements and related artifacts. For example, a trace between a requirement and a test case indicates that the test case checks code against the requirement.

Such traces can be used for change impact analysis: if a requirement changes, a test engineer can efficiently follow the traces from the requirement to the related test cases and identify the correct test cases that have to be checked, adapted, and re-run to systematically re-test the software product.

However, in a real-world project full tracing of all requirements on the most detailed level can be very expensive and time consuming. Thus, the costs and benefits to support the desired fast and complete change impact analysis need to be investigated with empirical data. While there are many methods and techniques on how to technically store requirements traces, there is very few systematic discussion on how to measure and compare the tracing effort and effectiveness of tracing strategies in an application scenario such as re-testing.

This paper proposes an initial *tracing activity model* (TAM), a framework to systematically describe and help determine the likely efforts and benefits, like reduced expected delay and risk, of the tracing process in the context of a usage scenario such as re-testing of software. The TAM defines common elements of various requirements tracing approaches: trace specification, generation, deterioration, validation, rework, and application; and parameters influencing each activity like number of units to trace, average effort per unit to trace, and requirements volatility.

The model can support requirements and test managers in comparing requirements tracing strategies, e.g. for tailoring the expected re-test effort and risk based on selected parameters: process alternatives, expected test-case creation effort, and expected change-request severity. We apply the TAM in a feasibility study that compares effort, risk, and delay of three tracing strategies: no tracing at all (no-T), full formal tracing (full-T) for re-testing, and value-based tracing (value-T).

The remainder of the paper is organized as follows: Section 2 summarizes related work on requirements tracing and requirements-based testing; Section 3 introduces the tracing activity model and research objectives. Section 4 outlines the feasibility study and summarizes the results. Section 5 discusses the results and limitations of the study and lessons learned; finally Section 6 concludes and suggests further work.

2 Related Work on Requirements Tracing and Re-testing

Several approaches have been proposed to effectively and efficiently capture traces for certain trace applications like change impact analysis and testing [1][4][7].

Many standards for systems development such as the US Department of Defense (DoD) standard 2167A mandate requirements traceability practice [23]. Gotel and Finkelstein [11] define requirements tracing as the ability to follow the life of a requirement in both a backward and forward direction. Requirements traceability is an issue for an organization to reach CMMI level 3 making tracing an issue that many maturing software development organizations have to consider: the assessment for maturity level 3 there contains questions concerning requirements tracing: whether requirements traces are applied to design and code and whether requirements traces are used in the test phases.

The tracing community, e.g., at the Automated software engineering (ASE) tracing workshop TEFSE [7][8], traditionally puts somewhat more weight on technology than on process improvement. Basic techniques for requirements tracing are cross referencing schemes [9], key phrase dependencies [18], templates, RT matrices, hypertext [20], and integration documents [21]. These techniques differ in the quantity and diversity of information they can trace between, in the number of interconnections between information they can control, and in the extent to which they can maintain requirements traces when faced with ongoing changes to requirements.

Commercial requirements management tools like Doors, Requisite Pro, or Serena RM provide the facility to relate (i.e. create traces between) items stored in a database. These tools also automatically indicate which artifacts are effected if a single requirement changes (suspect traces). However, the tools do not automate the generation of trace links (capturing a dependency between two artifacts as a trace), which remains a manual, expensive, and error-prone activity.

Watkins and Neal [24] report how requirements traceability aids project managers in: accountability, verification (testing), consistency checking of models, identification of conflicting requirements, change management and maintenance, and cost reduction.

Gotel and Finkelstein [11] also state the requirements traceability problem, caused by the efforts necessary to capture and maintain traces. Thus, to optimize the cost-benefit of requirements tracing, a range of approaches focused on effort reduction for requirements tracing. In general, there are two effort reduction strategies: (1) automation, and (2) value-based software engineering.

1. Automation. Multiple approaches have been developed to automate trace generation: Egyed has developed the Trace/Analyzer technique that automatically acquires trace links based on analyzing the trace log of executing key system scenarios [5][6]. He further defines the tracing parameters: precision (e.g., traces into source code at method, class, or package level), correctness (wrong vs. missing traces), and completeness. Other researchers have exploited information retrieval techniques to automatically derive similarities between source code and documentation [1], or between different high-level and low-level requirements [17]. Rule-based approaches have been developed that make use of matching patterns to identify trace links between requirements and UML object models represented in XML [25]. Neumüller and Grünbacher developed APIS [22], a data warehouse strategy for requirements tracing. Cleland-Huang *et al.* adopt an event-based architecture that links requirements and derived artifacts using the publish-subscribe relationship [3].

2. Value-based software engineering. The purpose of a value-based requirements tracing approach is not to reduce effort of each unit to trace (like automation) but to trace all requirements with varying levels of precision, and thereby reduce the overall effort for requirements tracing [13], e.g., high-priority requirements are traced with a higher level of precision (e.g., at source code method level), while low-priority requirements are traced with lower precision (e.g., at source code package level).

The effort used to capture traces should be justifiable with the effort that could be saved by using these traces in software engineering activities like change impact analysis, testing [4][16][19], or consistency checking. It is a matter of balancing agility and formalism to come close to an optimal level of cost-benefit [2]. The approaches described above serve the purpose of reducing the effort to capture traces.

$$\text{Effort for capturing tracing + effort for trace application by using traces} < \text{effort for trace application without using traces} \quad (1)$$

Equation 1 captures this idea from a value-based perspective: to achieve a positive return on investment of requirements tracing the effort of generating and using traces should be lower than the effort for a trace application without traces. Such trace applications are (amongst others) change impact analysis and re-testing [10]. Besides effort of capturing traces, reduction of risk due to missed traces and delay due to the need to update traces are criteria that determine the usefulness of tracing approaches for software engineering activities.

Changes of requirements affect test cases and other artifacts [12]. Change impact analysis is the activity where the impacts of a requirement's change on other artifacts are identified [18]. Usually, all artifacts have to be scanned for needed adoptions when a change request for a requirement occurs. A trace-based approach relates requirements with other artifacts to indicate interdependencies. These relations (traces) can be used during change impact analysis for more efficient and more correct identification of potential change locations.

In [13] we proposed an initial cost-benefit model, where the following parameters that influence the cost-benefit of RT are identified: number of requirements and artifacts to be traced, volatility of requirements, and effort for tracing. In [14] we further discussed the effects of trace correctness as parameter influencing the cost-benefit of RT. However, tracing activities were not modeled explicitly, which would facilitate a more systematic discussion of the merits of different tracing approaches.

3 An Initial Tracing Activity Model

Most work in requirements tracing research has focused more on technology than on processes supported by this technology to generate and use traces. For the systematic comparison of tracing alternatives we propose in this section a process model, the tracing activity model (TAM), which contains the activities and parameters found in research tracing approaches; we label the framework as initial, although it is based on a systematic review literature and tracing activities in practice, as the external validation process has not yet concluded. The model can be used as basis to formally evaluate and compare tracing approaches as well as their costs and benefits.

3.1 Tracing Process Variants, Activities, and Parameters

The tracing activity model (TAM) in Figure 1 depicts a set of activities to provide and maintain the benefits of traces over time. The model is a framework to measure the cost and benefit of requirements tracing in order to compare several tracing strategies for a development project. The framework is based on previous work that identified tracing parameters, e.g., [3][7][8][13][14][17].

The activities in the model are building blocks identified from practice and literature and follow the life cycle of a set of traces.

Trace Specification is the activity where the project manager defines the types of traces that the project team should capture and maintain. For example, the project manager can decide to capture traces between requirements, source code elements, and test cases. This activity influences tracing effort based on the following parameters: Number of artifacts to be traced, number of traces, and artifacts to be traced. Other relevant parameters are tracing scope, precision of traces [7][8][13],

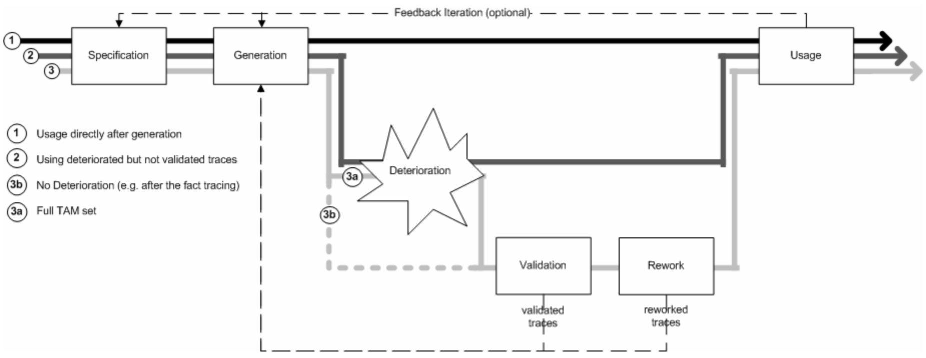

Feedback Iteration (optional)
1
2
3
Specification
Generation
Usage
1 Usage directly after generation
2 Using deteriorated but not validated traces
3b No Deterioration (e.g. after the fact tracing)
3a Full TAM set
3a
Deterioration
3b
Validation
Rework
validated traces
reworked traces

Fig. 1. Tracing activity model: Activities and process variants

Trace Generation. Trace generation is the activity of identifying and explicitly capturing traces between artifacts. Methods for trace generation range from manually capturing traces in matrices or requirements management tools that automatically create traces between artifacts based. The effort to generate traces in a project depends on the following parameters [13]:

- Number of requirements: in a software development project; the effort for tracing increases with increasing number of requirements.
- Number of artifacts to be traced to the higher the number of artifacts, the higher is the effort to create traces between them.
- Average trace effort per unit to trace, which depends on the used tools and the point in time of tracing.

Other relevant parameters are: number of traces, tool support, point in time of trace generation in the software development process, complexity/size of tracing objects, value of traces [13], correctness and completeness of traces.

Trace Deterioration. Trace deterioration is more the impact of external events than an activity. Traces can degrade over time as related artifacts change. If only the artifacts are updated, e.g., due to change requests, and the traceability information is not updated, the set of existing traces is likely to get less valid over time. Deterioration of traces affects the value of traces, because it reduces the correctness and completeness of traces.

Trace Validation and Rework. Trace validation is the activity that checks if the existing traceability information is valid or needs to be updated, e.g., identify missing trace links. In the example above, when artifact *A* changes fundamentally so that there is no longer a relationship to artifact *B*, trace validation would check the trace between *A* and *B* and flag it as obsolete. Trace validation is necessary to keep the trace set (traceability information) correct and up to date, so that the traces are still useful when used, e.g., for change impact analyses. We call the updating of traces "trace rework". Trace validation and trace rework are often performed together as they ensure correct and up-to-date traces and counter trace deterioration effects. The effort for validation and rework depend partly on the volatility of requirements.

The tracing activities are not necessarily performed in sequence. Furthermore, some activities are mandatory, like trace generation, whereas other activities are optional, as indicated by the arrows in Figure 1:

- *Trace Usage directly after generation (process variant 1 in figure 1):* Trace deterioration depends on the changes made to certain artifacts. If traces stay valid over time and do not deteriorate, validation and rework are not necessary so that the existing traces can be used, e.g., for change impact analyses.
- *Using deteriorated traces (process variant 2 in figure 1)* without validating and reworking them before is possible, but reduces the traces' benefits, because wrong or missing traces may hinder the supported activity more than they help
- *No Deterioration (process variant 3b in figure 1):* Traces can be validated after generation whenever the project manager wants, even when they did not deteriorate.

Trace Usage. Finally, traceability information is used as input to tracing applications like change impact analysis, testing, or consistency checking [24]. The overall effort of such a tracing application is expected be lowered by using traces. The benefits of tracing during trace usage depend on parameters explained in [15].

The cost-benefit of requirements traceability can be determined as the balance of efforts necessary to generate, validate, and rework traces (cost); and saved efforts during trace usage, reduced risk and delay of tracing (benefits during change impact analysis). To maximize the net gain of requirements tracing the effort of generating, validating and reworking traces can be minimized, or the saved effort of trace usage can be maximized.

3.2 Research Objectives

The value of tracing comes from using the trace information in an activity such as retesting that is likely to be considerably harder, more expensive, or to take longer without appropriate traces. If a usage scenario of tracing is well defined, trace generation can be tailored to provide appropriate traces more effectively and efficiently. Keeping traceability in the face of artifact changes takes further maintenance efforts.

The tracing activity model allows to formally define tracing strategies for a usage scenario by selecting the activities to be performed and by setting or varying the activity parameters.

We address the following research question:

- *RQ1: How useful is the TAM to model requirements tracing strategies and to determine and compare their efforts?*
- *RQ: To what extent can we balance the agility of a re-testing approach without using traces and the formalism of a systematic tracing approach for re-testing with a value-based approach?*

In order to evaluate the usefulness of the tracing activity model we conducted a small feasibility study in the finance domain, where we applied the TAM to 3 tracing strategies for the trace application re-testing. We discussed the usefulness of the re-testing strategies and the tracing model with development experts. If useful, the lessons learned from our evaluation could be a basis for extrapolation of tracing strategies and cost-benefit parameters to larger projects.

Re-testing is a software engineering activity that can be supported well by requirements tracing. The goal of a trace-based testing approach can be to make testing less expensive, less risky, and to reduce the delay. For a positive return on investment of tracing the effort to generate and maintain traces plus the effort of re-testing has to be lower than the effort of testing without tracing support.

4 Application of the TAM in an Industrial Feasibility Study

This section describes a feasibility study to validate the initial TAM framework concept. Together with practitioners from the quality assurance department of a large financial service provider we modeled 3 tracing strategies by using TAM building blocks and parameters and calculated tracing efforts of each strategy, their risks and delay in order to support the practitioners in deciding which tracing strategy provides the best support for re-testing in the practitioners' particular project context. This section describes an overview how we modeled each tracing strategy; detailed information of the study context can be found in the technical report [15].

The main focus of the study was to compare the efforts of each tracing strategy and the expected benefits of trace usage for re-testing. The TAM output variables were (1) the total effort of re-testing, (2) the risk of each strategy, and (3) the delay. Input variables were parameters covered the number of test cases, effort to create a trace, effort to create a test case, change impact analysis effort, etc. (see [15] for a comprehensive list of parameters).

Based on discussions with the experts in the industry environment and suggestions from literature we defined and compared 3 tracing strategies for re-testing: no tracing at all (no-T), full formal tracing (full-T), and value-based tracing (value-T). The data from this study can provide an initial snapshot in a typical scenario to find out whether the framework are useful to provide data and the proposed tracing strategies seem worthwhile for further discussion.

No tracing at all (no-T). As a baseline strategy we used the no-T strategy, which was the standard strategy in the feasibility study context; in this traditional re-testing process there is no trace support. Thus the activities of the tracing activity model are not performed and re-testing has to cope without traces: For each change request, the testers create new test cases instead of re-using and adapting existing ones. Obsolete test cases are replaced by new ones in order to avoid the risk of having redundant or inconsistent test cases, and to make sure everything is tested and test cases are still valuable after the change.

$$E(\text{no-T}) = \#cr * \#tc * tcn + dor. \tag{2a}$$

$$\textbf{E(no-T)} = 20 \text{ change requests} * 6 \text{ test cases} * 1\text{hrs} + 6*700*8 \text{ min} = 120 \text{ hrs} + 560 \text{ hrs} = \textbf{680 hrs}. \tag{2b}$$

Equation (2a) calculates the overall re-testing effort following the no-T strategy: for each change request (#cr), new test cases are created with the expected effort (#tc* tcn). Finally, the testers have to check newly created test cases with existing test cases and delete redundant (obsolete) old test cases (dor).

In the particular study the total effort for no-T was as calculated in Eqn 2b (see [15] for detailed explanation.

Full formal tracing (full-T) for re-testing. In the full-T strategy, testers systematically establish traceability by relating requirements and test cases (full tracing) via a tool, the Mercury Test Director. When a change request occurs, they check, and adapt existing test cases whenever possible; else they create new test cases.

$$E(\text{full-T}) = \#tntc * te + cia\_T * \#cr + tcnra * \#tc * \#cr \tag{3}$$

Equation (3) calculates the overall re-testing effort following the full-T strategy: The formula consists of 3 parts: (a) upfront traceability effort (#tntc * te), which establishes traceability for each existing test case, (b) the effort to identify affected test cases for each change request (cia_T * #cr), and (c) the effort needed to either reuse (tcr) or adapt (tca) existing test cases, depending on the severity of the change requests (#cr). If existing test cases can neither be reused nor adapted, new test cases have to be developed (tcn).

The shares of test cases that can be reused, adapted, or need to be created anew typically has an important impact on the overall effort of re-testing.

The effort of full-T for change impact analysis depends on how many traces between requirements and test cases can be reused, have to be adapted, or must be created. These values depend on the type of change request, as not every change request effects artifacts in the same way, e.g., there are simple low-effort change requests, e.g., affecting locally the user interface, whereas more severe change requests may need more extensive adaptations in several software product parts. Eqn 4a and 4b depict the efforts for full-T.

$$\text{CIA\_T effort overall} = 54 + 86 + 33 \text{ hrs} = 173 \text{ hours} \tag{4a}$$

$$\textbf{E(full-T)} = \text{upfront trace effort} + \text{CIA\_T} = 350 + 173 = \textbf{523 hrs} \tag{4b}$$

Based on effort reports for typical change requests in the case study context we categorized change request into the classes: Mini (small), Midi (medium), and Maxi (severe) (see [15] for details).

Value-based tracing (value-T) is a hybrid between full-T and ad-hoc tracing. Usually the upfront effort for full-T is considerably high, because all existing requirements have to be traced to test cases. value-T tries to reduce this tracing effort by establishing traceability on a coarse level (to test case packages instead of particular test cases) and to refine them ad-hoc when necessary. That means that all requirements are traced to test case packages and when change requests occur for some requirements, the traces from these test cases are refined to particular test cases to improve change impact analysis. Equation (5) calculates the overall re-testing effort following the value-T strategy:

$$\text{E(value-T)} = \text{upfront trace effort (on package level)} + \text{change impact analysis (value-T)} \quad (5a)$$

$$\textbf{E(value-T)} = 70 \text{ hrs} + 325 \text{ hrs} = \textbf{395 hrs} \quad (5b)$$

The upfront tracing effort for value-T is lower since traces have to be captured on more coarse level of detail than with full-T (70 hrs in comparison to 350 hrs with full-T). The change impact analysis effort for value-T consists of refining traces from changing requirements to the affected test case packages. The effort for identifying particular test cases by refinement was 325 hrs in the study resulting in a total effort of 395 hrs for the value-based tracing strategy to support re-testing.

5 Discussion

The purpose of the case study was to evaluate the feasibility of the TAM to model tracing strategies, in our case with focus on effort, also considering delay, and risk.

For practical reasons, the case study size and context was chosen to allow evaluating the approaches in a reasonable amount of time. However, the case study project setting seems typical in the company and financial service sector; the project context allows reasonable insight into the feasibility of the trace-based re-testing strategy in this environment.

In the feasibility study project, we deliberately applied a simple process variant from the TAM focusing on the activities trace specification, trace generation and the usage of generated traces for re-testing. Trace deterioration, validation and re-work were not enacted; rather we assumed for trace usage all generated traces to be correct. While this reduction of scope limits the experience this focus was found beneficial to make sure that the proposed process is actually applied in the practical setting.

As with any empirical study the external validity of only one study can not be sufficient for general guidelines, but needs careful examination in a range of representative settings. Furthermore, we analysed only a simple instantiation of the tracing activity model in the case study; consisting of trace generation and trace usage, but without considering trace deterioration, and consequently neither trace validation nor rework. In practice incorrect traces and trace deterioration can considerably lower tracing benefits and need to be investigated.

Modelling the 3 tracing strategies by using TAM activities and parameters provided data points for effort of each strategy. As these are single data points in a specific study setting, we see the results as snap shots, which should motivate further data collection to allow statistical data analysis and sensitivity analysis.

Comparing the 680 person hours effort of the no-T strategy, where new test cases are created for each test case, with the full-T alternative, with 523 person hours, full-T takes around 20% less effort. In this case the upfront investment into traceability pays off. In many cases, full tracing (tracing each requirement to each relevant test case) can cause considerably high effort which may prevent tracing in practice. Here, the study results suggest that the value-based strategy to trace requirements to test cases on a coarse level and refine them later on demand to be a promising approach that can significantly save efforts.

Besides effort, the alternatives also differ in delay when traces can be used for the trace application, in our case re-testing. value-T has a larger delay, because trace refinement has to be done before re-test. Concerning risk, no-T would be more risky if obsolete test cases were not checked. Inconsistent or redundant test case sets could then result in increased hidden testing effort or lower-quality test sets.

Lessons Learned from the Feasibility Study. The tracing activity model was found useful for systematically modeling the tracing alternatives, e.g., no tracing, systematic full tracing, and value-based tracing for the certain tracing application re-testing. The model helps make alternative strategies comparable, as it makes the main tracing activities explicit and allows mapping relevant parameters that influence tracing costs and benefits. Some input parameters (like number of change requests in the project, or effort to create a test case) had to be estimated based on practitioners' experience. Other data elements could be measured in the project context, e.g., number of requirements. The TAM allows choosing from the listed tracing activities and parameters and selecting the relevant ones to model tracing strategies for a particular usage scenario.

According to the expert feedback the calculated efforts provide a good input to reason about which tracing strategy seems most beneficial in a particular project context.

The lessons learned of our study for trace-support change impact analysis are:

- TAM provides useful building blocks for reasoning about relevant parameters (efforts, risks, etc.) of a tracing strategy and estimation of outcomes in advance helps to rationally discuss candidates for the best-fitting strategy.
- The volatility of traces is a major risk for full tracing. In volatile parts of the project, agile (just in time) or value-based approaches are favorable as full tracing has a particularly high risk of loosing upfront investments in tracing in these volatile areas.
- Full tracing provides detailed traces, which are particularly useful for situations when artifacts are not volatile and quick feedback is at a premium, e.g., for comprehensive cross checks at milestone reviews.
- If calculated efforts of tracing strategies do not differ significantly, choose a value-based strategy to provide full (complete) traceability at a coarse level of detail. This coarse-level traceability can than be refined on demand with reasonable total effort for change impact analysis.

6 Conclusion and Further Work

In the paper we proposed an initial tracing activity model (TAM) as a framework for defining and comparing tracing strategies for various contexts. For each tracing activity, relevant parameters were identified from related work and practice and mapped into the model. The model allows to systematically compare tracing strategy activities, their costs and benefits. We performed a small study in the financial service domain, where we evaluated the feasibility of the tracing activity model.

Main results of the study are: a) The model was found useful to capture costs and benefits of the tracing activities and compare different strategies; b) for volatile projects or project parts just-in-time tracing seems favorable; c) for parts that need quick feedback detailed upfront preparation of traces can be warranted; d) a combination of upfront tracing on a coarse level of detail (e.g. package or class level) and just-in-time detailed tracing of really needed traces can help balancing agility (important from the project point of view) and formality (that allows evidence-based software process improvement and is important from the software organization point of view).

Further work will be a) to use TAM as a framework for a systematic literature review concerning requirements tracing and b) to apply TAM for studies on tracing strategies in other contexts.

References

1. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. IEEE Transactions on Software Engineering 28(10), 970–983 (2002)
2. Boehm, T.: Balancing Agility and Discipline. Addison Wesley, Reading (2005)
3. Cleland-Huang, J., Zemont, G., Lukasik, W.: A Heterogeneous Solution for Improving the Return on Investment of Requirements Traceability, RE 2004, pp. 230–239 (2004)
4. Elbaum, S., Gable, D., Rothermel, G.: Understanding and Measuring the Sources of Variation in the Prioritization of Regression Test Suites, IEEE METRICS (2001)
5. Egyed, A.: A Scenario-Driven Approach to Trace Dependency Analysis. IEEE Transactions on Software Engineering 29(2) (February 2003)
6. Egyed, A., Grünbacher, P.: Automating Requirements Traceability: Beyond the Record & Replay Paradigm. In: Proceedings 17th International Conference on Automated Software Engineering, ASE 2002, Edinburgh, pp. 163–171 (2002)
7. Egyed, A., Biffl, S., Heindl, M., Grünbacher, P.: Determining the cost-quality trade-off for automated software traceability. In: ASE 2005, pp. 360–363 (2005)
8. Egyed, A., Biffl, S., Heindl, M., Grünbacher, P.: A value-based approach for understanding cost-benefit trade-offs during automated software traceability. In: Proc. 3rd int. workshop on Traceability in emerging forms of SE (TEFSE 2005), Long Beach, California (2005)
9. Evans, M.W.: The Software Factory. John Wiley & Sons, Chichester (1989)
10. Frankl, P.G., Rothermel, G., Sayre, K.: An Empirical Comparison of Two Safe Regression Test Selection Techniques. In: Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE 2003) (2003)
11. Gotel, O.C.Z., Finkelstein, A.C.W.: An analysis of the requirements traceability problem. In: 1st International Conference on Requirements Engineering, pp. 94–101 (1994)

12. Harker, S.D.P., Eason, K.D.: The Change and Evolution of Requirements as a Challenge to the Practice of Software Engineering. IEEE, Los Alamitos (1992)
13. Heindl, M., Biffl, S.: A Case Study on Value-Based Requirements Tracing. In: Proc. ESEC/FSE, pp. 60–69 (2005)
14. Heindl, M., Biffl, S.: The Impact of Trace Correctness Assumptions. In: 5th ACM/IEEE International Symposium on Empirical Software Engineering 2006 (ISESE 2006) (2006)
15. Heindl, M., Biffl, S.: An Initial Tracing Activity Model to Balance Tracing Agility and Formalism - Requirements Tracing Strategies for Change Impact Analysis and Re-Testing, Technical Report, TU Wien (2007), `http://qse.ifs.tuwien.ac.at/publications`
16. Hsia, P., Gao, J., Samuel, J., Kung, D., Toyoshima, Y., Chen, C.: Behavior-based Acceptance Testing of Software Systems: A Formal Scenario Approach. IEEE, Los Alamitos (1994)
17. Huffman Hayes, J., Dekhtyar, A., Karthikeyan Sundaram, S.: Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods. IEEE Trans. on Software Engineering 32(1) (January 2006)
18. Jackson, J.: A Keyphrase Based Traceability Scheme. IEE Colloquium on Tools and Techniques for Maintaining Traceability during Design, 2-1–2-4 (1991)
19. Juristo, N., Moreno, A.M., Vegas, S.: Reviewing 25 Years of Testing Technique Experiments. Journal Empirical Software Engineering 9(1-2), 7–44 (2004)
20. Kaindl, H.: The Missing Link in Requirements Engineering. ACM SigSoft Soft. Eng. Notes 18(2), 30–39 (1993)
21. Lefering, M.: An Incremental Integration Tool between Requirements Engineering and Programming in the Large. In: Proc. IEEE International Symp. on Requirements Engineering, San Diego, California, January 4-6, pp. 82–89 (1993)
22. Neumüller, C., Grünbacher, P.: Automating Software Traceability in Very Small Companies: A Case Study and Lessons Learned. In: Proc. IEEE Automated SE 2006, pp. 145–156 (2006)
23. Ramesh, B., Powers, T., Stubbs, C., Edwards, M.: Implementing Requirements Traceability: A Case Study. IEEE, Los Alamitos (1995)
24. Watkins, R., Neal, M.: Why and how of Requirements Tracing. IEEE Software 11(7), 104–106 (1994)
25. Zisman, A., Spanoudakis, G., Perez-Minana, E., Krause, P.: Tracing software requirements artefacts (2003)

Support for Cooperative Design of End-User Tailorable Software

Jeanette Eriksson

Blekinge Institute of Technology, School of Engineering,
P.O. Box 520 S-37225 Ronneby
jeanette.eriksson@bth.se

Abstract. Many contemporary business areas are dynamic and characterized by change. End-user tailorable software that allows the users to continue its evolution and adjustment is suitable in such environments. Unfortunately, the changes in the environment make it hard to know what flexibility to build into the software. The study presented here was aimed at providing an instrument that makes it possible to distinguish between different types of end-user tailoring, and to support discussions between users and developers concerning what kind of tailorability to build into the software. The study was performed in cooperation with a telecom company where tailorable software is essential to keep up with the fast changing market. The study resulted in ten attributes characterizing end-user tailorable software and a matrix capturing the values of the attributes. The matrix can be used as a guide and a basis for design decisions when implementing end-user tailorable software.

Keywords: Attributes of end-user tailoring, Design support, User participation.

1 Introduction

A fast changing world requires more and more flexibility in software, to provide support for higher reusability and to prevent the software from expiring too fast. One way to provide this kind of flexibility is via end-user tailoring. A tailorable system is modified while it is being used, as opposed to being changed during the development process. Tailoring a system is "continuing designing in use" [10, p. 223]. It is possible for a user to change a tailorable system through the support of some kind of interface.

Tailorable software is needed when the environment is characterized by fast and continuous change. As Stevens and his colleagues put it "The situatedness of the use and the dynamics of the environment make it necessary to build tailorable systems. However, at the same time these facts make it so difficult to provide the right dimensions of tailorability." [19, p. 273]. The study presented in this paper aims at providing an instrument that can support the work of finding the right dimension of tailoring when designing end-user tailorable software.

When discussing what we here call *tailorability* with people in industry, they seldom think of or talk about this kind of software in terms of tailoring; instead they simply call it flexibility. When observing work with tailorable software, or performing interviews or discussing tailorable software with people in industry, it emerged that there was confusion in the discussions between users and developers when they discussed flexibility. The reason for this is that they view flexibility from different perspectives. Flexible software is one thing when using it and a totally different thing when building the software. Accordingly, we have to look at tailoring from both the system and the user perspective, [20] as the user perspective reflects how users work with tailoring and the system perspective elucidates important issues from the developers' point of view.

There was even misunderstanding amongst the developers themselves. The reason for this was revealed to be the fact that the perspective on the software seamlessly alters between a system and user perspective. The developers in particular make this shift without thinking. The reason for this is of course that they have to take both perspectives into account in order to make good software. The fact that the differences between the two perspectives are considerable and that they are unaware of the shift in perspectives makes discussions about flexibility very complex. Under such circumstances it is hard to reach a consensus about which flexibility to implement whilst at the same time being convinced that the chosen type of flexibility is best for the situation. To make software successful it is important that there is a consensus between users and developers about how the system must work. Users and developers must have a common understanding of the phenomenon to reach a valid agreement [15]. If both developers and users understand tailoring and the differences inherent in it, then it is easier to discuss design issues and to make informed design decisions.

From an industrial perspective we end up with two issues to be dealt with:

- It is hard to know which dimensions of tailoring to implement.
- It is hard to discuss tailoring, since users and developers have different understandings of the phenomenon.

There are several conditions concerning user knowledge, technical issues and business organization that must be fulfilled to make a tailorable system work in the long run, and the tailorable software has to be supported by a collaboration between developers and users [5]. The development of tailorable software is an ongoing process where users are co-designers, [7] as it is users who evolve the software at use time. This kind of ongoing design can be called Meta Design [7]. Meta-Design is a development process where stakeholders are co-designers. Participatory Design (PD) [18] is another paradigm that includes stakeholders in the design process. PD has historically focused on involving users in the design process at the time for design, but the Participatory Design focus can be broaden to even include user involvement in design during use time [8]. Informed Participation [3, 4] is related to PD, as Informed Participation also lets people other than developers collaborate in design efforts. Informed Participation addresses open-ended design issues and tries to obtain an ownership of the problems among participants and to make the participants actively contribute to the design activities. The matrix presented in this paper is intended as support for *informed participation* in a development project. Often users' participation in development projects is mainly concerned with the user interface. We agree with [11] that the users' view of the system is not only the interface. Task related needs are what motivate end users to make changes to the system [14].

Human-centred design is necessary when designing tailorable software, since the users are co-designers. The users bring profound knowledge of the business process and organizational issues into the development project, which should be used in the design of the technical solution [9]. Gasson [9] also argues that there is a need for a dialectic process between organizational problems, implementation of changes in the business process and technical solutions to achieve a balance between human-centeredness and the design of technical solutions. The study presented in this paper aims at providing an application of Gasson's ideas in the context of tailorable software. The application, or matrix, is targeted to deal with the issues of deciding what dimension of tailoring to implement, by supporting a common understanding of end-user tailoring among users and developers.

A classification is a useful tool to aid an understanding of a phenomenon such as tailoring. A classification consisting of four categorises of tailoring is presented in [6]. The categorization is designed to take both the user and system perspectives into account, so that it can act as a basis for communication between developers and users when designing tailorable software. The categorization presented in [6] was found to be promising for use in industry. The categorization of end-user tailorable software is intended as a means of communications to involve the users more in the design process and is therefore suitable as a basis for supporting the cooperative design of end-user tailorable software.

The categorization is presented in Section 2. The formulation of the categories is at a rather abstract level and to make the categorization more precise and easier to use in practice, the categories should be assigned tangible attributes or characteristics. The idea is that the attributes of the categories can guide you to the most appropriate type of tailoring for a specific situation after you have pinpointed what type of business environment the software will be a part of, the skill and knowledge of the users and how much the developers are able to contribute to the tailoring process after the software has come in use.

In summary we have two research questions to answer to be able to deal with the industrial problems discussed above:

1. Which attributes characterize end-user tailorable software?
2. How can different dimensions of end-user tailoring be distinguished?

To answer the questions, a study was performed in cooperation with a major telecom company in Sweden. Developers and users were interviewed to elucidate which attributes are relevant to describe tailoring and how they perceive different kinds of end-user tailoring.

The rest of the paper is structured as follows. The next section will present the categorization of tailoring that acts as a base for the study. Section 3 describes the research method applied. The results of the study are presented in Section 4. The section consists of two parts, each answering one of the research questions. The first research question resulted in ten attributes characterizing end-user tailorable software and the second research question resulted in a matrix summarizing the values of each of the attributes for the four different categories of tailoring. The matrix can be used to *support the cooperative design process* when designing tailorable software. Finally, the paper ends with a discussion and conclusions.

2 Categorization of Tailoring

The categorization proposed by Eriksson et al. [6] is intended as a means of communication between developers and users in situations when deciding what kind of tailorability to implement. The categorization takes into account both a user perspective and a system perspective. The user perspective represents which changes can be made or the purpose of the activity, while the system perspective corresponds to how the change is achieved in the system (on a high level). The categorization is shown in Table 1.

Table 1. Categorization of tailorable software

	User Perspective	System Perspective
Customization	The end-user makes small changes, e.g. sets parameter values.	Parameter Values are interpreted and used in existing code.
Composition	The end-user relates different existing components to each other.	The relationships between the components are defined by a composition language. (It does not matter which programming language)
Expansion	The end-user creates a new component.	Components are integrated into the software by the implementation language and the new component does not differ from the pre-existing components. The composed component is used as a starting point for further tailoring. The software may generate code that is added to the pre-existing code, or incorporate the new component into the application in some other way.
Extension	The end-user adds code to the software.	New code (implemented by the end-user) is added to the pre-existing code. The application may also generate code to integrate the end-user's code into the software.

Customization is the simplest way of doing tailoring. It means that the user sets some values of one or more parameters and those parameters manage what functionality that is used. *Composition* means that the user has a set of components to choose from and he or she can connect them in specific ways to gain the desired functionality. *Expansion* also means that the user chooses components from a set, but the difference is that the users' combination of components is build into the system to become an integrated part. The new component is treated in the same way as the predefined components and will be accessible in the set to choose from next time the software is tailored. *Expansion* is the category which provides for the highest flexibility. It means that the user writes code that is integrated into the system either by wrapping up the new code into system generated code or, if written in a predefined way, through simply adding it to the code mass of the software. The user can either write the code in a high level language or in a visual programming language.

3 Research Method

Tailoring is especially well suited for applications used in a business environment that is characterized by fast changes, such as that in the telecom business. For example, new services continuously evolve and the supporting business systems therefore have to adapt to the altered requirements. The study was performed in cooperation with a telecom operator in Sweden. The company is dependent on flexible software that allows the user to alter the software when the need arises. Accordingly they have many tailorable systems running. The study aimed to elucidate (1) which attributes can be ascribed to tailorable software and (2) how different types of tailoring can be distinguished from each other. To achieve this, interviews were conducted where the categorization was used as a basis for the interviews.

We interviewed six developers and four users at the company. The developers were programmers, system owners and technical project leaders. The users all worked with several different systems, but their main tasks were within the same system. The users were a system coordinator, a work manager, users responsible for working with new requirements, and users helping out with further development of the system.

The interviews lasted approximately one hour to one and a half hours. Since a pilot study made it clear that it might be necessary to elaborate on some of the questions, we performed semi-structured interviews [17] which means that the respondents were asked the same questions in the same order, but follow-up questions were asked and explanations were given.

To be able to discuss the four categories on equal terms with both developers and users, the categories were translated into four representative examples. The examples were at a rather high level, free from unnecessary details, but concrete enough to allow the respondents to discuss the examples. The examples were not confined to the tasks in the telecom company. A summary of the examples in English can be found at http://www.ipd.bth.se/jer/tailoring/examples.htm

The interviews were audio taped and transcribed in full to provide for traceability. The individual transcriptions and the analysis of the material were sent to the respondents for verification.

3.1 Design of Interviews

The researcher interviewed one respondent at a time. First the developers were interviewed, and then the users. The interviews were conducted according to a specific sequence. First the respondents read the examples of the different categories and thereafter they were asked if they could spontaneously assign attributes and qualities to the first example representing customization. Thereafter they had to answer some statements about the example and at the end they were asked if they could find any resemblances between the given example, and systems they work with or know about at the company. The procedure was the same for all four examples representing customization, composition, expansion and extension respectively.

After reading the examples and spontaneously expressing their views on the characteristics of the categories, the respondents had to take a standpoint on eleven proposed attributes that originate from the cooperation with the telecom company. The attributes have emerged through participant observations, discussions and interviews.

The interviews made it clear that changes may be required because of changes in the business environment, because of a need for improved usability or because of internal issues in the system itself. The attributes can be divided into corresponding groups. One group concerned the category's suitability for different types of *business changes.* Another group of attributes related to *usability* and a third group involved *software attributes*. The attributes are listed below.

Business changes

Attribute 1:	Frequency of change – how often the business changes occur, frequently or infrequently.
Attribute 2:	Anticipation of change – to what extent it is possible to anticipate the business changes.
Attribute 3:	Durability of change – how long the business changes last.
Attribute 4:	System support for change – how well the software supports business changes
Attribute 5:	Consequences if handled wrongly – how extensive the consequences would be for the company if the changes are handled wrongly.

Usability issues

Attribute 6:	Simplicity – how easy it is to realize the changes in the software
Attribute 7:	User control – how much control the users have of what happens in the software
Attribute 8:	Accountability – how easy it is for the users to know if the result is correct.
Attribute 9:	Realization speed – how fast it is to realize the changes in the software.

Software attributes

Attribute 10:	Fault tolerance– to which degree the software prevents mistakes.
Attribute 11:	Complexity– how complex the software is

3.2 Analysis

The analysis has been performed in a systematic way, according to a specific, predefined schema. The material from the interviews consists of spontaneously stated attributes, predefined attributes, comments, and feedback from respondents. The four components have been considered in the analysis.

The analysis of the interviews consists of two parts that respectively correspond to the two research questions.

Analysis 1: Analysis to determine what attributes characterizes end-user tailorable software.

Analysis 2: Analysis to determine how the respondents perceive the different types of tailoring and decide a value for each attribute, to be able to distinguish different dimensions of tailoring.

Analysis 1. The first step in Analysis 1 is to compare each attribute to see if they are perceived in the same way for all four categories. If they are the same for all the categories then they do not add any information that could be used to distinguish between the categories. All attributes are compared and if they are not the same for all categories they are added to the pile of remaining attributes. If the attribute is the same for all four categories the respondents' comments are consulted to determine if the attributes really were perceived as the same. Perhaps the respondents had made a statement based on different interpretations of the proposed attributes. If the attributes are found to be the same they are removed, otherwise they are added to the pile of remaining attributes. To facilitate determination of whether the attributes were perceived as the same, all statements were assigned a value. A statement interpreted as positive towards an attribute generated a score of 300 and a negative statement was assigned 100 points. Accordingly a neutral statement generated 200 points. Initially, to see if the attributes were the same for all categories, the value of the attribute was summarized. For example if all the users think that Example 1 has high fault tolerance the sum is 1200 points (4 users x 300 points) and if all the users think that Example 4 has low fault tolerance it generated a total of 400 points (4 users x 100 points). The sums for each category are compared and if they are the same they have to be examined further and each comment must be checked more closely.

The second step in Analysis 1 is an examination of how a respondent's answers relate to the other answers in the group. The coefficient of variance has also been used as a measure of the disagreements between respondents [16]. If the respondents' view of the attributes of the examples varied a lot, then the attributes should be removed, as this does not reveal anything about the category. The remaining attributes from step A were examined. If there is a deviation in opinions within the group the respondents' comments were checked. Based on the comments, the relevance of the attributes was questioned. If the attribute was found relevant it was added to the pile of remaining statements otherwise it was removed.

In step three of Analysis 1, the respondents' spontaneously assigned attributes were listed and compared with the pre-defined attributes. If they were the same the attributes were added to the comments, otherwise they were considered as attributes of the intended category.

Analysis 2. The remaining attributes from Analysis 1 were analysed to explore how the user group relates to the developers group, per attribute. The median value for each attribute was used for guidance. If the users and developers agree upon the attributes the attributes were collected into one pile, but if opinions differ, the respondents' comments are considered and the user specific-and developer-specific statements are accumulated into separate piles.

4 Result

When examining the totals in the first step of Analysis 1 there were some attributes that had the same total, but when the individual scores and the comments were inspected it was revealed that they actually differed. The result of the analysis is that none of the attributes was perceived as the same for all four categories and therefore none of the attributes could be excluded at this stage.

The second step in Analysis 1 resulted in the removal of three attributes (3, 5 and 6), as there were strong disagreement among the respondents. The attributes concerned durability of changes, consequences if handled wrongly and simplicity. The respondents regarded durability of change and simplicity as somewhat unimportant, and their answers were therefore fairly random. The consequences of the change being handled wrongly were too difficult to state as it is highly dependent on the situation.

The users found it difficult to spontaneously come up with attributes describing the four examples. They experienced difficulties in moving from the concrete example to a more abstract level. They found it to be easier to associate the example with a system they work with. It was much easier for developers to come up with attributes for the four examples and the developers came up with a couple of attributes each.

When comparing the developers' attributes with the pre-defined attitudes it was found that most of the attributes were the same. The attributes that differed from the pre-defined ones related to usability issues and were mentioned by several of the developers. The attributes were of two kinds and concerned:

Frequency of use: how often the end users use the software and thereby the degree of familiarity the users have with the software, and
User competence: how skilled the users of the software are.

Analysis resulted thereby in ten relevant attributes that can be used to describe end-user tailorable software (see Table 2).

The results of Analysis 2 showed that users and developers had the same perception of Example 1 (customization).

For Example 2 (composition) the users and developers had slightly a different perception of user control, accountability, fault tolerance and complexity. When it comes to user control and accountability the users judge the accountability and control to be medium high, while the developers think it is somewhat higher. In other words, the developers thought that Example 2 contains slightly more accountability and user control that the users did. For fault tolerance and complexity there were also some small differences. The users considered the fault tolerance and complexity to be medium high for Example 2, whilst the developers thought that fault tolerance is somewhere between medium high and low and the complexity between medium high and low. (See Table 2)

Also for Example 3 (expansion) there were some differences in views. One thing is that the developers had a unanimous view that Example 3 is well suited when there is a need for high support for changes, but the users are not that sure. They believe that such software provides quite a lot of flexibility, but they are not certain that Example 3 really supports change so well that it should be regarded as giving "high support of change". A small variation also exists in the judgment of the amount of user control and accountability provided by Example 3. The developers consider Example 3 to provide medium high user control and accountability while the users believe it to be somewhere between medium high and high. The differences of opinion in this case were however very small. A more significant difference was found when it came to anticipation of change. Here the users and developers had diametrically opposite opinions. The users thought that Example 3 was suitable for situations characterized by a high degree of anticipated changes. The developers thought to a higher degree that Example 3 was also well suited for unanticipated changes (See Table 2).

The issue of user control and accountability for Example 4 (extension) resulted in some discussions of which knowledge is build into the system and what should be controlled by the user. Both users and developers agreed that it is possible to view Example 4 as supporting either high control and accountability or low control and accountability. There is very little user control and accountability built into Example 4,but on the other hand the user handling the software should be skilled and know what he or she is doing. Thereby you could say that the software gives control to the users. User control and accountability should therefore be regarded as high. The uncertainty is represented by question marks in Table 2.

Table 2. Matrix of the attribute values of the four categories of end-user tailoring. (L=Low, M=Medium, H=High, ?= Uncertainty of how to use the attribute).

Characteristics		Customization	Composition	Expansion	Extension
Business Changes	Frequency of change	M	M	H	H
	Anticipation of change	H	M	L-H[1]	L
	System support of change	L	M	M-H	H
Usability Issues	User control	H	M-H	M-H	?
	Accountability	H	M-H	M-H	?
	Realization speed	H	H	M	M
	Frequency of use	L	H	-[2]	-
	User competence	-[3]	-	M-H	H
Software Attributes	Fault tolerance	H	M-H	M	L
	Complexity	L	L- M	M	H

[1] Users thought the example was highly suitable for anticipated changes, developers thought the example was not so suitable for such situations.

[2] The spontaneously given attributes were not stated for Example 3 and 4.

[3] The spontaneously given attributes were not stated for Example 1 and 2.

Note that there are two pairs of attributes that show a dependency (Table 2). User control and accountability have corresponding values for all categories. When user control is perceived as high, accountability also has a high value. Fault tolerance and complexity also seem to be related. If fault tolerance is high then complexity is low and vice versa.

When it came to the spontaneously stated attributes, example 1 was considered suitable when there are many end users that use the software only occasionally and Example 2 was regarded as fitting when there are few end users who use the software frequently. Examples 3 and 4 were believed to be appropriate when the end users are skilled and used to computer work, but Example 4 was judged to be appropriate only for a few users that are extremely skilled super users.

The matrix can be used when the tailoring capabilities are insufficient and a new feature is needed, and a development team is put together with both users and developers. In such situations, the matrix can act as a gateway to the categorisation of end-user tailoring and point to a type of tailoring that may be appropriate for the specific feature. The matrix is intended as a basis for discussion between users and developers and the matrix has to be accompanied by complementary tools that relate the different categories of tailoring to implementation techniques to be able to make a decision of how to implement the feature.

The matrix should be seen as a guiding tool. The matrix should not be seen as providing the absolute truth. When designing a tailorable system the matrix could be used as a basis for discussions of the needs and requirements of the specific situation. What the matrix can do is to help the participants focus on a subset of tailoring possibilities and make it easier to choose the right type of tailoring for a specific situation. What can be expected from different types of tailorable software is listed in the matrix, but it is the participants in the project that have to make the tradeoffs between the attributes.

5 Discussion

The matrix is intended for a design environment where the users are informed participants, and where users and developers claim a common ownership of the software product developed. The purpose of the matrix is to act as a basis for design discussions where the users and developers discuss the requirements of the tailorable software to understand better the domain and design problems. The matrix can help the design team to pinpoint issues to discuss and to reach a consensus to enable decisions concerning the dimensions of tailoring needed in the given context. By consulting the matrix and comparing the values of the attributes with what is needed in a specific context, it is possible to get an indication of the kind of tailoring to implement and to be able to make informed design decisions.

There is a similarity between assigning quality attributes to software and assigning attributes to tailoring categories. Both aim to describe a phenomenon by assigning characteristics to it. There are several software quality models, for example [1, 12, 13], and their common effort is to manage quality issues in software development. There is a resemblance between these quality models and the software attributes extracted from our study. Some of the attributes in the matrix can also be found in some quality models. The intention of the matrix is not however to give a general overview of different quality attributes. The matrix aims to distinguish between different types of tailoring and to support design decisions when designing tailorable software. However, there are some similarities. McCall's model, for example, is an effort to bridge the gap between the users' view and the developers' view [13]. The matrix also aims to bridge the gap between users and developers by providing a means of communication, and although we do not claim it to be as complete as McCall's model, the study gives us a good indication of which characteristics can be assigned to the different types of tailoring.

Bosch [2] advocates assessment of the quality attributes during architectural design. The attributes are used for evaluating the architecture to determine if the architecture has to be transformed or not. The attributes in the matrix are not used for evaluation. The intended use of the matrix could be said to be a bottom up approach in comparison with Bosch's method. The four categories could be seen as a kind of "design pattern light" for tailorable software. Instead of imposing a design pattern after the architecture has failed to provide for the required quality attributes, the matrix starts out from the categories that have assigned attributes and trade-offs are made. The architecture is then built based on the selected category. Another difference between Bosch's approach and ours is that Bosch presumes that it is possible to assign an exact, measurable value to the quality attribute, but we only assume that the participants can grade the attributes from low to high.

6 Conclusion

The study made visible ten attributes of end-user tailoring. In interviews with users and developers at a telecom company the respondents were asked to give their opinions of what characterizes four categories of end-user tailoring. Their perceptions of the categories were analysed and it was possible to process their views into a matrix representing four types of tailoring in the form of attribute values. The attributes represent organizational, business and technical issues to consider and can be used in a dialectic process to balance the human-centeredness and the technical solution, as Gasson requires [9].

The matrix can be used as guidance and a basis for design decisions when implementing end-user tailorable software. The attributes are at a level that can be understood by both users and developers and as shown, even though differences exist, the opinions of users and developers are quite similar. The matrix makes it possible to distinguish between different dimensions or types of tailoring, by providing values for the attributes that characterize end-user tailorable software.

The categories and attributes of the categories, together with the matrix and examples, facilitate the understanding of different types of tailoring and should make it easier for developers and users to discuss tailorability and the requirements associated with tailorable systems.

Acknowledgment. This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.bth.se/besq).

References

1. Boehm, B.W., Brown, J.R., Lipov, M.: Quantitative Evaluation of Software Qualities, North Holland. In: Proceedings of the 2nd International Conference on Software Engineering, ICSE 1976, California, USA (1976)
2. Bosch, J.: Design and use of Software Architectures: Adopting and evolving a product line approach. Pearson Education. Addison-Wesley and ACM Press, Reading (2000)
3. Brown, J.S., Duguid, P.: The Social Life of Information. Harward Business School Press, Boston (2000)
4. Brown, J.S., Duguid, P., Haviland, S.: Toward Informed Participation: Six Scenarios in Search of Democracy in the Information Age. The Aspen Institute Quarterly 6(4), 49–73 (1994)
5. Eriksson, J., Dittrich, Y.: Combining Tailoring and Evolutionary Software Development for Rapidly Changing Business Systems. Journal of Organizational and End User Computing (JOEUC) 19(2) (2007)
6. Eriksson, J., Lindeberg, O., Dittrich, Y.: Four Categories of Tailoring as a Means of Communication. Journal of Software and Systems (submitted, 2007)
7. Fischer, G.: Meta-Design: Beyond User-Centered and Participatory Design. In: Proceedings of HCI International 2003, Crete, Greece, June 2003, pp. 88–92. Lawrence Erlbaum Associates, Mahwah (2003)
8. Fisher, G., Ostwald, J.: Seeding, Evolutionary Growth, and Reseeding: enriching Participatory Design with Informed Participation. In: Proceedings of the Participatory Design Conference (PDC 2002), Malmö University, Sweden, pp. 135–143 (2002)
9. Gasson, S.: Human-centered vs. user-centered approaches to information system design. JITTA: Journal of Information Technology Theory and Application 5(2), 29–46 (2003)
10. Henderson, A., Kyng, M.: There's No Place Like Home: Continuing Design in Use. In: GreenBaum, J., Kyng, M. (eds.) Design at Work, 1st edn., pp. 219–240. Lawrence Erlbaum, Hillsdale (1991)
11. Ilvari, J., Iivari, N.: Varieties of User-Centeredness. In: Proceedings of the 39th Annual Hawaii International Conference on System Sciences, HICSS 2006. IEEE, Hawaii (2006)
12. ISO: ISO/IEC 9126 Information Technology - Software Quality, International Standard Organization
13. McCall, J.A., Richards, P.K., Walters, G.F.: Factors in Software Quality Nat'l Tech Information Service, 1, 2 and 3 (1977)
14. Nardi, B.A.: A Small Matter of Programming - Perspectives on End User Computing. MIT Press, Cambridge (1993)
15. Preece, J., Sharp, H., Rogers, Y.: Interaction Design - beyond human-computer interaction. John Wiley & Sons, Inc., New York (2002)
16. Regnell, B., Höst, M., Natt och Dag, J., Beremark, P., Hjelm, T.: Visualization of Agreement and Satisfaction in Distributed Prioritization of Market Requirements. In: 6th International Workshop on Requirements Engineering: Foundation for Software Quality, Stockholm, Sweden (2000)
17. Robson, C.: Real World Research, 2nd edn. Blackwell Publishers Ltd., Oxford (2002)
18. Schuler, D., Namioka, A.: Participatory Design: Principles and Practices. Lawrence Erlbaum Associates, Hillsdale (1993)
19. Stevens, G., Quaisser, G., Klann, M.: Breaking It Up: An Industrial Case Study of Component-Based Tailorable Software Design. In: Lieberman, H., Paternò, F., Wulf, V. (eds.) End-User Development, vol. 9, p. 492. Springer, Dordrecht (2006)
20. Stiemerling, O.: Component-Based Tailorability, Dissertation. Bonn University, Bonn (2000)

Manifoldness of Variability Modeling — Considering the Potential for Further Integration

Mark-Oliver Reiser[1,2], Ramin Tavakoli Kolagari[2], and Matthias Weber[3]

[1] DaimlerChrysler AG, Research & Technology, GR/ESM,
Alt-Moabit 96a, D-10559 Berlin
moreiser@cs.tu-berlin.de

[2] Technische Universität Berlin, Fakultät IV, Lehrstuhl Softwaretechnik,
Franklinstraße 28/29, D-10587 Berlin, Germany
tavakoli@cs.tu-berlin.de

[3] Carmeq GmbH, Carnotstrae 6, D-10587 Berlin, Germany
matthias.weber@carmeq.com

Abstract. Variability management has become an important concern in software and systems engineering. Especially in industrial settings a rigid management of variability has been identified as an important prerequisite for further optimization of the development process, e.g. for reuse of software sub-systems across vehicle models such as the Mercedes Benz A-Class and C-Class. In response to this growing practical interest, the scientific community has come up with numerous concepts and techniques for modeling variability. However, despite initial attempts to integrate or unify some of these manifold approaches, a clear understanding of how they precisely relate to each other is still not yet achieved.

In the paper, various techniques for variability modeling are elaborated and a basic classification scheme is proposed. From this we derive their common capabilities, which arguably embody the essence of variability modeling in general. On this basis, a discussion is presented that concerns the potential and feasibility of integrating all these diverse techniques into a single, common technique for variability modeling.

Keywords: Software product lines, Variability management.

1 Introduction

Over the past decade product line engineering became a popular approach to software development both in classical software engineering domains as well as for industrial software-intensive systems. A *software product line* is a set of software products that share a certain degree of commonality while still showing substantial differences and that are developed from a common set of core assets in a prescribed way [1]. In other words, whenever a company has several similar software products on offer it makes sense to consider developing only a single, but variable product instead of developing the products in parallel and independently from one another, thus shifting the focus of development from the individual products to the product line. Key to all product line engineering is *variability management*, i.e. the documentation and management of the commonality and variability between the products within the scope of the product line.

According to the paradigm of orthogonal variability modeling [2], the variability between the products in a product line is documented and managed as a separate, orthogonal aspect of development, called *variability dimension*. This dimension is thus clearly set apart from the *artifact dimension*, i.e. the definition of the development artifacts, such as requirements, component diagrams, state charts and test cases.

However, variability is often not only addressed in the variability dimension alone. Instead, it is common to describe the variability's precise impact on the development artifacts within these artifacts themselves, i.e. it is explicitly defined at what location in an artifact certain variability shows up and what alternative forms the artifact can take at that location. The fact that in these cases some aspects of variability are also defined in the artifact dimension need not necessarily be seen as a violation of the orthogonal variability modeling paradigm (even though it is sometimes seen as such), because the definition of variability aspects in the artifact dimension only relates to where and how the artifact is affected by variability. The primary focus of variability management—i.e. the presentation of an overview of the entire product line's variability, definition of dependencies between variations and the global coordination of variability across several artifacts—is still, mainly, the variability dimension.

Over the past decade, a multitude of different techniques have been proposed for both the variability dimension (esp. feature modeling [3,4,5,6,7]; decision tables [8,9]; decision diagrams/trees [10,11]) and for defining variability in the artifact dimension (esp. approaches for explicitly defining variation points and their variants in various types of artifacts). Most of these techniques come in a variety of flavors; an attempt to unify some of them has already been undertaken or is currently in progress, e.g. for feature modeling [12,13]. When considering all these methods, a few basic groups of techniques and thus a few fundamental approaches towards variability modeling can be identified, for example feature modeling and decision tables. Unfortunately, how these main approaches relate to each other is not examined in detail and is not well understood. Are they merely different forms of presenting the same information or are there fundamental differences in how they address variability modeling? Since all these techniques are aimed at variability modeling, this situation is unsatisfactory from a theoretical and conceptual point of view: when proposing different ways to treat variability, it should be clear how they differ and why the distinction is necessary. Moreover, there is also a practical problem with this splitting up of basic approaches: When two or more independent product lines need to be related to each other or integrated into a single higher-level product line, different variability modeling approaches are usually applied in the individual product lines. In this case, it must be clear how these approaches relate to each other. Such product line integration is of particular importance in industrial settings; for example, in the automotive industry car manufacturers usually need to integrate the products, i.e. sub-systems, from numerous suppliers' product lines.

In the remainder of this article, we discuss what basic groups of variability modeling techniques can be identified and how they relate to each other. This is done for the variability dimension in Section 2 and for variability in the artifact dimension in Section 3. Then, in Section 4, we discuss the potential and benefit of a further integration of these techniques.

2 Variability Dimension

Roughly, three forms of variability modeling in the variability dimension can be distinguished: *decision tables*, *decision trees/graphs* and *feature models*. Figure 1 shows an excerpt from a decision table, inspired by an example in [8]. A decision table usually refers to one or more variable development artifacts, in the example a use case diagram for the scenario 'send message' (not shown). Each line in a decision table represents a decision to be taken in order to configure the corresponding variable artifact(s) of the table. Each such decision has a name as its unique identifier, a question that formulates the decision to be taken, a list of possible resolutions, i.e. possible answers to the question, and one effect or action per resolution that describes how the corresponding variable artifacts have to be changed in order to configure them in line with the decision taken. Constraints allow defining interdependencies between decisions in order to restrict the available resolutions depending on decisions taken earlier or to hide decisions when they are no longer valid because of some other decision taken earlier. For example, if the decision 'Does the phone have a camera?' was answered with 'no', the decision 'What is the camera's resolution ?' is no longer valid and can be hidden during configuration. The number and precise meaning of each column in a decision table varies from one approach to another, but the example given here illustrates the basic idea of decision tables.

Similarly, decision trees also define decisions to be taken in order to configure one or more variable artifacts. However, the decisions are represented and arranged graphically. Figure 2 shows a small example of such a decision tree. The advantage here is that some selected dependencies between the decisions can easily be defined in this way. For example, the fact that the decision 'What is the camera's resolution?' is invalid if the camera is previously deselected altogether is

Name	Description	Constraints	Resolution	Effect
T9	Does the phone have T9 support ?		yes no	step 6 of use case 'send message' is obligatory; extension 6a of use case 'send message' is obligatory step 6 of use case 'send message' is removed
Cam	Does the phone have a camera ?		yes no	... ...
CamRes	What is the camera's resolution ?	Cam==yes	1MegPix 2MegPix	–
	...			

Fig. 1. Excerpt from a sample decision table (cf. [8])

clearly visible in the tree. Also, the number of possible product configurations is easily ascertainable, because each leaf in the decision tree corresponds to exactly one product configuration. However, this also points at an important problem with decision trees. They tend to become extremely large in complex cases. This can be avoided by using directed acyclic graphs instead of trees. Decision tree approaches (e.g. [14]) differ from one another in many details, but these are not required for the following discussion.

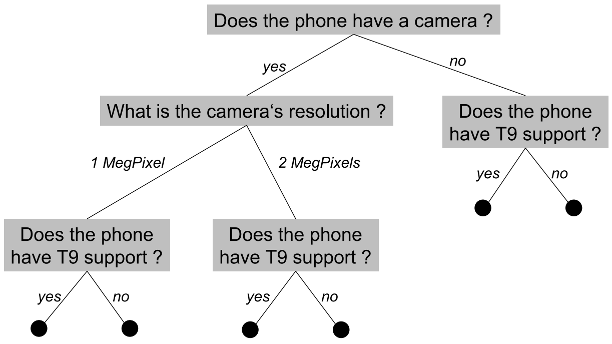

Does the phone have a camera ?
yes
no
What is the camera‘s resolution ?
Does the phone have T9 support ?
1 MegPixel
2 MegPixels
Does the phone have T9 support ?
Does the phone have T9 support ?
yes
no
yes
no
yes
no

Fig. 2. Example of a decision tree

Feature models are the third form of variability modeling in the variability dimension. A feature is a characteristic or trait that an individual product instance of a product line may or may not have [15]. The purpose of a feature model is to provide an overview of both the common and variable characteristics of the product instances and the dependencies between them. Figure 3 shows an example. Each node in the tree depicts a feature (e.g. `CruiseControl`, `Wiper`). During configuration, features are selected or deselected. Child features may only be selected if their parent is. Each child feature has a cardinality stating whether it is mandatory, i.e. it needs to be selected if the parent is, optional, i.e. it may or may not be selected if the parent is, or if it can be selected more than once (so called cloned features ; e.g. `Wiper`). When a feature is selected more than once, all its descendants can be configured separately each time the feature is selected. For example, if two wipers are selected during configuration of the feature model presented in Figure 3, then the `RainSensor` can be configured independently for each of the two. In addition, several children of a single feature can be grouped to express a certain dependency between them, e.g. the alternativity between `Simple` and `Adaptive` in Figure 3. More general dependencies between features of different subtrees can be expressed through feature links which usually are depicted as an arrow (e.g. between `RainSensor` and `Radar`). Furthermore, features may be parameterized meaning that if the feature is selected during configuration, a value of a certain type has to be provided, for example when `Radar` is selected, the minimum distance to the next car has to be supplied as an integer value (cf. Figure 3).

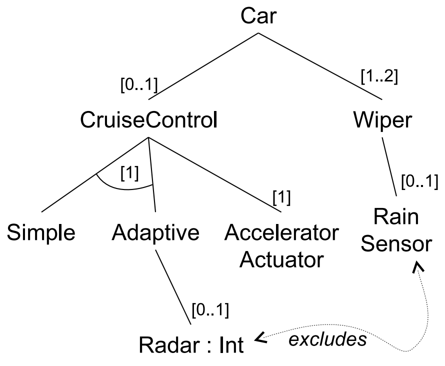


Fig. 3. Example of a feature model with advanced concepts

Again, the details of feature modeling approaches (for an overview refer to [12]) differ greatly; but for the discussion presented here, the basic idea of feature modeling suffices.

Since all three forms of variability dimension modeling basically have the same purpose—presenting an overview of the product line's variability and providing a basis for configuration—it makes sense to ask whether they are basically equivalent and are merely different ways of presentation for the same information. In order to tackle this question more systematically, we examine whether the different forms of variability modeling can be translated into one another without loss of information.

Translating a decision table into a feature model is quite straightforward. For yes/no decisions, a simple feature is created; for value decisions, a parameterized feature is added; and for decisions with a finite set of enumerated resolutions, a parent feature is created together with a child for each of the allowed resolutions. Decisions' constraints are turned into feature links. The problems with this translation are:

(1) The natural-language description expressing the decision to be taken cannot be expressed in the feature model. The features' textual descriptions are not normally formulated in such a way. However, the description of a feature could still be used for this purpose, or an additional attribute could easily be introduced, if desired.

(2) Decision constraints can refer to several other decisions in a complex way. Since feature links are often defined as links from one single source feature to a single destination feature, this technique is less expressive. Again, this is not a fundamental problem for the translation because a more flexible feature linking concept could be provided.

(3) In addition, the feature model created from a decision tree in this way will be very flat. Since we only require that all the information from the decision table can be expressed and is therefore present in the feature model, this is not really an obstacle to such translation. However, it already points to an important problem that we will encounter below when examining translation in the opposite direction.

Despite these limitations, the translation from a decision table to a feature model works relatively well. Unfortunately, this is not true for the opposite direction. Basically, we can create a decision for each feature that is not mandatory as follows: for simple features a yes/no decision is created, and for parameterized features a value decision is provided; alternative features are merged into one decision with one resolution per feature. Parent-child relations are mimicked with decision constraints. While this mapping works well in principle, we identified several critical mismatches and problems during our investigation:

(4) Feature links can easily be formulated as decision constraints. However, in that case the dependency needs to be added to either the source or target decision, while a feature link represents a dedicated entity between the two. Also, one feature link can easily be kept apart from other feature links affecting the same feature and from dependencies that are expressed as parent-child relations, feature groups, etc. In decision constraints, all these dependencies get mingled within a single constraint.

(5) *The hierarchical structuring defined through the parent-child relationships gets lost.* Although the dependency expressed in a parent-child relation (i.e. the child may only be selected if the parent is selected) can be preserved in the corresponding decision constraint, it is not possible to document the fact that this dependency came from a parent-child relation. In other words, when looking at the decision table, it is no longer possible to distinguish between the dependencies that are to be interpreted as parent-child relations or hierarchy and those that are to be interpreted as feature links. This problem could be solved by introducing hierarchy in decision tables. However, it would then no longer be possible to edit them with standard office applications, which is one of the most important advantages of decision tables.

(6) Typed edges, i.e. types of parent-child relations, cannot be expressed in a decision table.

(7) *Cloned features cannot be translated into standard decision tables.* Of course, a similar concept could be incorporated in decision tables—i.e. several lines of the table would be replicated during configuration and then configured separately for each copy—but such a mechanism is not available in any of the common decision table approaches.

(8) *Mandatory features cannot be translated into decision tables.* This results in the most important difference between the forms of variability dimension modeling: in contrast to decision tables and decision trees/graphs, feature modeling does not primarily focus on the decisions to be taken during configuration and the resulting effects on the variable artifacts. Instead, feature modeling focuses directly on the differences and similarities between the product line's individual products. More specifically, feature models list all important characteristics of the individual products and state whether these characteristics are common to all products or vary from one product to another.

To this extent, decision diagrams/trees are very similar to decision tables. There is only one additional difference that arises when comparing them to the other two forms of variability dimension modeling:

(9) Decision diagrams/trees bring all configuration decisions into a certain order. For example, if feature f1 and f2 exclude each other (defined by a feature link), then neither has priority over the other. By contrast, in a decision tree with decisions d1 and d2—corresponding to f1 and f2, respectively—either d1 is asked before d2 (and consequently d2 won't be asked at all if d1 is answered with yes) or d2 before d1 (and d1 is therefore skipped in the case of a positive answer to d2). In the first case, d1 has "priority" and in the second d2. Even though this does not make a difference on a technical level, it is of great importance from a methodical point of view.

In summary, we can say that there are fundamental differences between the three forms of variability dimension modeling.

3 Artifact Dimension Variability

Successful management of variability also includes handling artifact variability, i.e. variability of software development assets on different realization levels. Artifact dimension variability can be thought of as being described in different ways:

– *Internal*: artifact variability is explicitly expressed within the artifact meaning that the possible design decisions and alternatives are explicitly given in the artifact descriptions.
– *External*: artifact variability is described outside the artifact meaning that the possible design decisions are captured elsewhere than within the artifact specification. Here, often the variability specification that forms the variability dimension is used as the location where the artifact variability is specified, i.e. the variability dimension is augmented by information on the variability of the artifact dimension.

Also the configuration of artifacts (process of binding variability) can be managed differently:

– *Generative/constructive artifact configuration*: artifact configuration is realized by generating the final, configured artifact and/or by composing it out of basic elements from an overall pool of (variable) artifact elements.
– *Alterative artifact configuration*: artifact configuration is realized by changing, i.e. enhancing or reducing, a default artifact model.

Bearing these possible differences in mind, one can derive basic artifact dimension variability approaches and essential concepts for the modeling of artifact dimension variability as well as the configuration of variable artifacts. In this section we provide an exemplary overview of artifact dimension variability by introducing a category matrix of artifact dimension variability management approaches, shown in Table 1.

Table 1 gives an overview of basic groups of current approaches for artifact dimension variability definition. The idea is not to have a complete overview of existing approaches but to motivate the differences between these four essential concepts represented as the four fields in the matrix.

Table 1. Category matrix of artifact dimension variability management approaches

	Internal	External
Generative / constructive	—	e.g. decision models, feature models
Alterative	e.g. explicit variation points and variants	e.g. aspect-oriented model transformation

External and Generative Variability Management Approaches
The field at the top right of the matrix represents approaches that model variability externally to the artifact and that obtain an artifact configuration by way of generation or through a composition of individual artifact elements or fractions of artifacts (e.g. [16] and [17]). As described above, artifact dimension variability is in this case incorporated into the variability dimension specification, where decision models for a set of assets as gained from domain engineering—as presented in PuLSE-CDA [8]—are an example of such an approach (see also the previous Section 2). Variability and dependency relations are only described in the decision model—not in the artifact elements. The anchor in the artifact elements for the decisions is described as part of the decision rule, e.g. by using a unique identifier for artifact elements. Decisions only refer to the variability or dependency relations at different levels of detail: thus one can either select or deselect a specific element or one may set specific values for parameters (see Figure 1). Invariable elements are thus selected automatically from the element pool, and transitive or technical relations are also applied automatically. An external variability modeling approach for a pool of artifact elements is easily applicable and does not need to be tool-supported in the first place. Complex variability and dependency relations can be modeled and the expressiveness of decision models is high because the complete arrangement of the artifact elements can be described. Besides decision models as an external variability modeling approach for an elementary artifact pool also feature models can be thought of to be an applicable approach. The problem here is that feature models used for the configuration of artifact elements need explicit links to the artifact elements. Furthermore, it must be described at the artifact elements which constellation of features leads to which configuration of the artifact elements.

Internal and Alterative Variability Management Approaches
The field at the bottom left of Table 1 contains such approaches that model variability explicitly within the artifact elements of any kind of "default" model, which is changed in the course of the instantiation process, i.e. either the default model is enhancing because new elements are added, or it is reducing because variable elements are dropped from the model. An example of an approach describing variability internally for an enhancing default model is the explicit description of variation points, variants and dependency relationships between them. The default model then comprises all the characteristics (artifact assets) of the whole product family. The asset abundance is constantly reduced through variability binding. Describing variability within a variable default model calls

for explicit scoping efforts in an early domain engineering phase because all products with their differences and commonalities are derived from the default model. Furthermore, it is not easy to obtain an overview of the artifact variability from a conceptual point of view because often a single conceptual variability (e.g. the wiper has a rain sensor) affects an artifact at many different locations and this variability's definition is therefore split across many variation points and, similarly, at a single location many different conceptual variabilities can have an impact and are thus mingled into a single variation point. Thus if the differences and commonalities are clearly defined and the artifact variability is complex in the sense that it is local or only technically based, then an artifact-internal variability description is reasonable. In order to obtain an overview of the variability and to facilitate the instantiation process the internal variability description will in this case often be complemented by an external variability modeling approach in the variability dimension, e.g. feature models.

External and Alterative Variability Management Approaches
The bottom right field in the matrix corresponds to external variability modeling approaches that change a default model. Once again—as described in the previous paragraph—the default model can be reduced or enhanced during the variability instantiation process. An example of such an approach is aspect-oriented model transformation. Differences between products are captured in aspects that are described in the form of a transformation rule. The rule consists of a point-cut and an advice, the point-cut specifying the place in the original model (join-point) that has to be amended with the model fragment described in the advice of the rule. This kind of variability modeling approach can handle complex variability and cope with many shortcomings of the internal approach as described in the previous paragraph (esp. splitting of a single conceptual variability across many variation points). However, a difficulty with this approach is that things become extremely complex when several transformations affect the same location in an artifact. Basically, external variability modeling changing a default model can be used for all kinds of artifact variability, but its actual power lies in its great expressiveness: in contrast to the internal variability modeling approach, which is bound to the principal design of the default model, an external variability modeling approach can change entire parts of the design and rearrange or exchange completely independent design fragments. Thus especially in cases where variability results in a rearrangement of the design, an external approach would be worth considering.

Internal and Generative Variability Management Approaches
Remarkably, so far no approaches have been published in the literature describing variability in the artifacts with a generative/constructive artifact configuration (field in the top left of the matrix). This may be due to the fact that a variability description in artifact elements is usually limited to capturing simple variability and dependency relations in order to remain straightforward and manageable. As mentioned above, complex variability and dependency modeling calls for an orthogonal view of the variable artifact elements because coherences with

respect to variability and dependency of variable artifact elements cannot merely be described at one element but are rather crosscutting. Here, a possible artifact dimension variability approach would assume a domain with simple and local variabilities and only technical dependencies (e.g. communication relationships). In this case, the variability and dependency relationships can be described locally at the artifact elements, and a configuration can be derived only via the element-based variability and dependency description. Although not an artifact dimension variability management approach, a Java development library can be thought of as the simplest approach to locally describe dependency relations and where the configuration is constructive, though not tool-supported.

Discussion
The remainder of this section discusses the results of table 1:

(1) Currently no internal and generative variability management approaches exist. Potential use could be in a case of simple, local occurrence of variability. Such a technique would have to be simple and may influence other variability management approaches to become easier applicable in various practical situations.

(2) External variability management approaches augment the variability dimension with additional, artifact related information and therefore—at least implicitly—establish a link between the artifact and variability dimensions. Use of feature models as well as decision tables proved successful, especially in the case of complex and highly interrelated variability. Use of feature models and an explicit description of the artifact configuration based on a feature selection ends up in a mixture of internal and external description of artifact variability. Thus, it can be said that there is a continuum rather than a two-valued scale from internal to external variability modeling approaches.

(3) The most flexible technique to manage artifact variability is an internal and alterative approach. A default model containing variation points is changed in the course of the variation point instantiation. This technique is broadly used in programming (e.g. abstract parameters, types) and therefore often the first choice to manage variability of artifacts. In complex cases however, the spreading of variability information throughout the system models my prove infeasible. In this case a mixture between both an internal and external, generative approach would be needed.

(4) A more complex but powerful way to change a default model is the use of model transformation rules. With the help of these rules complete parts of the model can be rearranged and changes in different artifacts at various places can be defined together in a single rule. But because of its complexity model transformation should only be used in cases where this flexibility is needed. In all other cases an internal way to change the default model should be chosen.

It can be observed that most of the approaches applied in practice are mixtures between internal and external, generative and alterative variability management techniques; this is beneficial because the decoupled approaches complement one another.

4 Potential for Further Integration

Based on this survey of fundamental approaches to variability modeling and their interrelations presented in the previous two sections, we can now come back to our initial question whether this diversity of variability modeling techniques is actually required, or whether it would make sense to aim for replacing them with a single, comprehensive variability modeling technique.

First of all, a single technique for variability modeling both in the variability and in the artifact dimension is not realistic. These two cases of variability modeling are of very different nature actually: In the variability dimension, an overview of variations across many artifacts is to be provided on an abstract, conceptual level while in the artifact dimension, the variations of an individual artifact, i.e. the impact of variability, need to be defined precisely. Thus, the technique for the artifact dimension introduces variability in an existing artifact while the technique for the variability dimension constitutes an additional artifact of its own.

Consequently, the highest degree of integration that is conceivable would be to have a single technique for managing variability in the artifact dimension and another for the variability dimension. In the artifact dimension, however, the fundamental approaches towards variability modeling illustrated in Table 1 differ greatly and are aligned with very diverse methodological requirements (as described in Section 3). Therefore an integration would not make sense at this point. The next lower level of integration would be to provide a single, integrated technique for each of the four cells of Table 1. But also this is not feasible in practice, because of the great diversity of the development artifacts that need to be covered by these techniques. For example a requirements specification may call for very different means to express variability than a test case description; similarly, the concept of aspect-orientation proved feasible for weaving variability into program code but its application to design models is still a challenging research issue. This diversity can be seen as being orthogonal to the two dimensions of Table 1. In order to ensure usability it is necessary to tailor the variability technique to the specific characteristics and needs of the artifact in question. Hence, the artifact dimension does not have great potential for a further integration of techniques.

For the variability dimension, on the other hand, this is much different. The variability dimension represents an artifact of its own and has a global perspective spanning all other development artifacts. It therefore needs to be independent of the artifacts' specificities anyhow. In addition, we identified that the different basic approaches towards variability modeling in the variability dimension all share the same major objective: establishing a global perspective on variations within the product line and defining dependencies between them. Practical considerations also suggest an integration of approaches: While the definition of the precise impact of variability inside an artifact (i.e. the purpose of variability modeling in the artifact dimension) usually is the responsibility of a single team working on the corresponding artifact, the global variability dimension is frequently subject to coordination between teams, departments and companies. Therefore, an integration in this area would be of high practical value.

However, we also identified substantial disparities between the basic approaches for variability dimension modeling, esp. the lack of commonality and hierarchy in decision tables. An integration will therefore be a challenging task and requires significant further research.

5 Summary and Conclusion

We surveyed current techniques for managing variability and, by categorizing them, identified several main approaches to variability modeling and examined how these are related. Based on this, we discussed the potential of integrating them into a single, common technique for variability modeling. We argued that such an integration effort should be targeted at the variability dimension only, both for practical and conceptual reasons.

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading (2002)
2. Bachmann, F., Goedicke, M., do Prado Leite, J.C.S., Nord, R.L., Pohl, K., Ramesh, B., Vilbig, A.: A meta-model for representing variability in product family development. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 66–80. Springer, Heidelberg (2004)
3. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) – feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute (SEI), Carnegie Mellon University (1990)
4. Asikainen, T., Männistö, T., Soininen, T.: A unified conceptual foundadtion for feature modelling. In: 10th International Software Product Line Conference (SPLC 2006), pp. 31–40 (2006)
5. Batory, D.: Feature models, grammars, and propositional formulas. Technical Report TR-05-14, University of Texas at Austin (2005)
6. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practices 10(1), 7–29 (2005)
7. Czarnecki, K., Kim, C.H.P.: Cardinality-based feature modeling and constraints: A progress report. In: Proceedings of the OOPSLA 2005 Workshop on Software Factories (oct 2005)
8. Muthig, D., John, I., Anastasopoulos, M., Forster, T., Dörr, J., Schmid, K.: Gophone – a software product line in the mobile phone domain. IESE-Report 025.04/E, Fraunhofer IESE (2004)
9. Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J.M.: Pulse: a methodology to develop software product lines. In: SSR 1999: Proceedings of the 1999 symposium on Software reusability, pp. 122–131. ACM Press, New York (1999)
10. TreeAge Software: TreeAge Software Inc. DATA Interactive White Paper (1999), `http://www.treeage.com/DIDocs/start/whitePaper.php3`

11. Apprentice Systems Inc.: Apprentice Decision Modeler (2005), `http://www.apprenticesystems.com`
12. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Feature diagrams: A survey and a formal semantics. In: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE 2006), pp. 136–145. IEEE Computer Society, Los Alamitos (2006)
13. Reiser, M.O., Tavakoli Kolagari, R., Weber, M.: Unified feature modeling as a basis for managing complex system families. In: Proceedings of the 1st International Workshop on Variability Modeling of Software-Intensive Systems (VAMOS), University of Limerick, Ireland (2007)
14. Tessier, P., Gérard, S., Terrier, F., Geib, J.-M.: Using variation propagation for model-driven management of a system family. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 222–233. Springer, Heidelberg (2005)
15. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005)
16. Czarnecki, K.: Overview of generative software development. In: Banâtre, J.-P., Fradet, P., Giavitto, J.-L., Michel, O. (eds.) UPP 2004. LNCS, vol. 3566, pp. 313–328. Springer, Heidelberg (2005)
17. Czarnecki, K., Eisenecker, U.: Generative Programming. Addison-Wesley, Reading (2000)

Author Index