ABSTRACT

Title of Document:	PROCESS CONFORMANCE TESTING: A METHODOLOGY TO IDENTIFY AND UNDERSTAND PROCESS VIOLATIONS IN ENACTMENT OF SOFTWARE PROCESSES
	Nico Zazworka, Doctor of Philosophy, 2010
Directed By:	Professor Victor R. Basili
	Department of Computer Science

Today's software development is driven by software processes and practices that when followed increase the chances of building high quality software products. Not following these guidelines results in increased risk that the goal for the software's quality characteristics cannot be reached. Current process analysis approaches are limited in identifying and understanding process deviations and ultimately fail in comprehending why a process does not work in a given environment and what steps of the process have to be changed and tailored.

In this work I will present a methodology for formulating, identifying and investigating process violations in the execution of software processes. The methodology, which can be thought of as "*Process Conformance Testing*", consists of a four step iterative model, compromising templates and tools. A strong focus is set on identifying violations in a cost efficient and unobtrusive manner by utilizing automatically collected data gathered through commonly used software development tools, such as version control systems. To evaluate the usefulness and correctness of the model a series of four studies have been conducted in both classroom and professional environments. A total of eight different software processes have been investigated and tested. The results of the studies show that the steps and iterative character of the methodology are useful for formulating and tailoring violation detection strategies and investigating violations in classroom study environments and professional environments.

All the investigated processes were violated in some way, which emphasizes the importance of conformance measurement. This is especially important when running an empirical study to evaluate the effectiveness of a software process, as the experimenters want to make sure they are evaluating the specified process and not a variation of it.

Violation detection strategies were tailored based upon analysis of the history of violations and feedback from then enactors and mangers yielding greater precision of identification of non-conformities.

The overhead cost of the approach is shown to be feasible with a 3.4% (professional environment) and 12.1% (classroom environment) overhead.

One interesting side result is that process enactors did not always follow the process for good reason, e.g. the process was not tailored for the environment, it was not

specified at the right level of granularity, or was too difficult to follow. Two specific examples in this thesis are XP Pair Switching and Test Driven Development. In XP Pair Switching, the practice was violated because the frequency of switching was too high. The definition of Test Driven Development is simple and clear but requires a fair amount of discipline to follow, especially by novice programmers.

PROCESS CONFORMANCE TESTING: A METHODOLOGY TO IDENTIFY AND UNDERSTAND PROCESS VIOLATIONS IN ENACTMENT OF SOFTWARE PROCESSES

By

Nico Zazworka

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2010

Dedication

I dedicate this thesis to my parents,

Frank and Karin Zazworka,

who made all of this possible.

Acknowledgements

This dissertation would not have been possible without the support and help of many individuals.

I would like to thank my parents and family for guiding my education in my early years and teaching me the importance of education. The studies presented in this thesis were always a team effort could not have been conducted without the combined effort of following researchers:

at the University of Maryland: Victor Basil and Marvin Zelkowitz

at the Fraunhofer Center: Forrest Shull, Christopher Ackermann, Michele Shaw, Daniela Cruzes, and Lucas Layman

at the University of Maryland, Baltimore County: Carolyn Seaman

at the University of Hanover, Germany: Eric Knauss, Kai Stapel, and Kurt Schneider

at the University of Applied Science, Mannheim, Germany: Jan Schumacher, Steffen Olbrich, Selcuk Imal, Frank Breitinger, and Christian Conrad

I am particularly indebted to my advisor, Victor Basili, for his boundless patience, motivation, and for letting me pursue my idea of process conformance testing.

I am grateful to the members of my dissertation committee: Victor Basili, Atif Memon, Amol Deshpande, Marvin Zelkowitz, and William Dorland. Their time and feedback have improved the quality of this work.

I would also like to thank my friends and roommates, Hassan Sayyadi and Anand Veeraragavan, for making “4322 Rowalt Drive” feel like home.

Lastly, I would like to thank Tersha Langdon, for her endless patience during homework, paper and proposal deadlines, for her proof reading of this thesis, and for her love and support throughout the time of my studies in Maryland.

Table of Contents

List of Tables

List of Figures

1 Introduction

Building high quality software products within time and budget remains the most challenging task in software engineering. Reports from the Standish Group (Rubinstein) indicate that *"Software development shops are doing a better job creating software than they were 12 years ago"*, but still only *"35% of software projects in 2006 can be categorized as successful meaning they were completed on time, on budget and met user requirements"*. The "Chaos Reports" are often cited in conjunction with the "Software Crisis" and opinions exist that the study might be biased towards failing projects (Glass). However, project managers and researchers have come to understand that a series of challenges exist when building a software product.

To overcome these challenges and to build better products software developers and researchers have formulated and advanced ideas about how they can support software development in order to ***increase the chances of developing a product of superior quality***. Part of these ideas manifests themselves in **guidelines** or in "a set of instructions" (i.e. **software processes**) that when being followed by developers improve certain quality aspects of the final product.

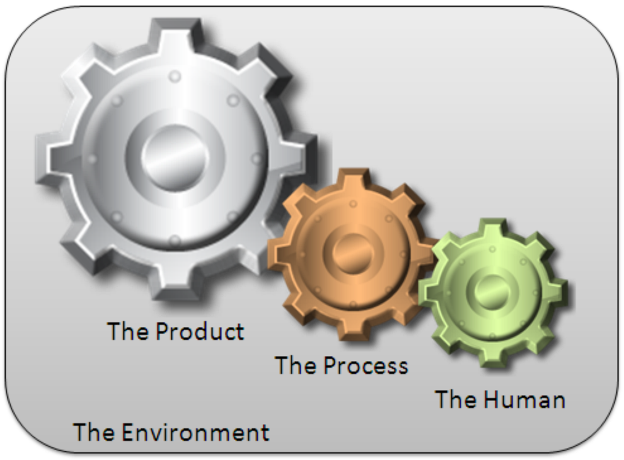


Figure 1: Relationship between products, processes, and humans

Figure 1 illustrates the relationships. For example, developers performing software inspections help to improve the correctness of a product, or programmers following an agile software development lifecycles (e.g. Scrum) promise it to be more flexible to late changes of product requirements. The two examples show that processes can focus on very different dimensions of the product. Further, the activities and steps defined by these processes can differ in the amount of specificity and detail.

One of the dilemmas with processes is that humans perform them. Hence, software development processes inherit a broad range of **human issues** that play a central (if not the most important) role when executing them. For example, when a project manager decides to implement a new process, such as *Pair Programming*, into the development environment she/he has to be aware of potential problems, such as initial resistance from developers to the practice. Various research efforts support this claim. In a qualitative study introducing Pair Programming to a development team, Gittins and Hoppe (Gittins and Hope) report that in the initial survey *"28% of* *developers preferred to work independently, 57% didn't think they could work with everyone, and 57% stated that pair programmers should spend on average 50% of their time alone". Williams and Kessler* (Williams and Kessler) write, "*Most programmers are long conditioned to working alone and often resist the transition to pair programming. Ultimately, however, most make this transition with great success."*

Besides the ability to adapt to new processes each developer possess a very unique set of skills. These skills can reach from technical knowledge (e.g. about different programming languages) to the ability to write error free and understandable code, to the proficiency to perform a complete code review. In a multi-national, multi-institutional study of assessment of programming skills of first-year computer science students conducted by McCracken et.al. (McCracken, Almstrum and Diaz) results showed that students scores were very diverse and the distribution of scores had bi-modal characteristics (even if they were taught the same material). In fact, the authors state that *"We need to keep in mind that different groups of students have different needs and strengths; we must ensure that the results from one group do not obscure our view of the other."*

Lastly, lots of processes are defined as a set of steps, like one would define an algorithm to be run (repeatedly) on a computer. However, humans are not computers and might forget to execute specific steps or intentionally modify the process for their own reasons. The latter scenario can be caused by shifting priorities (e.g. time pressure to finish a project) or by process definitions that cannot be executed by the developer on a recurring basis.

Besides human involvement in the development process, a set of project and environmental variables influence process performance and project success. For example, software lifecycles will compensate differently for late changes in the software requirements. Or, different organizational structures (e.g. outsourcing of code development) will require different processes and management strategies. In practice, processes are hardly ever executed in isolation but are part of a framework of processes. For example, if one studies the number of defects that can be found by *code reviews* one has to consider in which phase code reviews are performed in the overall development lifecycle: performing code reviews continuously in an iterative model will lead to different results than performing code reviews in only one of the later phases in the waterfall model. Also, carrying out other processes to identify defects (e.g. unit testing) might affect the number of defects found by code reviews. Many of these challenges in software engineering are described in Fred Brook's essay "No Silver Bullet: Essence and Accidents of Software Engineering" (Brooks).

All these issues play a critical role that determines if a process can be executed as originally designed by its inventors. If a process is not executed as intended, then its output will differ from the intended output. In other words, quality characteristics of the software product will change. To sum up and to conclude, the assumption and motivation for this research work is the following:

Research assumption:

Following the process as intended increases the chance of building a product with the desired quality characteristics.

If this is true then:

Not following the process as intended introduces a risk that the desired quality characteristics cannot be achieved.

Therefore, detecting process violations and improving process conformance is the main goal of this thesis.

1.1 Motivation

Every part of research has to be motivated by real issues coming either from the research community itself or from outside (e.g. the industry) that are required to be solved. Therefore, I will motivate the work by presenting two example areas that illustrate the importance of investigating process conformance. In the first example area (Section 1.1.1), the focus will be on project management and how monitoring and improving process conformance can help illuminate the shortcomings of processes in building better products. The second example (Section 1.1.2) area will illustrate that process conformance plays a critical role in the execution of empirical studies in the research field of software engineering.

1.1.1 Problem Area 1: Software Development Projects and Management

During the development of a software product, processes, methods, techniques, and best practices are applied. The rationale behind a chosen set is the manager's or programmer's belief that these intended activities improve a set of project-important quality characteristics, e.g. completing the product in time and cost, or assuring high reliability and maintainability. Not following the defined processes leads to an increased risk of not achieving these goals.

Several works in the practical, applied software engineering field recognize the importance of investigating process conformance:

ISO

The need to check for process conformance has been widely noted in the field of *software process improvement* and *quality management*. Various ISO standards emphasize process conformance: ISO 9000 recommends we *"initiate action to prevent the occurrence of any nonconformities relating to product, process and quality system"* (Standardization, International Organization for) and ISO 12207 on software life cycle processes states *"It shall be assured that those life cycle processes (...) comply with the contract and adhere to the plans."* (International Organization for Standardization)

CMM

The importance of complying with a process is part of SEI's[1] well-known Capability Maturity Model (CMM[2]) and its successors. Humphrey (Humphrey) writes that if process violations are not identified they will *"accumulate and degrade it [the*

[1] Software Engineering Institute, Carnegie Mellon University: http://www.sei.cmu.edu/

[2] CMM: Capability Maturity Model characterizes the state of companies current software practices and has been developed at Carnegie Mellon University. Capability Maturity Model is a service mark of Carnegie Mellon University. CMM is registered in the U.S. Patent and Trademark Office

process] beyond recognition". Humphrey further points out that if developers are carrying out the process in their own way that they will *"continue to make and remake the same mistakes"*. CMM introduces steps that describe that process activities should be reviewed. However, the model does not define how to do this nor does it provide tools to do so.

To support Humphrey's view, a study investigating the relationship between conforming to CMM processes and software correctness by Krishnan and Kellner (Krishnan and Kellner), the results *"(...) indicate that software projects that consistently adopt the CMM practices exhibit significantly lower numbers of defects. Thus, our results provide a link between consistent software processes and reduced field defects in the resulting product."* They provide evidence for a correlation of process conformance and product quality characteristics (i.e. correctness). To assess the amount of conformance to a CMM practice project managers had to rate their own conformance on a 5-point scale. A set of 45 software projects in one organization was investigated in this study.

Industrial Case Studies

More specific results of varying degree and effects of process conformance can be found in a case study by Arisholm et al. (Arisholm, Skandsen and Sagli). They describe process improvement activities in a real world project using a Rational Unified Process[3] model (RUP). They report "*In this case study, testing was performed too late in comparison with the prescribed process. Although it is, in retrospect,*

[3] www.IBM.com/Rational , *registered trademark of IBM*

uncertain whether this lack of process conformance could have been avoided by the development team, it is likely that it contributed to many costly last-minute changes to the software." This case study provides evidence for a negative impact of a lack of process conformance on the development cycle. The case study's method for detecting process deviations was interviews with the developers. When searching for reasons for low process conformance the researchers state that "*One explanation for this lack of process conformance was that the initiation and execution of the Genova process [scaled down version of RUP] at Braathens were quite informal".*

In the studies in professional environments presented in this thesis I will provide additional evidence that a lack of process conformance contributes to late projects. In the first study a lack of conformance to a process that lays out a development plan for the software could be identified, and indeed, the project was late in the end. For detailed results please see Section 6.3.

1.1.2 Problem Area 2: Empirical Studies

Empirical studies of processes in computer science help us understand the effect of different approaches and what environmental variables influence their behavior. Understanding and quantifying the effectiveness of a process in different environments helps selecting the right process in a given environment, and to verify that a process actually works as expected. While studying a process, different study designs can be used, e.g. controlled experiments. These experiments provide evidence for the process' applicability and effectiveness.

The importance of process conformance is especially stressed in the field of software engineering due to issues that arise when performing empirical studies.

When studying a process, the process itself is the central controlled variable of the experiment. It also can be seen as the treatment of the experiment. A number of other independent variables (e.g. experience of process enactors) can be either controlled, randomized, or uncontrolled. The measures for effectiveness of the process are typically the dependent variables.

The process, as being the most important controlled variable, is often times assumed to be executed as defined. In other words, it is controlled by explaining the steps and importance of the process to the process enactors. In many cases, the proper execution of the process is then taken for granted and not further validated. Working with humans, however, introduces a larger set of concerns and random behaviors:

After explaining the process, study subjects might still have a different understanding of the steps to be executed depending on various factors. These factors include but are not limited to:

Desire to succeed: subjects might have their own goals in mind when executing a process. In the classroom, subjects might be (or might not be) motivated by a grade they receive for performing the process. In industrial settings, process enactors might be differently motivated based on their role in the organization. For example, a quality assurance manager might be more motivated when executing a quality process than a temporarily hired student programmer.

Personal skills: some processes, process steps, or process definitions might be complex and hard to understand and require an amount of upfront training. If training is not provided, differently skilled subjects will follow the studied process more or less closely.

Prior experience: subjects might behave differently based on their prior experience with the process, or similar processes. For example, subjects having had negative experience with the studied process in the past might tend to modify the process steps, specifically the steps that they see as cause for the negative impact.

Duration and long-term motivation: In long-term studies subjects might intentionally or unintentionally leave out steps of the process or modify them in other manners. For example, some of the process steps might be too hard, or infeasible, or too costly to execute.

Several research works have emphasized these issues: Shull et al. (Shull, Carver and Travassos) state that: *"Data collection of all types in empirical studies must address the question of process conformance. Empirical results are not of much use if the researcher cannot be sure of which process produced them!"*

Further, evidence has been presented that supports the belief that process conformance is an essential ingredient when performing controlled experiments. And that subject's conformance indeed varies:

In an empirical study investigating reading techniques conducted by Lanubile and Visaggio (Lanubile and Visaggio) researchers found that *"(...) less than one third of Checklist reviewers could be trusted to have used the checklist and one fifth of the PBR [PBR = perspective based reading] reviewers could be trusted to have followed the assigned scenario."* They concluded that *"This experiment provides evidence that process conformance issues play a critical role in the successful application of reading techniques and more generally, software process tools."*

Another study on reading techniques by Laitenberger et al. (Laitenberger, Atkinson and Schlich) reported: *"With PBR it is possible to check process conformance explicitly by examining the intermediate documents that are turned in. We did this, and determined that the subjects did perform PBR as defined."* Interestingly, the first experiment investigated conformance with post study interviews. The second one used artifacts collected during process execution. Both studies come to very different results regarding conformance. A standardized way to detect and measure conformance would have helped to understand whether the subject's conformance differed or whether the process differed.

More recent work of Kou and Johnson (Kou) builds mechanism to classify different kinds of Test Driven Development. Their survey on related work shows that experimental results investigating the impact of Test Driven Development is highly diverse. They state that *"[...] research on TDD suffers from the "process compliance problem". In other words, the experimental designs do not have mechanisms in place to verify that subjects who are supposed to be using TDD practices are, indeed, using them."* In one of their studies verifying a classifier to distinguish between test-first and test-last order they conclude: *"A provocative result of this study is that half the episodes (46) were classified as test-last, even though the subjects were instructed to do test-first development."* [4]

[4] In our own work (described later in Chapter 6.2) investigating conformance to XP practices results show that students followed Test-Driven Development in at most half of the cases.

The reasons for subjects to violate processes can vary. Basili et al. (Basili, Shull and Lanubile) point out that human subjects are often motivated by their own goals during the study:

"Subjects are not malicious, but will sometimes concentrate on successfully accomplishing what they see as the goal, even if it means straying from the process assigned." (Basili, Shull and Lanubile). In an experiment investigating the effectiveness of a reading technique to detect defects in requirement specifications they found that the student subjects reported many false positives because they believed that the more defects they find the better the grade they will receive from the professor.

In summary, the above sources provide evidence for process compliance problems in empirical studies. Some research work has tried to check for conformance at the end of the study through an end-of-study questionnaire. However, no generally applicable and scientifically accepted approach has been presented yet to check for conformance in studies *during* their execution. This work will propose such an approach and will check its validity by applying it in typical controlled study settings in classroom.

1.2 Terminology

Throughout this thesis a set of expressions and terms is going to be used that first need introduction. This chapter will provide detailed explanations and definitions.

Process (in software engineering) is usually *a set of steps* performed by process enactors on an input (i.e. software artifacts) producing a desired output. The output can either be a transformed version of the input (e.g. code development) or a product distinct from the input (e.g. extracting a list of defects from a requirements document).

I am going to use the term *process* in a different, much broader sense. *Process* will be an **umbrella term** for a vast set of terms used by software engineering literature such as: software life cycle model, method, technique, software process, and practice. These terms mainly distinguish between different levels of process application. For example, high level software life cycle models can be decomposed into lower level processes, e.g. the waterfall life cycle can be decomposed into processes for requirements specification, design, coding, and testing. Processes can be further decomposed into methods and techniques. However, what they all have in common is a set of instructions or expectations that have to be followed in order to produce outputs. Further, I include weak and informal descriptions of processes, such as *guidelines and practices,* that only define "what should be done" and not "how exactly it should be done" (e.g. as a sequence of steps). An example guideline could be that "*all developers in a team should write test cases*". Guidelines do not define specific steps but will still be testable in the proposed approach of this thesis.

Process Definition is the representation of the process that can be communicated across process designers and enactors. It can be thought of as a model containing the process specific details. On the scale of process specificity, a definition can range from an informal guideline, given in natural language, to a formal process specification that describes the order of expected steps of the process (e.g. given as a finite state machine).

Process Enactment (also executed process) is a series of actions performed by the process actors in reality. Process enactment includes the steps that were performed, their order, the quality of execution, and the time or effort spent in executing the steps.

Process Conformance (also process compliance) is the concept that describes how closely a process enactment complies with the process definition. This follows earlier definition (Sorumgard): *Process conformance is the degree of agreement between the software development process that is really carried out and the process that is believed to be carried out.*

Process Violations are errors in process enactment that violate the process definition, i.e. specific non-conformances. For example, process violations are omission of steps, modification of steps, rearrangement of the order of steps, or the introduction of new steps. Additionally, violations include poor qualitative execution of steps.

2 Research Problem

The observations and findings from the previous chapter highlight the importance of investigating process conformance. Before proposing methods to do so, a more precise description of the research problem is presented in this chapter. The problem will be broken down into sub problems and tradeoffs that exist when solving these problems will be discussed.

The main research question this work is trying to answer can be stated and broken down as follows:

(A) Can mechanisms be built to detect process violations in software processes

(B) and are those findings useful for

- understanding which aspects of the process are not being applied properly and why?
- improving conformance and increasing chances of achieving desired quality characteristic of the software product?

The research question can be broken down into two parts. Each of these parts is discussed in greater detail in the two following subsections. The first part (A) is concerned with the formulation and detection of violations against a planned and expected process. The challenge is to build a method that is on the one hand general enough to detect process violations for wide range of software processes and on the other hand cost efficient enough to be feasible to be applied in practice. The second part (B) is concerned with the interpretation of the violations in the context of the analyzed process within the observed environment (e.g. the software development project). Finally, part (B) questions how one can improve conformance in the long run so that the processes' quality goals can be predicted more precisely.

2.1 Violation Detection Mechanisms

In order to detect process violations during process execution one has to compare the actual executed process (i.e. the **process enactment**) to the **process definition**. To do so, the executed process has to be instrumented and measured. Ideally, one could measure all steps of the process including the quality with which it was executed and the time it took to execute the steps. In reality (i.e. in empirical studies and in industrial software projects) the amount of measurable steps of a process is limited by two tradeoffs that have to be balanced:

The **cost of measurement** has to be considered and should be proportional to the gain of knowledge produced by finding process violations. Ignoring the cost could lead to an approach that in theory can be shown to have certain properties, but will not be applicable in practice due to economic reasons. Further considerations concerning the cost are given in section 2.1.2.

Measurement activities can be **too intrusive** and may change the behavior of the process enactors and therefore the process execution itself. Details on intrusiveness are given in section 2.1.3.

In addition to these two tradeoffs there might be a set of privacy concerns attached with the instrumentation methods. For example, video and screen capturing of process enactors might also capture non-process relevant parts, such as email or personal conversations.

Before going into details on these two tradeoffs, I will discuss which instrumentation types exist and how the different types can be roughly classified.

2.1.1 Classification of data collection methods

A number of different methods help instrumenting and measuring process execution. They can be roughly divided into data acquisition methods that take *automatic* measurements[5] and methods that involve *manual* effort (Figure 2: y axis). The latter, manual methods require a fair amount of human involvement in the phase of producing measurements or in the phase of analyzing the data. For example, manual effort is required from the process enactors in order to fill in checklists and create

[5] Example for such tools are: Hackystat (Johnson, Kou and Agustin), UMDInst „ Subversion(http://subversion.tigris.org/), CVS (*www.nongnu.org/cvs)*, Cruise Control *cruisecontrol.sourceforge.net/)*

measurements. And, manual post analysis of video data requires additional effort from the data analyst.

Further, one can classify these data collection activities into *existing* and *supplementary* collected data (Figure 2: x-axis). Existing data is data that is already collected as part of the software project (e.g. a bug tracking system or a code repository). Supplementary data is data that is not yet collected, and requires additional cost to be collected (e.g. developers filling in checklists for the purpose of conformance measurement).

Both classification characteristics for measurement methods are likely to vary from project to project and from one process to another. Different projects will collect a different set of existing measures and different processes will require different types of automatic and manual data collection activities. For example, one project might already employ a version tracking system, whereas another one does not. Accordingly, some processes will be harder to measure automatically, e.g. if one wants to measure the thought process of subjects while they are executing a process one will likely have to use a manual method , such as an interview or questionnaire.

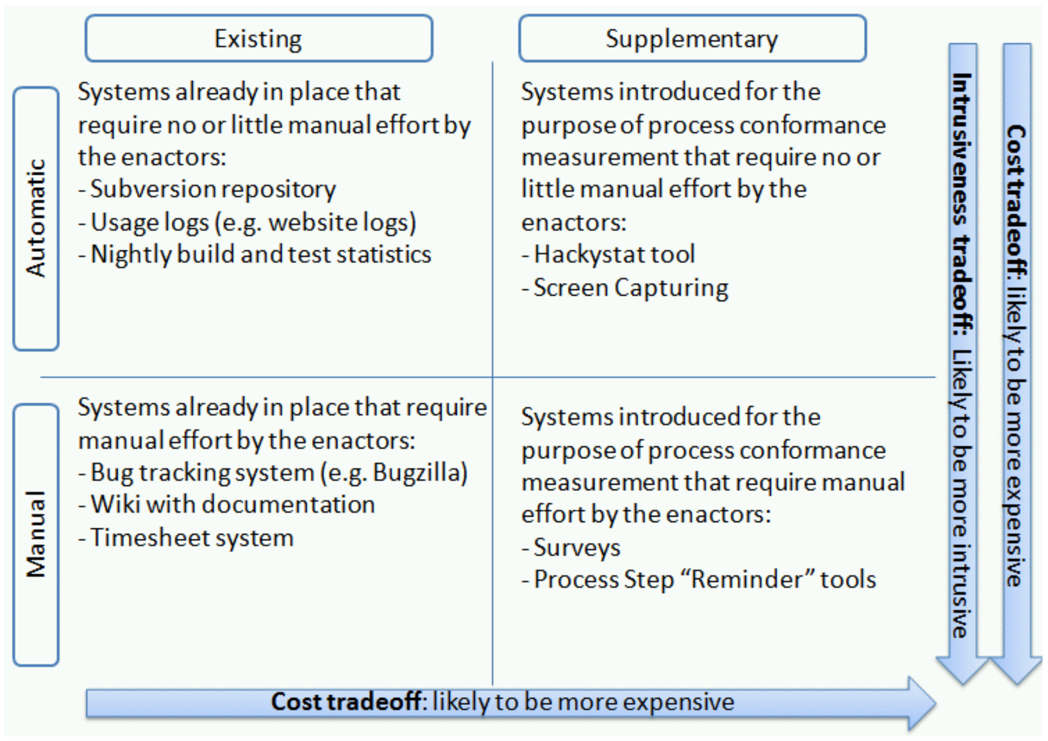

Existing
Supplementary
Automatic
Manual
Systems already in place that require no or little manual effort by the enactors:
- Subversion repository
- Usage logs (e.g. website logs)
- Nightly build and test statistics
Systems introduced for the purpose of process conformance measurement that require no or little manual effort by the enactors:
- Hackystat tool
- Screen Capturing
Systems already in place that require manual effort by the enactors:
- Bug tracking system (e.g. Bugzilla)
- Wiki with documentation
- Timesheet system
Systems introduced for the purpose of process conformance measurement that require manual effort by the enactors:
- Surveys
- Process Step "Reminder" tools
Cost tradeoff: likely to be more expensive
Intrusiveness tradeoff: Likely to be more intrusive
Cost tradeoff: likely to be more expensive

Figure 2: Classification of process instrumentation methods and tradeoffs between cost and level of intrusiveness for process instrumentation

2.1.2 Cost of Measurement (Cost Tradeoff)

The first tradeoff to be considered when measuring a process is the cost required to measure and analyze process conformance. Process instrumentation methods that are existing and already applied will require no or little additional cost (e.g. when they need to be slightly modified). However, even if the data comes for free, additional cost has to be spent during the data analysis process.

Supplementary methods will always require spending additional cost in data collection and analysis.

When comparing the cost of automatically collected data to manually collected data it is likely that the latter will be more expensive due to the human involvement.

A successful approach for detecting process violations will have to consider the cost tradeoff and primarily focus on the cheaply available, existing and automatically collected data.

As a rule of thumb, in industrial settings the cost overhead of the data collection and analysis methods has to be proportional to the gained insight and payoff in increased productivity triggered by improved process conformance. In empirical study settings where the process manager, i.e. the researcher, wants to limit threats to validity he or she will usually spend even more effort on ensuring process conformance.

2.1.3 Intrusiveness of Measurement (Intrusiveness tradeoff)

Whenever new manual methods for process instrumentation are used they are likely to influence the way process enactors execute the original, non-instrumented process. Firstly, methods that make enactors feel that they are being observed can change their behavior. This effect, also known as the Hawthorne effect (Roethlisberger and Dickson), embodies a strong threat to internal and external validity in controlled experiments. Subjects might stick to the process because they feel that their conformance is being studied. In a regular (unobserved) environment, however, the subjects might modify the process more freely.

A second issue can be that instrumentation methods impair the natural flow of a process by interrupting the enactors. For example, an instrumentation method compromising a check list that needs to be filled after each step will constantly interrupt enactors. Therefore, instrumentation methods should have a low level of intrusiveness. As a guideline, one should first use all possibly available automatic and unobtrusive instrumentation and only apply manual, intrusive methods if absolutely necessary. If one has to choose intrusive methods it is recommended to ask the process enactors whether they felt they were being observed, or if the instrumentation method interrupted their workflow.

Figure 2 illustrates the tradeoffs for process instrumentation methods and provides examples for an imaginary project.

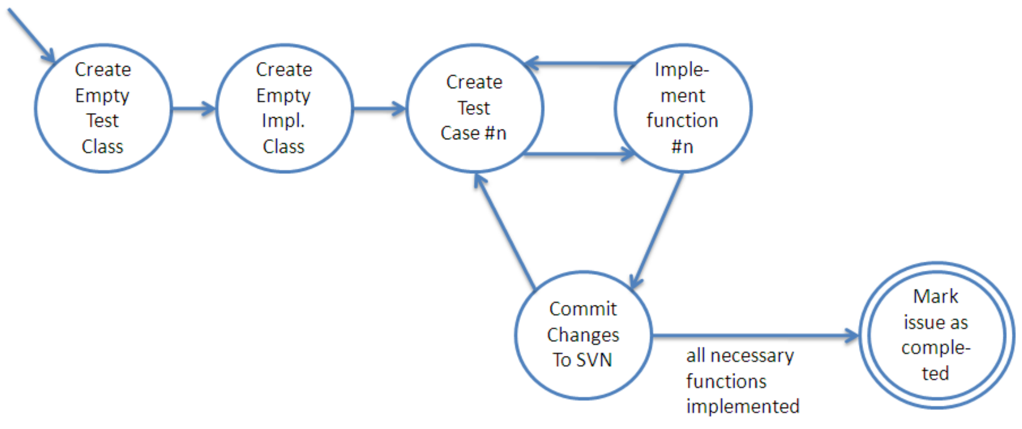


Figure 3: A sample Test-Driven Development process

To exemplify the tradeoffs, Figure 3 visualizes an example process graph for Test Driven Development. The idea of the process is that developers **first** implement a test case before writing the implementation of a function. A possible process flow is illustrated in Figure 3 and could require first creating empty classes for test and implementation, then iteratively creating test cases and implementing functionality. Optionally, developers are allowed to submit their changes to the code repository after they finish implementing the function (in the figure SVN=Subversion). When all necessary functionality is implemented, the developer should mark the task in the issue tracking system as completed. For the objective of this example one should assume that a SVN repository and a bug tracker are already part of the regular software development environment.

Investigating the different steps of the process will require different methods and it will differ in the amount of measurement cost. As an example, the steps “Commit Changes to SVN” and “Mark issue as completed” are cheap to measure since existing systems (SVN and issue tracker) capture this kind of data implicitly (i.e. existing data) and can be queried. The first steps of creating files on the local development machine could be measured by instrumentation tools (e.g. Hackystat (Johnson, Kou and Agustin)). The cost of measurement includes installing these tools and post processing the collected data. Last, the steps of iteratively creating test cases and implementation could be measured by either providing developers with a checklist that keeps track of the order of implemented functionality or by capturing screen content for a manual post analysis. The first solution might introduce effects that change the usual behavior of developers. The second approach requires costly post data analysis and inherits a set of privacy issues (e.g. capturing screen content with private information such as email content). In practice, these steps are costly or even infeasible to measure in the long run (e.g. over the duration of the project). In summary, one might *not* be able to measure all steps that are given in the above picture. This circumstance can be found in almost any process applied in practice, and it will influence the properties of the research question and approach.

2.1.4 Properties on an Incomplete Approach

As explained in the last section, within a fixed budget for measurement activities, the use of existing and automatic available data sources is recommended first. But measuring only a subset of process steps will hide certain details of the process execution and therefore will result in an approach that cannot detect **all** violations that occur. However, even detecting a subset of all process violations can give a fair amount of insight into process execution, as will be demonstrated in the studies that follow.

Based on the cost and intrusiveness tradeoff, **the approach presented in this thesis will be able to show the presence of violations but not their absence**. In other words, it will be able to detect violations against the process definition but not be able to **prove** that a process has been followed completely. The approach is therefore sound but not complete and can be thought of as "Process Conformance Testing" (as opposed to "Software/Product Testing").

Figure 4 illustrates the concept of violation detection by using limited measurement data. The red box shows that the process enactment in the picture violates the process definition in three different ways. Operating on a subset of data (in the figure: box Measured Process) allows only for the detection of a subset of actual violations.

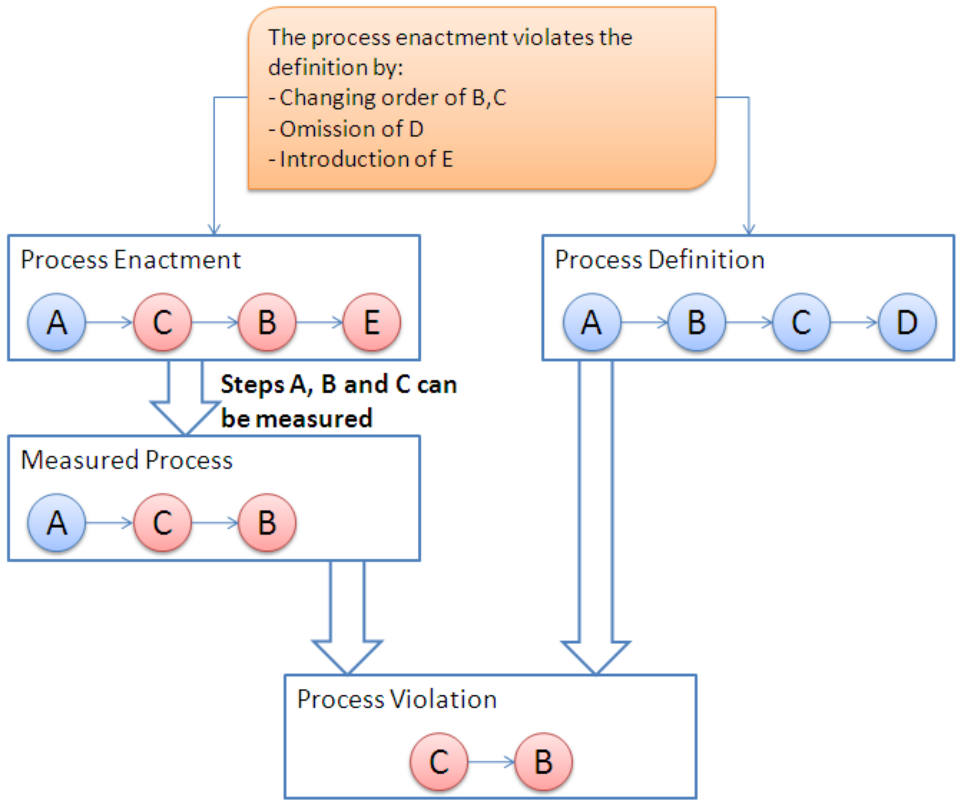


Figure 4: Detecting a Subset of Process Violations

2.2 Violation Understanding and Conformance Improvement

The first part of the research problem is concerned with the detection of violations against the defined process. Once violations are identified the person investigating process conformance has to be given the ability to improve conformance. A couple of questions arise when violations are detected, such as:

- What do the violations mean?
- How severe are the violations?
- What are the reasons for these violations?

Therefore the second part of the research question is concerned with "what are the right procedures and tools to further investigate identified violations?" These mechanisms should be able to give insight into different dimensions of the violations such as:

Number/percent of violations: that is, how often (out of all possible cases) did the developers not follow the process. For example, it would be perceived differently if a violation occurs in every second instance of the executed process, or just in 1% of all instances.

Type of violations: in the example process given in Figure 3 different steps can be violated. For example, if developers forget to close the issues in the last step then this type of violation would require a different reaction than a violation against the steps of implementing test cases prior to functionality.

Timing of violations: Violations in different stages of the software development life cycle might be perceived differently. For example, violating Test Driven Development in late stages of the projects might be more severe since there might be not enough time to test the code thoroughly.

Location of violations: if violations can be attached to specific parts of a software system then the location might play a role in how to react. For example, violations against Test Driven Development can be assigned to the source component that was not developed according to the process. Then, violations occurring in core components of the software system might be more severe.

Additional measures and information: additional measures can give further insight and understanding of the violations. For example, determining the developers associated with the violation can answer the question if only a few, or if all developers have problems following the process. This gives insight into the applicability of the process. Besides developers, different software measures can explain violations. For example, one might find that, in all the cases that the process was violated, the software components were extremely small (e.g. measured by lines of code). Depending on the process, one might conclude that the process is not applicable for small software components.

The last challenge in the process of improving conformance is how to use the understanding gained from the set of detected violations and to determine if the process violations have a deeper meaning. For example, violations can be symptoms of root causes that are often not immediately visible: developers might skip specific steps because of time pressure. Or, the process might not have been explained to them precisely enough. In other cases, a process might not be applicable in the given development environment and might need to be refined.

3 Related Work

As already stated, different parts of the applied and empirical software engineering fields have recognized process conformance as important ingredient for process analysis and improvement, e.g. ISO (Standardization, International Organization for) (International Organization for Standardization) and CMM (Krishnan and Kellner) report about desired activities to investigate conformance. In the empirical field, researchers have found process conformance to be important when running experiments (Lanubile and Visaggio) (Laitenberger, Atkinson and Schlich) (Kou) (Shull, Carver and Travassos). Related approaches to monitor, assess, and enforce process conformance have been proposed in the past and are presented in the following sections,

3.1 Review Procedure

In order to find closely related approaches *focusing on investigating whether software developers are following a planned process* a systematic review (Kitchenham, Pfleeger and Pickard) has been conducted. The goal of the review is to find all related research that tries to answer the stated research questions (Chapter 5). In the following I will describe the procedure of the review and its results.

3.1.1 Systematic Review Procedure

To find related research articles I used the Google Scholar search engine (http://*scholar.google.com*). The advantage of the search engine is that is searches a long list of publishers, journals and conference proceedings[6]. Different keywords for *paper titles* were used and are reported in the results section. Filtering of the results for all keywords was done in two steps: first the title and conference name could reveal that the work is not significant, second the abstract and conclusion were used to filter further.

The second step of the review was an inspection of the referenced works in both directions. That means that the list of references in the document itself was inspected and the list of referring documents was inspected too (Google Scholar provides this information).

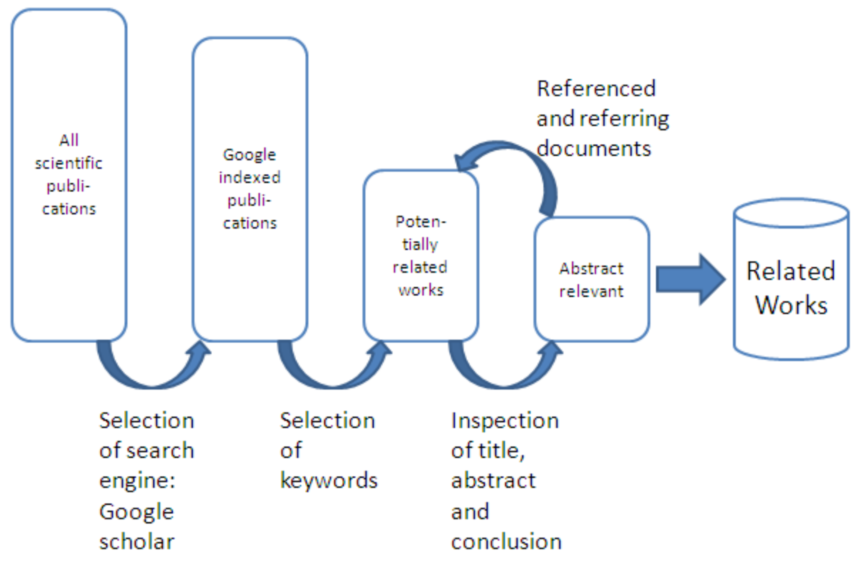

All scientific publi-cations
Google indexed publi-cations
Poten-tially related works
Abstract relevant
Related Works
Referenced and referring documents
Selection of search engine: Google scholar
Selection of keywords
Inspection of title, abstract and conclusion

Figure 5: Procedure for Systematic Review of Related Work for this Thesis

[6] Work by Walters **Invalid source specified.** showed that precision and recall measures of Google Scholar are higher than competing digital libraries. Therefore Google Scholar can be justified to be used in such a search.

Iteration	Keywords[7]	Identified documents	Remaining after filtering by title, abstract, and conclusion	Additional sources from referenced documents	Additional sources from referring documents
1	Process Conformance Process Compliance Process Mining Process Extraction Process Discovery Process Validation Process Violation Process Verification Process Enactment Workflow Mining Process Non-Conformance Process Non-Compliance Process Nonconformance Process Noncompliance[8]	564	29	1	3

The criterion used for inspecting and filtering the results by title and abstract was that the work had to deal with process conformance of software processes (and not business processes, or medical processes).

3.2 *Related Work*

3.2.1 Process Centered Software Engineering Environments

Multiple research activities, mostly developed in the early 1990s, focus on building *process centered software engineering environments* that support process enactment of software processes in an automated fashion (Bandinelli, Fuggetta and Ghezzi)

[7] Further it was selected in Google Scholar to search articles in the field of "Engineering, Computer Science, and Mathematics." (this option can be found in the advanced search options)

[8] The exact Google Scholar search string was: "Process Conformance" OR "Process Compliance" OR "Process Mining" OR "Process Extraction" OR "Process Discovery" OR "Process Validation" OR "Process Violation" OR "Process Verification" OR "Process Enactment" OR "Workflow Mining" OR "Process Non-Conformance" OR "Process Non-Compliance" OR "Process Nonconformance" OR "Process Noncompliance"

(Broynooghe, Parker and Rowles) (Leonhardt, Kramer and Nuseibeh) (Reis, Reis and Abreu) (Schramm, Verlage and Knauber) (Kroeger, Jacobs and Marlin). The goals of these environments is on the one hand to provide process designers with a process modeling language to express processes in an explicit form (e.g. PML: Process Modeling Language (Broynooghe, Parker and Rowles), SLANG: SPADE Language (Bandinelli, Fuggetta and Ghezzi), APSMEE-PML (Reis, Reis and Abreu), ProLan (Schramm, Verlage and Knauber)) and on the other hand to support the process enactors with an electronic system that lists the activities they have to execute next. Further, some of the environments (e.g. SPADE (Bandinelli, Fuggetta and Ghezzi)) are able to collect data from external tools automatically (such as a compiler). Different storage solutions (e.g. object oriented databases (Broynooghe, Parker and Rowles)) are used to keep track of process evolution.

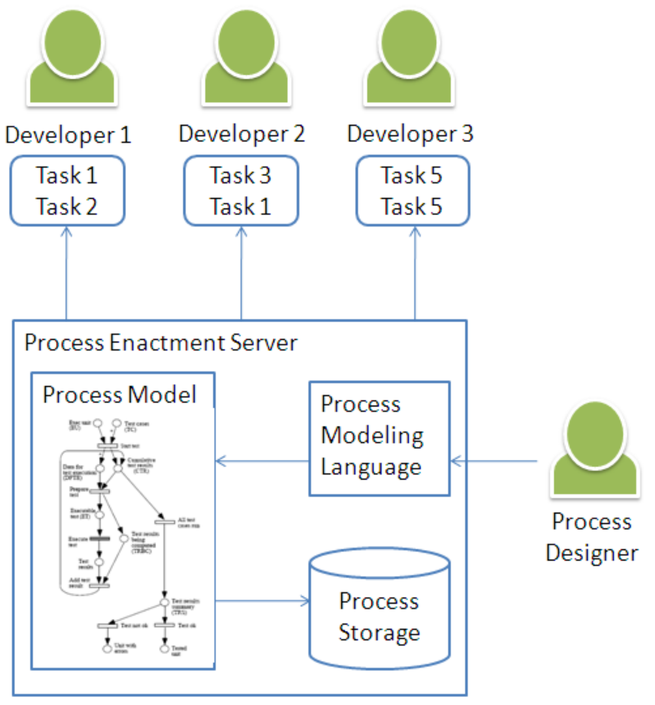


Figure 6: Process Centered Software Engineering

Figure 6 illustrates a general model of a process centered environment. The process designer uses a process modeling language to convert the process into an explicit form. Then the system uses this description to hand out tasks to the set of developers. Some of the proposed systems are not centralized as shown in the figure, but are decentralized (Leonhardt, Kramer and Nuseibeh).

The systems require that all development processes are translated into the specific process modeling language and that process enactors invest effort in maintaining the state of the process and strictly follow it. Bruynooghe et al. (Broynooghe, Parker and Rowles) claim that *"Ensuring conformance to process is often espoused as the main benefit of process enactment. For example, one can guarantee the timely performance of mundane repetitive tasks, which otherwise may be neglected by process participants"*.

Very recent work by Mishali et al. (Mishali) presents a system (TDD Guide) that supports developers performing Test Driven Development (TDD). In contrast to earlier approaches the system is tailored to one specific agile practice. It observes the steps of creating test and implementation classes in the developer's IDE and warns if the TDD practice is violated. The system assumes that developers know which steps they have to follow for TDD and acts passively (i.e. does not enforce the process). In the case study presented, data collected through questionnaires indicated that the system helped developers follow the practice. However, it was not investigated (e.g. through a control group) if developers follow TDD with higher conformity using the tool than without using it.

Commonalities and Differences to this work

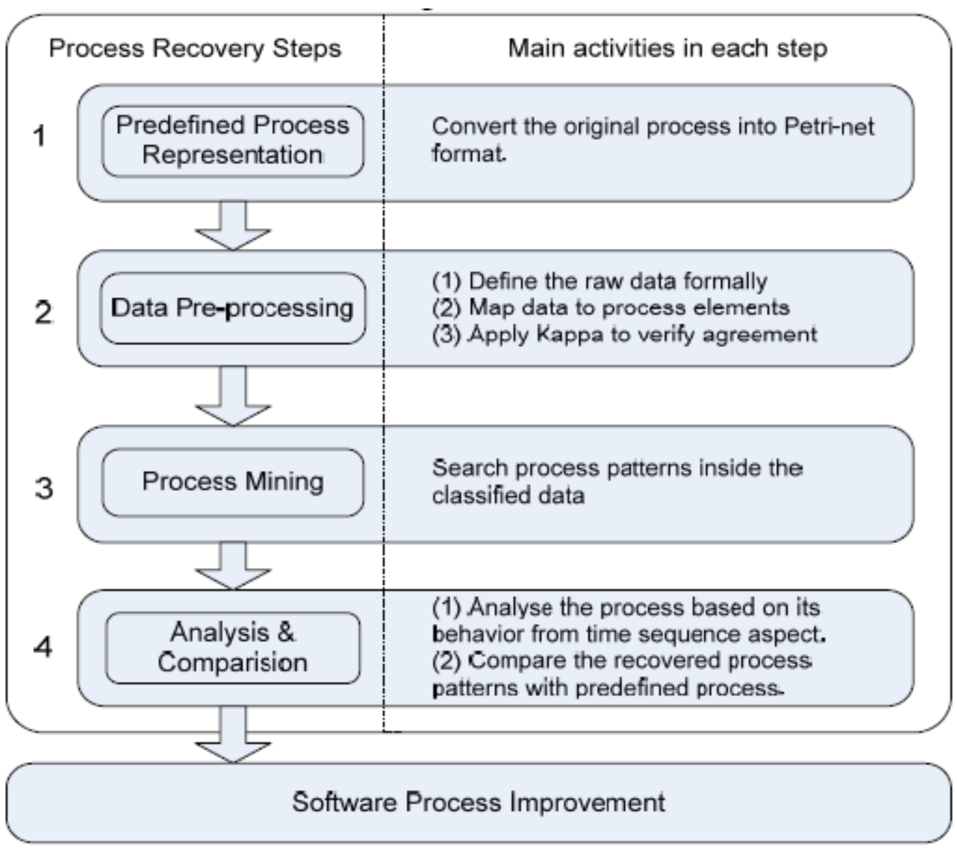
The work presented in this thesis takes a different approach for improving conformance to a software process. Instead of telling the developers *what to do* the approach analyzes process data and gives developers feedback on what *they did wrong*. Further, the approach will not be restricted to one specific process specification language (e.g. FSMs, Petri Nets) but will allow defining the process in *any formalism of choice*. I see this as an important property of a general approach. Different processes will require different models because each model brings along a different power of expressiveness. For example, FSMs are not able do model concurrency without state explosion, or Petri Nets are only able to defines temporal properties (e.g. in which order steps have to be carried out) but not qualitative properties (e.g. *how* steps have to be carried out). Another good example for the necessity of general models are *guidelines* that do not define steps at all. For example, the guideline *"Always write sufficient documentation"* does not define steps but developers should still adhere to it. FSMs and Petri Nets are not appropriate modeling mechanisms for such kind of development rules.

3.2.2 Process Mining and Process Discovery Approaches

The goal of approaches performing process mining is to discover process models from observed data. Those approaches assume that the process model is not given in advance, but can be constructed by investigating different type of data sources (e.g. logs or software artifacts).

One of the first and highly cited approaches that infers a process model has been presented by Cook and Wolf (Cook, Process discovery and validation through event-data analysis.) (Cook and Wolf, Discovering models of software processes from event-based data.). Assuming that process data is captured in the form of an event stream (the authors do not give specifics on how to measure this data in practice) three different known techniques (i.e. KTail, Markov, RNet) are used to construct finite state machines (FSMs) that represent the process model.

In more recent work Hou et al. (Huo, Zhang and Jeffery, An exploratory study of process enactment as input to software process improvement. In) (Huo, Zhang and Jeffery, A Systematic Approach to Process Enactment Analysis as Input to Software Process Improvement or Tailoring) build upon Cook and Wolf's work and extend it to map higher level events, such as major phases during the software development lifecycle. They show in a pilot case study that it was possible to build a high level Petri Net modeling dependencies between three high level processes (Collect Requirement, Software architecture design, Analysis). Mined low level activities were manually mapped to high level process elements by experts. Then this discovered model was compared to an expected one so deviations could be identified. Figure 7 (from their paper) illustrates their process.



Process Recovery Steps
Main activities in each step
1
Predefined Process Representation
Convert the original process into Petri-net format.
2
Data Pre-processing
(1) Define the raw data formally
(2) Map data to process elements
(3) Apply Kappa to verify agreement
3
Process Mining
Search process patterns inside the classified data
4
Analysis & Comparision
(1) Analyse the process based on its behavior from time sequence aspect.
(2) Compare the recovered process patterns with predefined process.
Software Process Improvement

Figure 7: Hou et.al. Process Recovery Approach (copied from (Huo, Zhang and Jeffery, An exploratory study of process enactment as input to software process improvement. In))

Work by Jensen and Scacchi (Scacchi and Jensen) investigates how events can be extracted from existing historical data sources in Open Source Software Systems (OSSS). They describe how software repositories, forums, and issue trackers are promising candidates for event data mining.

Rubin et al. (Rubin, Günther and van der Aalst) describe in their work how software repositories can be used to derive explicit process models. Their ProM framework *"provides a variety of algorithms and supports process mining in the broadest sense."* Their idea is to map activities extracted from logs of the software repository (i.e. Subversion) to higher level events. For example, a modification of any source code file in the “/tests” directory was mapped to an event that describes that test cases were modified. In a case study they investigate several OSSS (i.e. five ArguUML subprojects) and derive a Petri Net that shows the order in which major parts of the software were created. They also describe how such a model can be used to check the compliance of activities in accordance to a Petri Net. Further, they describe an LTL checker to analyze the repository logs (i.e. the LTL rule defined that *“developers working on the source code should not write tests as well”*). The LTL checker, which is described in more detail in (De Beer and Van Dongen), does not require deriving a formal model first but is used for checking the collected data directly.

Commonalities and Differences to this work

The approach presented here assumes that the process is given and does not have to be mined and extracted from data. Therefore, this work is trying to answer different research questions. The assumption that a process definition is given is reasonable to make. In the studies conducted (in professional and classroom environments), the person interested in studying conformance usually had a good idea about what the expected process should look like. For example, in the professional programmer study the process was explicitly defined in an Excel spreadsheet. In the classroom study, the researchers picked well-known eXtreme Programming practices that were defined in literature. In the ongoing industrial case study the project manager provided verbal definitions of the processes and guidelines that he expects the developers to follow.

Additionally, my approach builds a whole framework with steps that describes the process, starting with defining how and what data has to be collected to detect process violations, and ending in a step that describes what actions can be taken to avoid non-conformance in the future. The approaches described above often do not investigate those steps. For example, the Cook and Wolf approach does not define what data to collect and how this can be done at reasonable cost and without interfering with the process itself.

The LTL checker presented in (De Beer and Van Dongen) defines undesired temporal patterns in the collected logs. Our work also defines undesired patterns in collected data. As explained in the previous section, picking one formalism (i.e. LTL) restricts what kind of patterns one can find (i.e. only temporal ones). Our approach allows a much broader range of checks, such as checks for qualitative measures (e.g. through code metrics) and checks of guideline rules that do not define a temporal order.

3.2.3 Process Verification and Violation Detection Approaches

The following research aims at verifying process execution and detecting violations against process definitions.

Cook and Wolf (Cook and Wolf, Software process validation: quantitatively measuring the correspondence of a process to a model.) offer a theoretical approach on how to measure the distance between an executed and defined process. Their work is based on the event based framework that was introduced earlier by the same authors. An executed process is expressed as a stream of events. They propose different string distance metrics to qualitatively assess the differences from the process model stream.

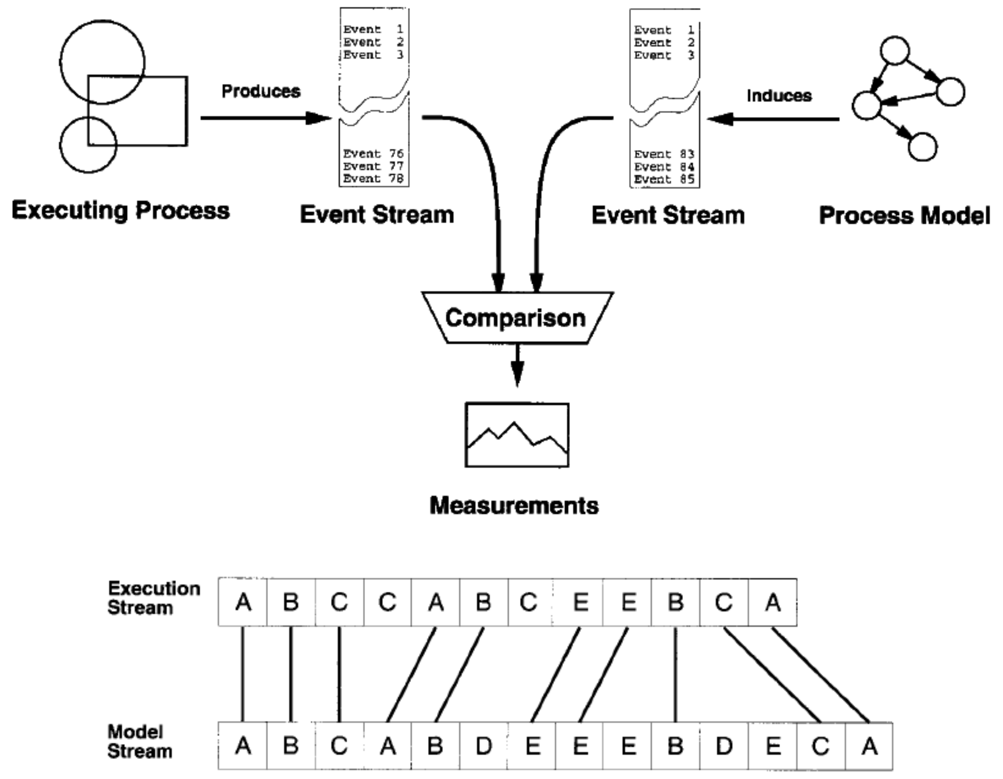


Figure 8: Cook and Wolf's approach (figure copied from (Cook and Wolf, Software process validation: quantitatively measuring the correspondence of a process to a model.))

Figure 8 shows the principle idea of the approach. In comparison to the work proposed here Cook and Wolf's approach requires a formal process model and the induction of event streams on both ends. The proposed approach will neither require converting collected data (e.g. data from a code repository) into events, nor process models that can be converted into event streams. Besides detecting deviations the proposed approach will expand beyond solely detecting violations. It will help to understand what data has to be collected and how conformance can be improved based on the findings.

Another approach to assess conformance as quantitative measurement was proposed by Sorumgard (Sorumgard). His idea is to take a series of scalar process measurements (e.g. total time spent, number of defects found) and to calculate a deviation vector based on the measured data and an expected vector. This method assumes that a process, when executed correctly, will always produce the same measurements.

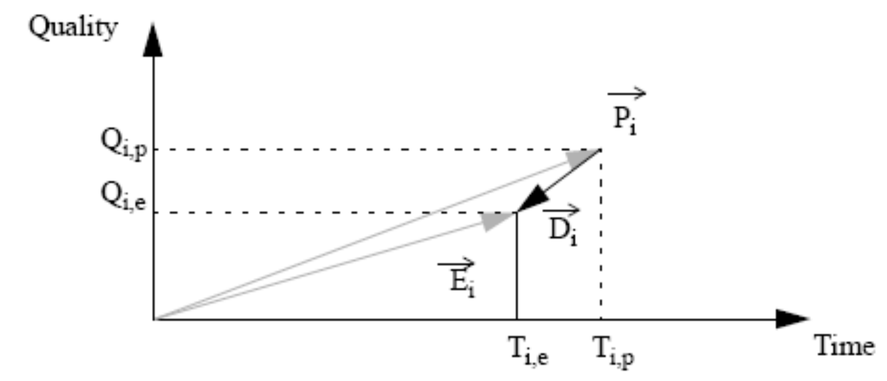
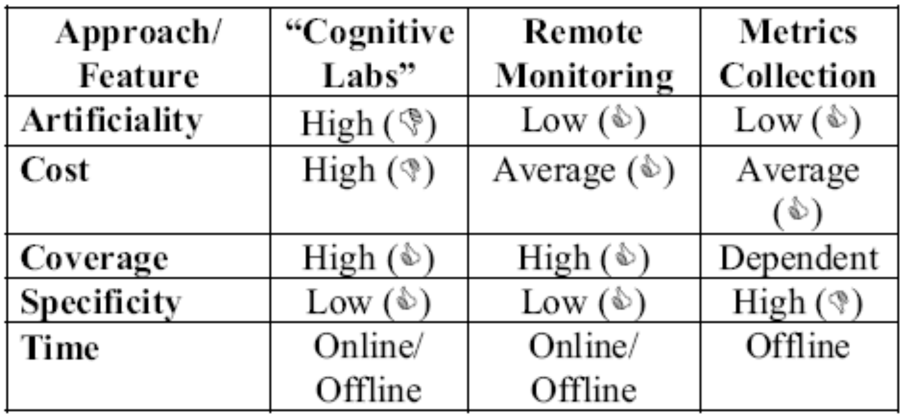


Figure 9: Sorumgard's Deviation Vector (figure copied from (Sorumgard))

In comparison to the proposed approach this approach does not provide insight into *what steps of the process* were violated. Further, Sorumgard's approach is applicable after all measurements have been taken (i.e. after the study, at the end the project). The proposed approach will be able to give live feedback on detected violations.

Silva and Travassos (Silva and Travassos) discuss different methods for observing process execution in experimental settings. The discussed methods are *Cognitive*

Labs, *Remote Monitoring*, and *Metric Collections.* The authors highlight the importance of avoiding the introduction of a Hawthorne effect. In Cognitive Labs settings, subjects are video recorded (screen, subject and audio) and observed through a one sided mirror. Remote Monitoring captures screen content only (by the use of special software), and the last approach (Metric Collection) collects process metrics. The latter is the one this thesis follows. They estimate the artificiality, cost (software license, researcher effort), coverage (i.e. amount of insight into process execution), specificity (of applicability), and time (online or offline) of the three approaches. Their classification is shown in Table 1.



Approach/ Feature	"Cognitive Labs"	Remote Monitoring	Metrics Collection
Artificiality	High (👎)	Low (👍)	Low (👍)
Cost	High (👎)	Average (👍)	Average (👍)
Coverage	High (👍)	High (👍)	Dependent
Specificity	Low (👍)	Low (👍)	High (👎)
Time	Online/ Offline	Online/ Offline	Offline

Table 1: Classification of three different monitoring methods in Silva and Travassos

The evaluative judgment of the different items in the table was done by argumentation and not established through experiments. I will argue against three of their classifications for the method of metric collection. First, the cost of collecting metrics should be downgraded to low while the time that has to be spent in data analysis for Remote Monitoring is magnitudes higher than Metric Collection. My argument is the following: remotely monitoring screen content will take about as long as it took the developer to execute the process. Especially for long lasting processes, costs can be infeasible. For example, monitoring Test Driven Development for seven developer teams over five days, as will be presented in the second study, would take weeks to analyze. However, in the study installing and analyzing the relevant measures took one day for each development day (and live results could be used to give subjects feedback). Second, the approach presented here shows that the specificity of the type of processes that can be monitored is not as high as claimed. The experiments show that very different processes could be investigated. From my experience one of the processes in the second study that was *not* investigated was if programmers really followed *Pair Programming.* I could not find a reasonable measure in the collected data (i.e. Subversion) that indicated if the programmers really took the assigned roles and if both worked together on writing the code. Interestingly, Remote Monitoring could also not have covered this process that takes place outside the measured environment (i.e. the screen content). Third and last, the approach presented here of using metrics will be able to report violations almost as timely as the other approaches. If desired, the approach can be run in intervals of one minute to produce the detected process violations. Again, these arguments are subjective and need further empirical investigation.

In the remainder of the paper Silva and Travassos introduce a tool to perform Perspective Based Reading. The tool has strong commonalities with the earlier mentioned process centered software tools that support developers by presenting the next steps of the process. The researchers describe how log data from this tool can be used to identify if subjects skipped certain screens (each screen is a step in the process). However, they do not report whether and how often this was the case in the two studies they conducted.

In recent work, Thomson et al. (Thompson, Torabi and Joshi) present an approach based on conditions that are checked against collected process data. This idea is similar to the construction of algorithms that detect violations in this work. The conditions in Thomson's paper define process states and values that should hold all the time. The approach presented in this thesis formulates this the other way around: conditions that should never happen. Besides the commonalities, the work presented here goes beyond purely building mechanisms to detect violations. It cares about how software processes can be measured and how violations can be inferred from the processes' definition. Further, the work presented here has to be shown applicable to different real software processes. Thomson's work was only evaluated on a small and artificially created banking example (the example showed how cash is deposited at a bank).

3.3 Conformance in Other Research Areas

When approaching solving problems in one research area (Computer Science) it is sometimes worth looking if similar issues have been addressed in other research areas. This can help identify terminologies and concepts that can be adapted for another domain. The following paragraph summarizes some of that work that helps to find and understand research questions related to conformance in our field.

"Drug Compliance in Therapeutic Trails" is work by Pol Boudes (Boudes) summarizing issues arising in medical research. In this field subjects (i.e. patients)

have to follow processes for taking medicine to better their health. Medical experiments are therefore concerned if patients follow the instructions (e.g. the frequency and order of intake) precisely. Without providing all related works that Boudes references in his survey, following evidence has been found by medical researchers:

"In clinical practice, roughly one-third of patients comply adequately, one third comply somewhat adequately, and one-third do not comply at all". This indicates that patients have, as well as software developers, problems following the recurring steps of a process precisely.

"Poor compliance affects the course of many diseases, even those with a fatal prognosis". A relationship between following a process and its affects has been established in this field. This is also one of the most interesting research questions in our field. Further, *"Poor compliance is the most common cause of nonresponse to medication."*

"Because poor compliance can undermine the execution and validity of clinical trials, it represents an essential parameter in the analysis of the results". In medical experimentation measuring of compliance has become a standard. This is not the case in software engineering experimentation yet.

"We discuss two possibly coexisting scenarios: (1) the participant takes an incorrect quantity of the study medication, and (2) the participant takes the correct quantity but in an incorrect manner (e.g., the wrong schedule of intake or the use of forbidden concomitant medication)." Interestingly, the proposed work presented here will also

distinguish between syntactic and semantic conformance violations that mirror the same principle: the first is concerned about the order of executed steps, the second one about their quality.

"Although poor compliance is easily defined in theory, it is frequently difficult to measure in practice." This seems to be true also in software engineering. Defining the difference between a planned and executed process (e.g. as presented by Cook and Wolf) seems easy compared to the difficulty of measuring (and defining) a process precisely in practice.

"In fact, when patients are able to explain their noncompliance, they mention two main reasons: 'I forgot' and 'I didn't have the drug handy' ". In software engineering we have not explored yet why developers are not following the process.

"Irrespective of the disease or disorder studied, self-administered treatments are associated with poor compliance [...], and compliance with long-term treatment is worse than adherence in short-term studies [...]." These are also interesting questions in our field: do self-managed processes show more violations? And, is process conformance better in short term studies of process conformance?

"Many studies have shown that factors such as age, education, gender, intelligence, and race have only a limited influence on compliance". Another interesting question in our field is if certain programmer types (e.g. novice vs. expert, "Hacker" vs. software engineer) show different conformance levels.

"The design of drug containers and packages may influence compliance." Boudes argues that that the easier drugs are packed and the clearer statements about their process of intake is described the better the conformance. In software engineering we

can see “process packaging” as how well a process is integrated in the usual development cycle. For example, does the programmer have to do 10 clicks to perform the process in a separate tool? Or, is the process nicely integrated into the IDE and requires only little overhead? How does integration influence the overall conformance?

“Questioning patients is the most widely applicable method for evaluating compliance. [...] Careful questioning might identify over half of the noncompliers”. The conclusion is that most patients admit in interviews to be non compliant. In our study conducted in classroom with students we made similar observations. Students (who were not graded based on conformance) admitted to have followed a set of XP practices only poorly. However, in an organizational context the honesty of answers might change.

“The reliability of data from interviews depends on the quality of the relationship between the patients and the clinical staff”. We can learn from this that in attempts to measure conformance through interviews we first have to gain the trust of the developers. For example, we might explicitly explain how the collected data will be used and who is going to see it.

“*A memory-equipped electronic device [...], a control system that checks the hour and date when the medication package is opened, can automate pill counts”.* Tools to support the patient have been built. Workflow management systems also support the automation of process execution.

“These devices also detect ‘‘white-coat compliance,’’ that is, increased compliance just before and just after appointments with the investigators.” If compliance to a

software process increases around the time developers are made aware of it, then this awareness process should be repeated frequently to keep up the high level of compliance.

4 Approach

In order to address the research problem stated in earlier chapters I developed a step-by-step approach to identify and investigate process violations was developed. At the highest level the approach follows a four step iterative process as shown in Figure 10. Each area in the picture represents one step. Steps are executed by different roles and each step takes defined inputs from the previous step to produce defined outputs for the next step.

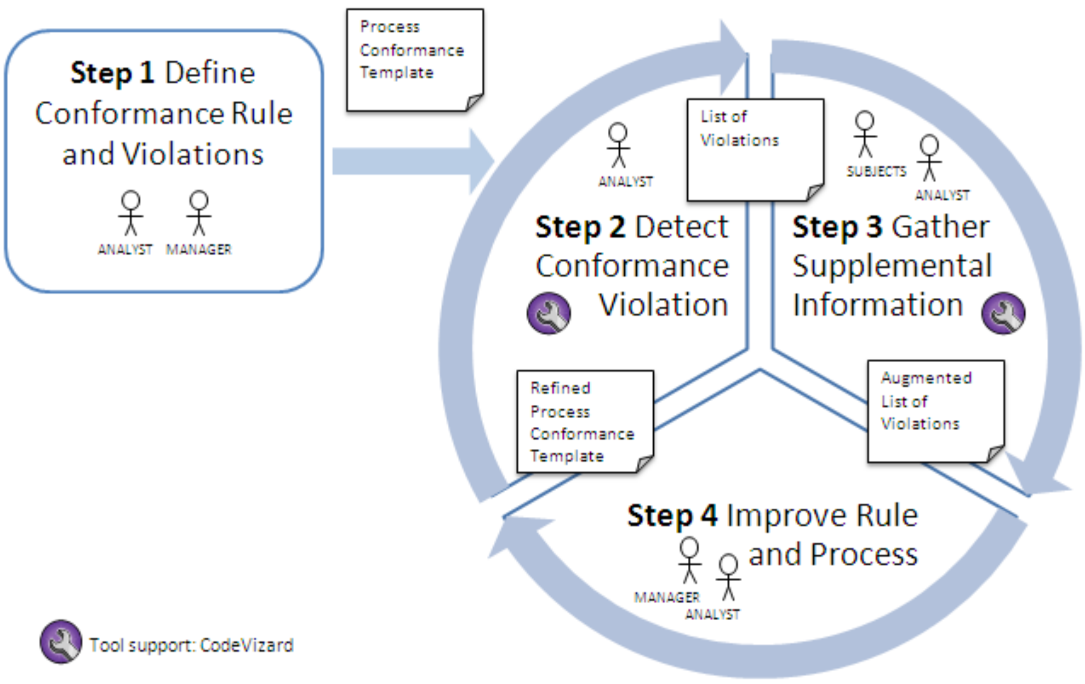

Step 1 Define Conformance Rule and Violations
ANALYST
MANAGER
Process Conformance Template
List of Violations
ANALYST
Step 2 Detect Conformance Violation
SUBJECTS
ANALYST
Step 3 Gather Supplemental Information
Refined Process Conformance Template
Augmented List of Violations
Step 4 Improve Rule and Process
MANAGER
ANALYST
Tool support: CodeVizard

Figure 10: Process Conformance Testing Approach

Primarily three different roles are important in the model:

1. *Process manager*: the person(s) interested in studying the process. In a professional environment this will be the person who tries to monitor and

improve the process (e.g. a manager, or a process coach). In research settings where the investigated process is checked for conformance this would typically be the researcher.

2. *Process enactors* (developers/subjects): this is the group of people performing the process. Typically, these roles are performed by software developers, testers and reviewers; depending on the investigated process. In empirical study settings these are typically the subjects performing the process to study.
3. *Conformance analyst*: the person(s) investigating process conformance. This role is ideally performed by an independent person, or in very small settings it can be performed by the process manager. The analyst is responsible for extracting process definitions, performing conformance analysis, investigating violations, and discussing possible improvements with the process manager.

Process Name	A unique identifier
Process Focus	Product quality characteristics or project characteristics that should be improved by the process. Examples for product characteristics are: • understandability • correctness • portability Examples for project characteristics are: • resistance against the loss of personnel • efficient training of new personnel
Process Definition	Formal or informal definition of the process
Collected Data	List of collected data sources and methods
Process Violations	Temporal Violations: Temporal patterns in the data violating the steps of the process. Quality Violations: Measures and thresholds derived from the collected data indicating low quality of process execution.

Table 2: Process Conformance Rule Template

In short, the first step (Figure 10: step 1) of the approach helps extracting the process definition, eliciting data collection methods and sources, and defining process violations. In the second step (Figure 10: step 2) these violations are then translated into machine executable algorithms that can be run on the data collected in the project or experiment. The third step (Figure 10: step 3) involves gathering insight into violations and quantifying their severity. The fourth and final step (Figure 10: step 4) aims at improving the conformance between expected process and process enactment.

4.1 Step 1: Conformance Rule Definition

The first step of the process is performed by the process manager and conformance analyst. The goal of the step is to elicit and capture the planned processes, the collected data sources, and to infer an initial set of process violations. All this data will be collected in a central document that is the output of the first step. A template for this document is given in Table 2 and I will refer to this as *Process Conformance Rule.*

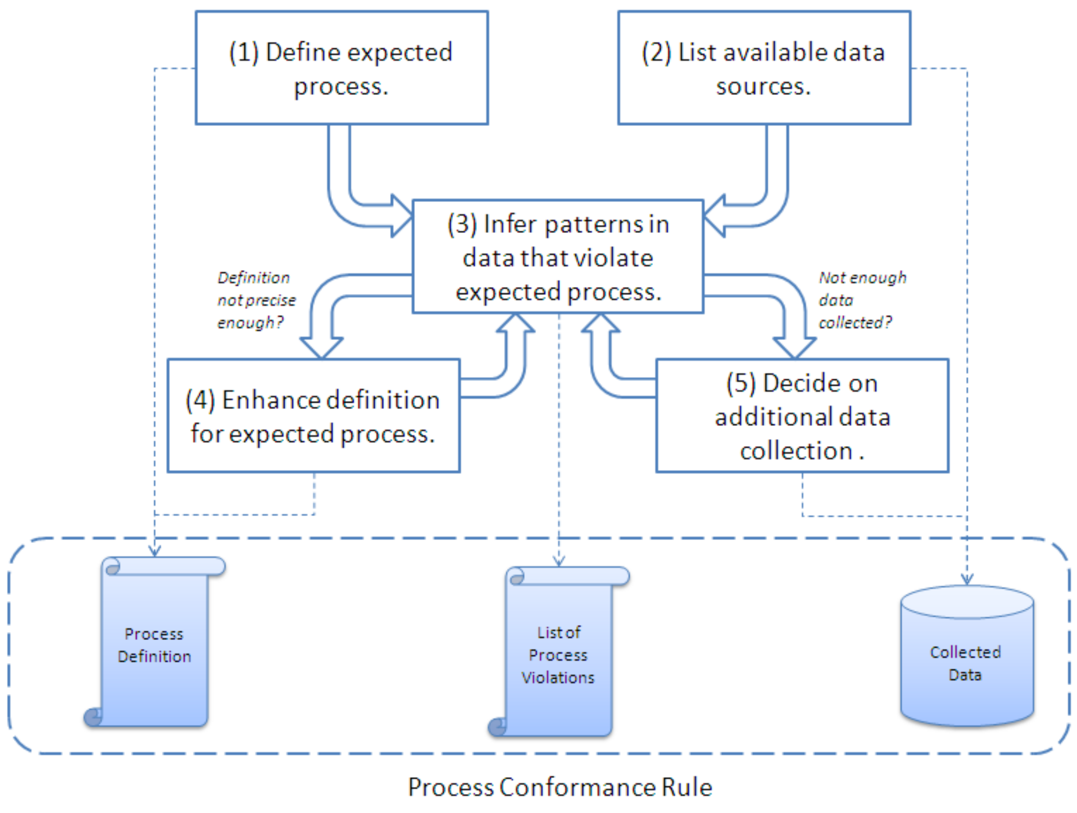

(1) Define expected process.
(2) List available data sources.
(3) Infer patterns in data that violate expected process.
Definition not precise enough?
Not enough data collected?
(4) Enhance definition for expected process.
(5) Decide on additional data collection .
Process Definition
List of Process Violations
Collected Data
Process Conformance Rule

Figure 11: Sub steps of Conformance Rule Definition

The sub activities of the step 1 given in Figure 11 start with the elicitation of the defined process (Figure 11: step 1) and data sources (Figure 11: step 2). For each process the process manager should define the expected process. Given the level of formalism it should be defined as precisely as possible (e.g. verbally, or as finite state machines).Next, the available data sources should be listed. These are typically all existing data sources related to the process, and depending on the project infrastructure might include software repositories, bug tracking, task tracking, and effort tracking systems.

The third step (Figure 11: step 3) is the central step and involves generating process violations. **Process violations are patterns in the collected data that (potentially) violate the process definition and can be thought of as failing "Test Cases" for the process.** There are two levels of process violations that are important.

First, *temporal violations* indicate that certain steps are executed in the wrong order or are not executed at all. Temporal violations aim at "*what* should be done and in which order" For example, if the process definition for Test Driven Development states that "test cases must be implemented before implementation classes" then a temporal violation would be to find a test class appearing *after* the implementation class in the collected data. To infer temporal violations one has to ask: "*Which temporal patterns in the data violate the process definition?*"

Second, a process can define certain qualitative aspects. For example, Test Driven Development not only expects test cases and implementation classes to be in a certain order, it also requires creating comprehensive and useful test cases. In other words, test cases should be of high quality. Therefore the second type of process violations is *qualitative violations*. These violations aim at *how it should be done?* Qualitative violations can be defined by asking: *"Which software metric values are indicators of poor process execution?"* Selecting the appropriate metrics, thresholds, and interpretation models might be difficult in the beginning because one might not yet understand that specific metric values indicate poor execution of the process. In this case the analyst should start with a first guess for metrics and thresholds and use the overall approach (Figure 10) to iterate and improve the selection over time. Another strategy for selecting the right set of metrics and thresholds is to use historical information (e.g. from the software repository) to infer reasonable thresholds. The latter can be seen as derivation of the process from its execution and assumes that the process was performed appropriately in the past.

In the process of creating a list of violations one might realize that the process definition does not contain an adequate amount of detail or that it is unclear or ambiguous. In these cases (Figure 11: step 4) the definition should be enhanced with these details.

Further, when creating violations one might find the collected data to be insufficient (Figure 11: step 5) to detect a certain violation. In some cases a small change in the collected data would allow defining more violations. As an example, the process analyst might discover that the changes in a code repository are *not* tagged with the names of the developers making the changes (e.g. this is the case if all developers are using the same account to access the repository). Capturing this information, however, could help to assess process specific violations (e.g. a process could require that “all developers write test cases”). Capturing this kind of data might be inexpensive since measurement procedures that are already in place only have to be slightly changed. In other cases, some processes might be very important to the organization or the study. As an example, the execution of a safety process might be exceptionally important when building a life critical system. In these cases, additional measures and data collection activities can be defined in this step.

To summarize, the procedure of the first step in Figure 11 offers a goal driven measurement approach that provides only those measures that are needed to identify potential violations.

4.1.1 Study Example

To exemplify the sub steps described in the previous section I will give a short report on the construction of one conformance rule for the third study in the professional environment. In one of the first meetings with developers (process enactors) and the technical project manager (process manager), the manager said that "*All developers should continuously contribute to test case development*". This statement describes a guideline that developers should follow and hence represents the *expected process* in Figure 10. The *collected data* sources (Figure 10: step 2) include firstly a Subversion code repository which contains information about which developers work on which part of a software system. Secondly it includes an automatic build and test system providing statistics on line coverage of test cases for each nightly build and test cycle. Test case development can be tracked in the repository very easily: all test case files are exclusively stored in a specific directory with the unique name "*tests*" and the repository provides information on which developer was working on files in this folder.

When inferring temporal violations (Figure 11: step 3) and asking "*which temporal patterns in the data violate the process definition?*" the first answer might be:

Temporal Violation:

"*Active developers are not continuously editing files in the 'tests' folder*"

Some clarification has to be given for this statement. First, one would solely be interested in "active" developers, because only "active" developers who have contributed to code development for a longer time (e.g. the last month) are supposed to write test cases. Second, the initial definition as given is not precise enough. The term "continuously" is ambiguous: it does not clarify whether developers are supposed to write test cases daily, weekly, monthly, or annually.

Therefore a second round of clarification was necessary. In another interview we first defined that "active developers" are the developers that made at least one change to source files (i.e. files ending with a specific postfix) in the last 30 days. Second, we defined that these developers must make at least one change to test code files in the same time frame. Finally, the violation rule could be rewritten to:

Temporal Violation:

"Developers who made source code changes in the last 30 days and did not make any test code changes in the same time"

The time window of 30 days can be seen as a parameter of the guideline and might change in future. The sub steps in this case helped to elicit and improve the preciseness of the guideline.

A second type of violations, qualitative violations, requires asking “*which software metric values are indicators for poor process execution?*” To detect if developers are constructing low quality test cases a test case line coverage measure can be used. This measure describes how many distinctive lines of code are executed during test case execution. As previously mentioned, when defining violations, interest is not based on whether high test case line coverage promises successful execution of the process but rather which values or behaviors indicate poor performance. In this case the process manager defined that a violation is detected if:

Qualitative Violation:

“The test case line coverage is declining over the last 30 days”

To highlight again the importance of both types of rules and how they go hand in hand one can think of two scenarios: in the first scenario, a single developer could write very good test cases for the project. This scenario would violate the temporal part only: the software quality characteristics might still benefit from the high quality test cases but the guideline is not followed and not all developers will get the same training in writing test cases. In the second scenario, all developers could collectively write poor test cases. This scenario would violate the qualitative part only: the guideline is being followed but not with the necessary quality and the quality of the product might suffer.

4.2 Step 2: Conformance Violation Detection

The next step in the process (Figure 10: step 2) is to "execute" the defined violations on the collected data in order to detect violations. Automated tools can be built to support this process. These automated tools are responsible for extracting data from the defined sources, calculating required software metrics, and applying the violations as defined in the previous step. The end result of these automated violation detection tools is list of violations. As part of this thesis such a tool (CodeVizard) was developed. CodeVizard is described in more detail in Chapter 4.6. Figure 12 illustrates the tool's data flow.

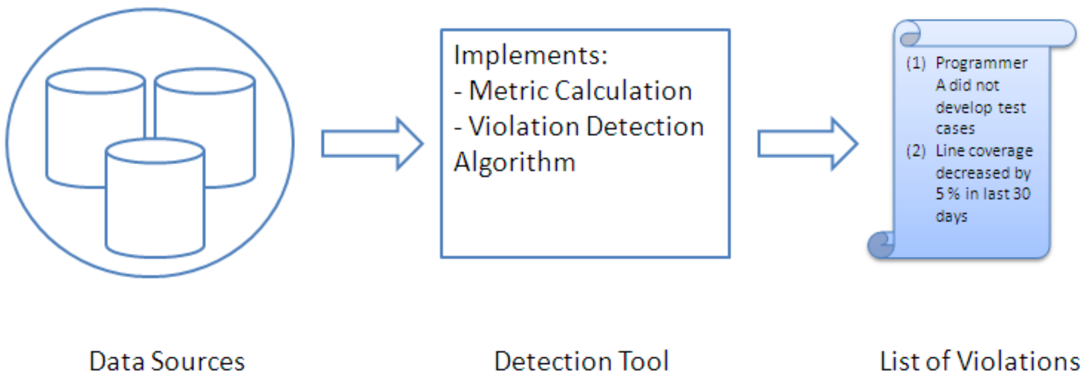

Implements:
- Metric Calculation
- Violation Detection Algorithm
(1) Programmer A did not develop test cases
(2) Line coverage decreased by 5 % in last 30 days
Data Sources
Detection Tool
List of Violations

Figure 12: Data Flow for a Violation Detection Tool

Besides the information that a violation is detected the tool should give additional information about the pattern in the data triggering the violation. For example, the name of the developers violating the process should be extracted. Or, the exact metric values that caused the violation.

The execution frequency of the tool might vary from one process to another. In most cases the manager wants to be informed about violations as soon as possible. Depending on the infrastructure of the project such tools could be integrated into the nightly build and test cycle to allow process violation detection on a day to day basis.

4.3 Step 3: Gathering supplemental information

After the list of violations has been created it is necessary to investigate them in more detail. The goal of this step is to get a better understanding of

- what violations means in the context of the process
- how severe the violations are,
- and what triggered the violations in the first place.

To collect this information the process analyst has two options. First, the analyst can look at additional, related measures. For example, if a process such as test driven development is not followed properly the analyst can look at the number of test cases generated *after* the code, or take a look at metrics such as code coverage (e.g. line coverage) that give insight into the quality of the current test cases. Further, the analyst might have a closer look at the components for which the practice was not followed to examine if only certain types of components are affected.

The second option to consider is to interview the developers causing the violations. Questions such as:

- Do you think the violation has been detected correctly?
- And if so, why did you not follow the process in this case?

will help to determine if the detection of violation has flaws (e.g. false positive warnings), and what caused the violation to happen. For example, in the case of test driven development, developers might argue that in certain cases test classes cannot be developed beforehand (e.g. for interface or skeleton classes). In other cases developers might indicate that they skipped process steps to compensate for a late project that had to be finished as fast as possible.

In either case, the additionally gathered information will help to make a more educated decision in the next step. This step focuses on changing the process or violation detection. Figure 13 illustrates the third step. The end result is a more detailed list of violations enriched by the information gathered in this step.

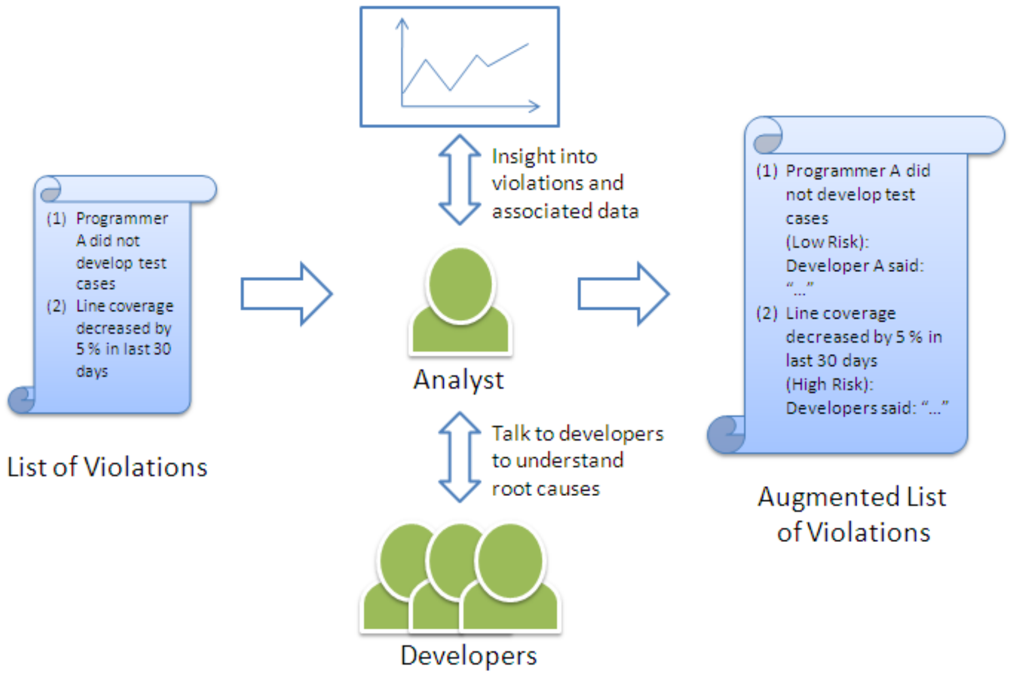

(1) Programmer A did not develop test cases
(2) Line coverage decreased by 5 % in last 30 days
List of Violations
Insight into violations and associated data
Analyst
Talk to developers to understand root causes
Developers
(1) Programmer A did not develop test cases (Low Risk): Developer A said: "..."
(2) Line coverage decreased by 5 % in last 30 days (High Risk): Developers said: "..."
Augmented List of Violations

Figure 13: Gathering additional information

4.4 Step 4: Rule and Process Improvement and Response

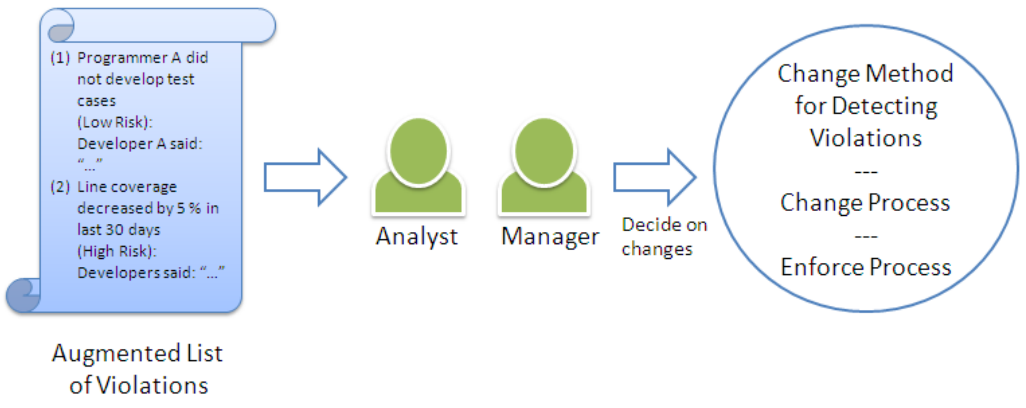
In the fourth and final step decisions have to be made about how the agreement between the defined and executed process can be improved. The manager's interest lies in minimizing violations over time in order to successfully achieve the quality goals for the developed product. Three different directions can be taken to minimize violations for the next application of the process:

1. If many of the violations are classified as false positive warnings or warnings with severity levels below the threshold of interest (i.e. violations that pose no risk to product quality), then the violation definition has to be changed. This can include modifying according thresholds (step 1), and detection algorithms

(step 2) with the goal of improving the precision (i.e., the true positive rate) of the detection method. On the other hand, if interviews provide evidence that the current definitions miss detecting important violations then changes have to be made to improve the recall of the method.

2. If it turns out that the defined process is not applicable in the current environment then modification (or tailoring) of the process can help improve conformance. This can be done by modifying the process' steps and definition. To illustrate, the manager might decide that Test Driven Development should be applied only in the beginning of a project's lifecycle. However, one has to keep in mind that a heavy modification will likely also affect the quality characteristics of the process.
3. Lastly, a manager might think about putting additional effort into enforcement of the process. This can include providing more resources to execute the process or giving penalties for not following a process. In empirical classroom studies where student subjects are graded based on artifacts they create during the study, e.g. code and documentation, the grade should really depend on their conformance to the process rather than on their performance. The subject's performance can be influenced by the process' performance, which is in most cases a dependent variable of the study. In industrial environments it can mean assigning more time and personnel for executing a process properly. For example, constantly feeding back process violations can remind developers of the importance of the process. Or, assigning a dedicated person to execute the process will help to concentrate on the specifics of the process.

After a decision has been made the process conformance template will undergo the necessary changes. Figure 14 illustrates the flow and involved actors of the fourth step.


(1) Programmer A did not develop test cases (Low Risk): Developer A said: "..."
(2) Line coverage decreased by 5 % in last 30 days (High Risk): Developers said: "..."
Augmented List of Violations
Analyst
Manager
Decide on changes
Change Method for Detecting Violations

Change Process

Enforce Process

Figure 14: Rule and Process Improvement Step

4.5 Knowledge Packaging and Transfer

After executing multiple iterations of the non-conformance process it is expected that the process conformance rules will become more and more stable. These rules now represent *transferable knowledge* that can be used as the starting point in future projects, or in future empirical studies investigating the same process.

To package knowledge effectively one has to decide which documents and artifacts to store during tailoring the rule and process. Three different levels of detail can be stored. Figure 15 shows three iterations through steps one to four.

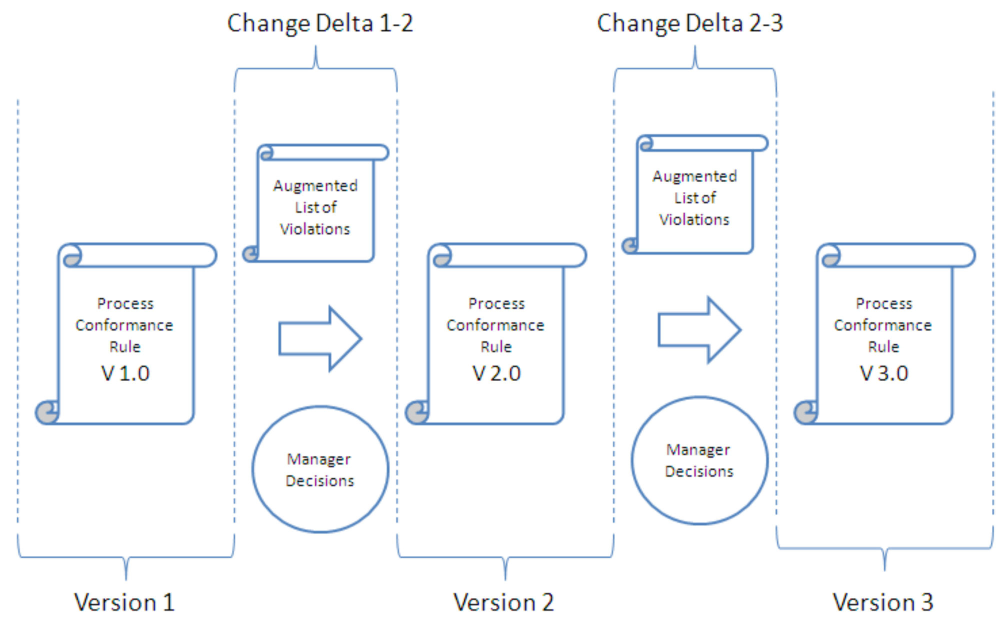

Change Delta 1-2
Change Delta 2-3
Augmented List of Violations
Augmented List of Violations
Process Conformance Rule V 1.0
Process Conformance Rule V 2.0
Process Conformance Rule V 3.0
Manager Decisions
Manager Decisions
Version 1
Version 2
Version 3

Figure 15: Knowledge Packaging and Transfer

The simplest way to capture knowledge is to only store the **latest version** of the conformance rule. This approach is useful for future studies and projects because these can make use of the optimized rule. However, information gets lost about what changes the rule has gone through. This might result in repeating some of the earlier work in the new project.

The second approach is storing the complete version history of the rule, i.e. **all versions** and differences between the versions **(change deltas)**. In a new project this strategy will give a better understanding of which modifications (of the process and detection) did not work in a previous project. However, the strategy does not include information on *why* certain changes were undertaken. The manager might not fully understand what lead to the different changes of the rule in the past.

The last and most complete approach is to **store all versions and all information that lead to the change**.

The changes in a rule are motivated primarily by two artifacts:

- the detailed list of violations received in step three of the approach and
- the managerial decision made in step four.

Storing all these artifacts will help give a better understanding of *which* changes the rule went through and *why* these changes were necessary.

4.6 Tool Support

Steps two and three require tools that support the detection of process violations and a detailed investigation of those violations. At the time of this thesis no tool was available that allowed me to encode and execute process violation checks on software development artifacts, such as code repositories. For this reason, I decided to use and extend an existing tool (CodeVizard) that was originally developed as part of a class project (Information Visualization, instructor: Prof. Dr. Ben Shneiderman) for a different purpose (Hochstein, Nakamura and Basili). CodeVizard started off as a visualization and inspection tool for Subversion repository data, and it was extended for this thesis by functionality to identify process violations in an automatic manner. The tool's functionality can be divided according to the two steps in Figure 10: violation detection and gathering additional insights.

4.6.1 CodeVizard Support for Step 2: Process Violation Detection

CodeVizard implements functionality to download and browse software repository data. In detail, it allows

- retrieving the complete historical data of a Subversion repository
- storing it in a fast and accessible relational database and
- browsing and querying it by using a Java API.

In addition, CodeVizard can compute a wide range of software metrics for Java and C# code. Thus it enables users to construct complex queries that would not be possible using Subversion's API alone, such as:

- Which programmers did modify test classes in the last 30 days?
- Which new code smells were introduced in the last week?
- Which of the test classes in the repository followed a test first order?

With these capabilities the tool supports step two of the conformance process the following way. To detect a violation based on subversion history, the conformance analyst has to add a new class (which inherits from *ProcessConformanceSensor*). The class has then to be equipped with an according detection function (it overwrites *detectViolations*) that returns a list of process conformance violations. The list of violations can then be generated as often as needed by the tool.

4.6.2 CodeVizard Support for Step 3: Gathering Additional Information

CodeVizard (Zazworka and Ackermann, CodeVizard: a tool to aid the analysis of software evolution) helps to support the third step of the approach by offering various visualizations based on the historical repository data. CodeVizard allows overlaying indications of the violations generated in the previous step on these visualizations. In many cases this helps to get a better understanding of violations. For example, it helps to understand if violations are clustered in one part of the system or if they are more uniformly distributed. Further, it gives insight into cause effect relationships, e.g., if violations are triggered by certain events such as a project deadline or a major refactoring of the software. To illustrate this in detail, a process from the second study will serve as an example.

Test Driven Development (TDD) was one of the practices checked for conformance in the classroom study. In short, the practice requires developing unit test case classes prior to implementing their corresponding code classes. CodeVizard was used to check for the following violation in the code repositories' data:

Violation 1: A new implementation class is added to the repository without a corresponding test class.

After implementing a detection algorithm that matches code and test classes and checks for test first order, CodeVizard allows the overlaying of violations in its System View. This view visualizes "life lines" for each file in the repository, indicating when the file was created, modified and deleted.

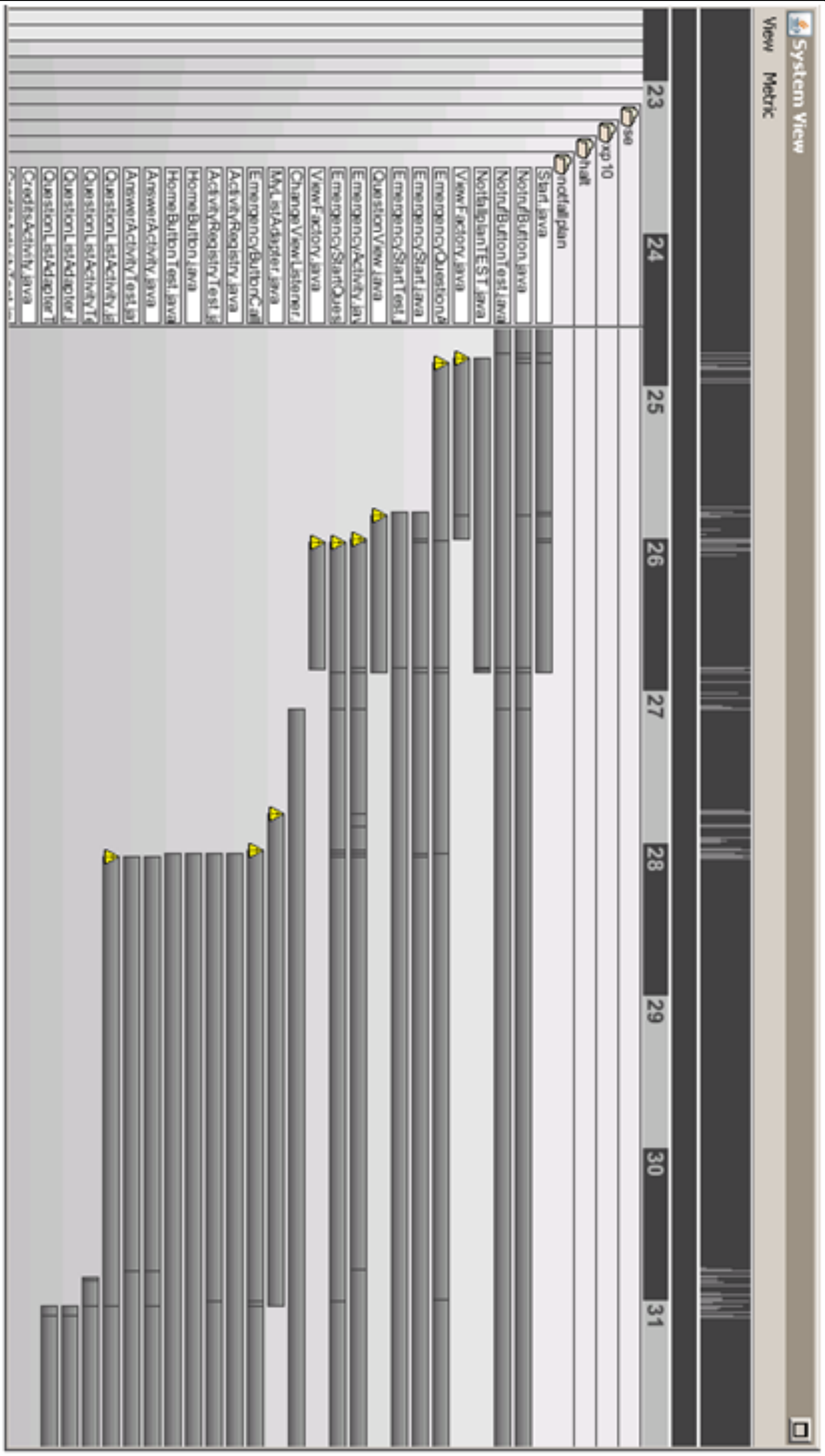


Figure 16: CodeVizard's System View (rotated) shows when software components are checked in, modified, and deleted. Yellow warning signs represent process violations against Test Driven Development: these components were added without having according unit test cases at check-in time.

Figure 16 shows one part of the software (the java package: se.xp10.halt.notfallplan) and how it developed over a time period of 5 development days (May 24, 25, 26, 27, 30). The commit activity in the repository can be read from the top bar (time ruler). Five clusters of commit activities (with a two day break for the weekend – May 28, 29) map to the five development days. The view shows further, that on each day new files were checked into the repository. The yellow warning signs indicate that a process violation was identified, i.e. the test first order was not adhered to. Following observations can be made: during the first two development days the practice was violated often. Six violations were identified for nine newly added components. Conformance to the practice improved on the later days. In the last two days only three violations were identified in this part of the system for a total of 14 new components.

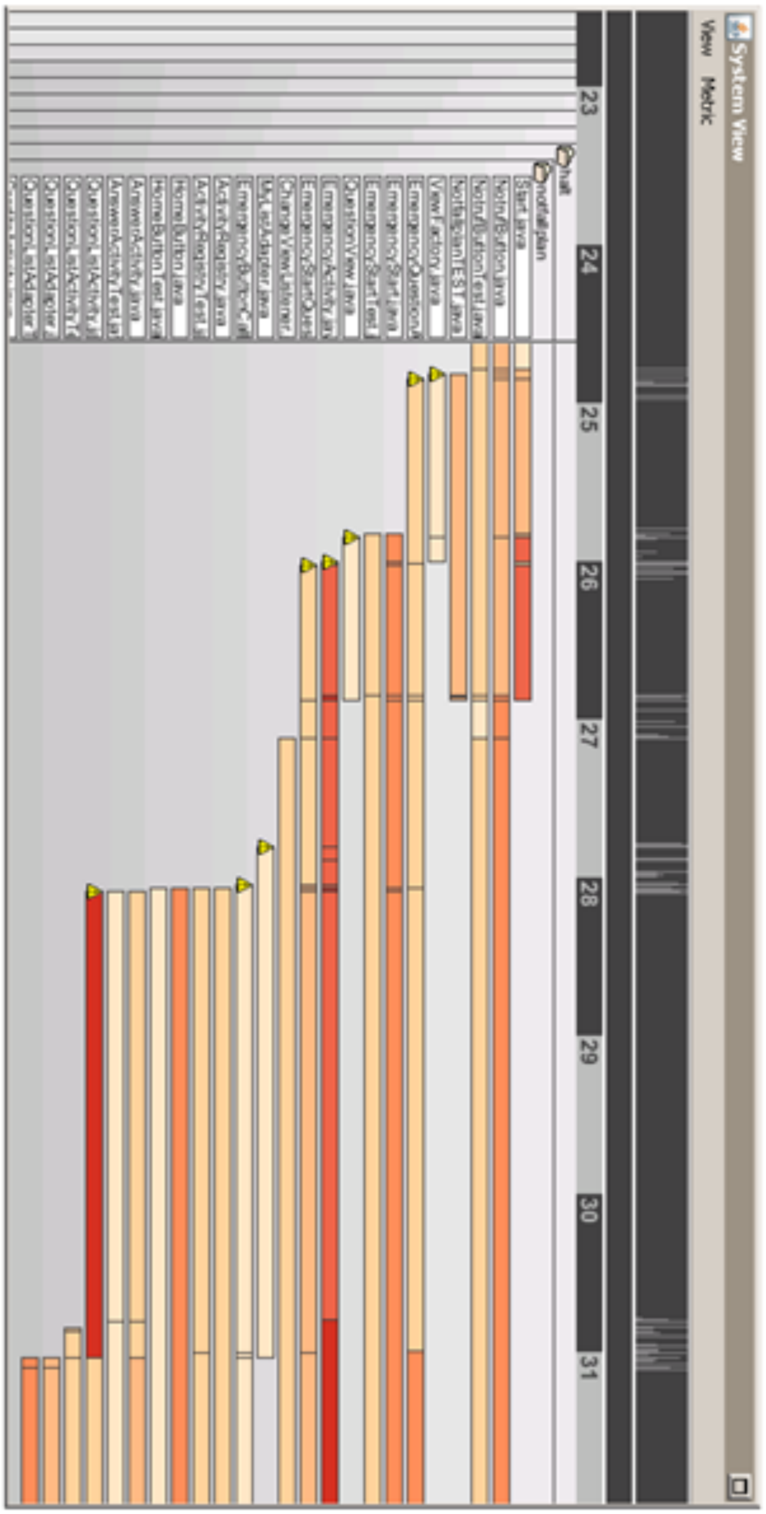


Figure 17: Metric Lines of Code (LOC). Dark red parts indicated larger components. (Figure rotated)

To gain more insight into the severity of the violations the conformance analyst might suggest that the size of the components plays an important role. For example, a very small class not being developed according to TDD might be judged as less severe than a larger one that implements a lot of functionality. To perform this analysis CodeVizard allows visualizing code metrics, e.g. a size measure such as lines of code (LOC), on top of the visualization. Figure 17 shows the same part of the system as Figure 16 with LOC shown. The Darkest red parts indicate largest components, and lightest red parts indicate smallest components. The analyst can now inspect if large classes were developed according to TDD. The picture shows that the two largest classes (*EmergencyActivity* and *QuestionListActivity*) were not developed according to TDD. This new insight can then be used when discussing violations with the process enactors and the process manager.

5 Research Questions and Study Methodology

The goal of this thesis is to investigate the research problem by building a framework and tools to **detect process violations** as well as perform a series of studies **investigating the feasibility, cost, and applicability** of the approach. The studies in professional and classroom environments aim at investigating why process enactors are violating process expectations and how these processes can be improved. Before presenting the work in detail, I will describe some restrictions that apply to the approach and the developed tool framework:

The approach introduces a general step by step framework that I claim to be applicable for most software processes. Further, the work presents one possible implementation of this framework by describing techniques and tools that can be used to enact the different steps of the framework. Specifically, the tools that have been developed during this thesis are tailored to mine data captured by **Software Configuration Management Systems (SCMs)**[9]. Validation of the work will primarily focus on processes that leave traces in SCMs. As will be shown in the studies, many software processes produce artifacts that can be found in SCMs. There are two strong arguments for choosing data from SCMs. Firstly, in practice, most medium to large size software development projects use FCMs to coordinate development efforts among a group of programmers. Secondly, by using this existing data no additional collection effort is introduced for measuring process conformance.

[9] Also known as: Version Control System, Revision Control System, or Software Repository. Popular version control systems are Subversion (http://subversion.tigris.org) and CVS (http://www.nongnu.org/cvs/).

The framework aims at finding process violations by applying test cases to the collected data. The number of identified process violations will depend on the set (e.g. number and quality) of formulated test cases. As explained earlier, **the approach will never be able to show the absence of violations**. In other words, the approach cannot **verify** that a process has been executed correctly. This property can be found in another popular method in software engineering: software testing. Writing test cases for software can help to find defects but can never show the absence of any defects in software. As with software testing, the effectiveness of the approach is dependent on the quality of the formulated test cases. Following the software testing metaphor, this approach can be described as:

Process Conformance Testing

5.1 Limitations in Measurable Processes

The proposed approach will focus on being generally applicable to a lot of software processes existing in current software engineering literature. However, some limitations do exist that prohibit the application for some classes of processes.

A) Implicitly defined and unknown processes: The approach requires an explicit process definition, as a set of steps or a guideline of what should occur. In some software environments, processes might be executed without being made explicit. For example, developers might use tacit strategies and steps to solve a particular problem, but these strategies and steps might only be unknown to the process enactor. If it is not possible to extract these steps, and to formulate a definition based upon them, then no process violations can be defined. Thus, the approach cannot be applied.

B) Insufficient data: some processes might leave only very little, or no traces and artifacts that can be checked for violations. For example, a process could require verbal communication whenever certain code parts are changed (e.g. if a commonly used code library is changed). If no data on this verbal communication can be collected then the approach will not be able to check for violations.

C) Mental processes: a last class of processes that cannot be checked for violations are processes that are completely executed in the mind of the developer. For example, a process could require a developer to to have the three most common security threads in mind when implementing a new feature to a system. In this case the approach will not be able to check for violations since it is, at the current state of science, impossible to collect data on the thought process, when not made explicit, e.g. through think aloud.

In summary, one can classify the set of measurable processes, i.e. the processes that are checkable for conformance, as the group of processes that can be defined and leave sufficient traces and artifacts behind.

5.2 Research Questions

To validate this work and to guide the studies a set of research questions and hypothesis was created. These questions and hypothesis were investigated incrementally by the different studies. While performing the research the questions were incrementally refined and transformed into testable hypothesis based upon feedback from the application of the framework. This natural, empirical learning process is reflected throughout the following description of the questions, hypothesis, and studies.

Research Question 1 (R1) – Feasibility

For the set of measureable processes, can the approach be used to find process violations using minimal intrusive methods?

The first research question addresses the feasibility of the approach. It states that the presented method can be used to identify process violations by using primarily cost effective, minimally intrusive instrumentation methods. All studies presented here will address this basic research question by simply showing that at least one violation can be identified for each of the investigated software processes. The first question builds the foundation of this work. The following questions and hypotheses build on top of this question and assume that it can be satisfied and process violations can be found.

Research Question 2 (R2) – Useful agreed upon insights

(R2A) Do the identified process violations give useful insights to the process manager and analyst and

(R2B) do they match the perceived conformance of the process enactors?

The second research question investigates whether the set of identified violations provide useful insights. Valuable insights contain information on problems with the process definition, the application of the process, the characteristics of the violations, and the measures of those violations. These insights can even contain valuable information on how to design potential changes to the process. Further, I investigate how the detected violations match the perceived conformance of the process enactors. The second part *assumes* that the process enactors are aware of their conformance (or non-conformance) to the process. Under that assumption, the number of violations and the perceived conformance of the process enactors should correlate.

Research Question 3 (R3) – Rule Improvement

Can the rules for detecting process violations be iteratively improved and tailored to the environment?

The third research question aims at tailoring the mechanisms and rules to better detect violations. It is expected that the initial models and parameters will need refinement based on feedback from their application. For example, detected process violations might turn out not to be real violations (false positives) or unimportant violations in the process context. A large number of false positives (vs. true positives) can lead to a more costly approach in practice, because every violation will have to be reevaluated by the process analyst. In the long run, the mechanisms for detecting violations should report only a few false positives and identify as many true positives as possible (i.e. have a high recall).

Research Question 4 (R4) – Process Enactment Improvement

Based on the feedback from the violations do the process enactors improve their conformance?

The fourth research question asks whether the insights generated by R2A can be used to inform the process enactors of problems and if they can use this information to improve process conformance (i.e. decrease the number of process violations). This question will help to understand whether enactors are simply forgetting to execute steps of the process (and need to be reminded), or whether they intentionally modify the process, e.g. because they see a need for tailoring it to the environment.

Research Question 5 (R5) – Rule Transfer

Can a new project in (a) the same or in (b) a different environment make use of the tailored conformance rules?

While building a rule set the conformance analyst will gain extensive knowledge about various parameters of the execution of both, the conformance process and the inspected software process. For example, the analyst will learn about the applicability of the software process, the kind of violations occurring, the frequency of process violations, and the kind of methods that successfully detect violations using a specific set of data sources. A successful approach should be able to capture that knowledge in a reusable format. For example, in a company performing the approach, a manager should be able to pick a set of rules for a new project from the collection of rules investigated in earlier projects. It is also possible to use the process rules as a starting point in another environment, and begin the tailoring process from there. The fifth research question aims at the reusability of the tailored rules in either the same environment (e.g. a different project in the same company) or in a different environment (e.g. in a different company using the same process).

Research Question 6 (R6) – Overhead Cost

What is the cost of the approach for each of the roles?

Every step of the proposed method may require additional effort from the various roles involved. For example, the manager has to look over the results of the violation insights (step 3) and make decisions about how to address these in future. The sixth research question addresses the cost overhead created through the approach. Answering this question will help a manager estimate the effort involved in applying the approach, provide the appropriate resources, and ultimately decide if process conformance analysis is worth performing.

The six research questions presented here investigate the approach from very different perspectives. The first three questions address the *feasibility and effectiveness* of the approach. Those should be answered positively to give strong support that the research presented here is a successful way to address the problem of analyzing process conformance. The fourth question deals with the human aspects of the research (are developers intentionally not following the process?) and either outcome will be of value for the body of knowledge. The last two questions supplement the first three and provide further understanding for the portability and cost effectiveness of the approach.

5.3 Research Hypothesis

To support the research questions given above, a set of testable research hypothesis was created. Research questions R1 and R2 were disassembled into hypothesis H1 and H2. Both hypotheses define a clear quality measure (precision & recall) for identifying violations. Setting a desired quality threshold for the both of them allows me to test the hypotheses and to make more precise conclusions about whether R1 and R2 are satisfied. In detail, I will provide additional evidence for feasibility (R1) and usefulness (R2) by showing that the approach finds a reasonable number of process violations (H2: recall) and valid process violations (H1: precision).

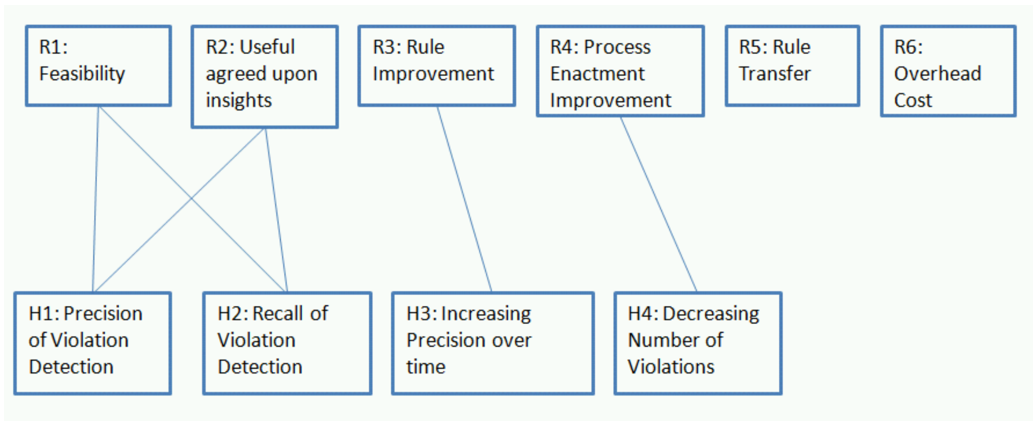

R1: Feasibility
R2: Useful agreed upon insights
R3: Rule Improvement
R4: Process Enactment Improvement
R5: Rule Transfer
R6: Overhead Cost
H1: Precision of Violation Detection
H2: Recall of Violation Detection
H3: Increasing Precision over time
H4: Decreasing Number of Violations

Figure 18: How Research Questions were refined to Hypothesis

Research Question R3 is refined into Hypothesis H3 that defines how a rule improvement can be measured and what characteristics it is supposed to show. Research Question R4 is refined in Hypothesis H4. The Hypothesis defines more precisely what an “improvement of process enactment” is by providing measures.

Hypothesis 1 – Precision of violation detection

For a given measurable process, rules can be tailored to detect process violations using the proposed methodology with a precision of greater than 50%?

The first hypothesis investigates the precision of the approach in order to support its feasibility by a precise measure. Precision is defined as the ratio of the number of true violations identified and the number of all identified violations. Setting a sufficiently high precision threshold provides evidence that the method does not provide the user (i.e. the conformance analyst) with an unfeasibly high number of false positives (i.e. identified violations that turn out not to be real violations). Setting the precision threshold to 50% means that after tailoring of the conformance rules, in worst case, only half of the violations will be a false positive warning.

A second benefit of explicitly measuring precision will be the possibility of comparing precision between two or more approaches identifying violations. Based upon my literature search, I have been unable to identify any research that reports on a precision measure for process violation detection. This is possibly due to the novelty of the approach. In this case, this work establishes a first baseline for precision in identifying process violations.

Hypothesis 2 – Recall of violation detection

For a given measurable process, rules can be tailored to detect a certain type of process violation using the proposed methodology with a recall of greater than 50%.

The second hypothesis emphasizes the recall of the approach. Recall is the measure of how many violations out of all occurring violations can be identified. Recall will decrease if the models for detection fail to identify real violations. As exemplified in Chapter 2.1, the approach is limited to the number of violations it can find by the amount of measurements that can be taken in reality. However, once the set of possible measurements and different types of process violations is defined one wants to detect *most* violations that can be inferred from that data set. Therefore, the recall in this hypothesis is meant to be the recall for a specific type of violation that is defined in the conformance template. A second issue with measuring recall based on collected data is that the number of all (real) violations cannot always be determined exactly. In most cases, data sets will be too large and therefore too costly to be searched for all violations in a manual way. In the following studies, I will limit the costly investigation of recall in the following way: statistical samples of the data will be investigated (by the conformance analyst and process enactors) to make a judgment about all true positive violations (for one specific type of violation) in the sample. This "ground truth" judgment will then be compared to the automatically identified violation set. I will estimate the true recall by calculating recall based on the comparison of the two sample sets.

The above described limitation and method of investigating the true recall has been used in the field of software engineering. For example, whenever defect identification methods and techniques, such as structural testing and code reviews, are studied (with respect to the number of defects they can identify) researchers estimate the number of all defects in the software (e.g. by inserting some defects and using those found compared to those not found as a basis for estimating the percent of defects actually left in the system: (Knight and Ammann)). However, in most cases the true number is unknown.

Hypothesis 3 – Increasing precision over time

The precision of identifying process violations increases *monotonically* over time using the methodology.

The third research hypothesis investigates the effectiveness of the four step iterative model in detail. If the iterative model helps tailor the rule set effectively for an environment (i.e. a software development project) then the precision metric should increase over time. Violations should be detected more effectively up to a point where they are stable, where no more improvement can be made.

Hypothesis 4 – Decreasing the number of Violations

After developers are informed about process violations the number of violations per analysis period will decrease.

Hypothesis 4 states that the number of true positive identified process violations will decrease once developers are informed about these violations. In other words, I investigate if feedback on non-conformance will have a positive impact on process conformance.

5.4 Study Methodology

A set of scientific methods can be used to test the research questions and hypothesis. Typically, these methods define *how* studies can be designed to provide evidence and how data analysis should be conducted. Potential study designs range from pre-experimental designs, quasi-experimental designs, case studies to controlled experiments. The study designs differ typically in the amount of artificiality in the study setting and control one has over the variables of the study. On one end of the spectrum, pre-experimental, quasi-experimental designs and case studies are usually conducted in vivo, i.e., during actual practice (e.g. at a company with professionals doing their normal activities). Randomization is not possible and the design provides little or no control over the variables. On the other end of the spectrum, controlled experiments are likely to be conducted in vitro, i.e., in an artificial/laboratory environments but with a higher degree of control of the variables. Quasi-experimental designs are a tradeoff between both ends of the spectrum and introduce some control of variables in a realistic setting. Data analysis methods include quantitative and qualitative techniques, which define how to collect and analyze data.

For this thesis I used pre-experimental and quasi-experimental designs, as well as quantitative and qualitative analysis methods to answer, support (or reject) the hypothesis and questions. The rational not to conduct controlled experiments is the following:

One of the main claims of the proposed approach is that it can be applied for a whole range of software processes as applied in practice. The nature of a controlled experiment would have required building an experiment "around" a designed process for the purpose of the study. This would have been subject to the criticism of bias in the selection of the process, i.e., towards choosing a process that "would work" with the approach. Further, study subjects would have executed a process for the purpose of the study only, which would also be subject to the criticism of bias as the subjects would have been focused on process conformance, rather than just applying the process to achieve the project goals, i.e., the subjects would have been strongly biased towards following the process since they would have seen it as their primary goal. In reality however, process enactors will more likely see the process as a tool to reach software development goals (e.g. finish a product within time and cost). The change of developer behavior and the selection of the process would have been a threat to internal and external validity in a controlled setting. Therefore, I considered a controlled experiment as not being the appropriate approach. It should be noted that one of the goals of the approach of doing experiential validation is to provide feedback on the method so it can be improved with each application. Therefore the chosen studies are rather exploratory in nature, opposed to being confirmatory.

To test the hypotheses and answer the research questions, I have conducted four studies. All four studies follow pre-experimental designs or quasi-experimental designs (as opposed to controlled experiment designs). Some characteristics of the studies are given below:

The studies were conducted *in vivo*, that means “in the field under normal conditions”. In this case the studied method was used to investigate conformance in realistically sized industrial projects and realistic classroom experiments. I consider the classroom studies as in vivo because one of the goals is to identify conformance in controlled experimental settings. The primary purpose of the classroom experiments was not to investigate process conformance, but to teach and measure the effectiveness of a programming paradigm (XP programming) in the classroom. The conformance measurement was “piggy backed onto this study”.

Because of the nature of the studies, e.g., limited numbers of subjects, the unit of analysis was not the subjects but the rate at which non-conformance violations occurred. I use scientifically accepted measures and statistics, such as precision and recall, to provide evidence for and against the earlier presented hypothesis using these statistics.

The pre-experimental and quasi-experimental study designs can be outlined as following for the four studies:

5.4.1 Study 1: Feasibility Study: Pre-experimental design

The goal of the first study was to test if it is feasible to identify process violations through the inspection of implicitly collected data (i.e. existing data). To do so, one industrial project was selected and a subset of the proposed steps was applied (steps 1, 2, and 3). The design can be described roughly as a "*one shot case study*". The scientific value of such a study might be low, due to the absence of control. However, it was used to evaluate if it was sensible to continue this stream of research. The analysis methods used in this study were of a quantitative nature since it was conducted a posteriori (after the fact) and developers were not available for further qualitative analysis.

5.4.2 Studies 2 and 4 (Classroom I and II): Multiple Group Equivalent Time Samples Designs

The design of the two classroom studies can be best described as a Multiple Group Equivalent Time Samples Design. This is a quasi—experimental design. In each study I observed two groups performing XP development. After each development day (equivalent time samples) conformance analysis was done and a report of violations was presented to the developers. Analysis methods included both, quantitative and qualitative methods (e.g. questionnaires), to provide insight into how often and why developers strayed from the XP processes.

Multiple Group Equivalent Time Samples Design:

R O_{A1} X_{A1} O_{A2} X_{A2} O_{A3} X_{A3} O_{A4} X_{A4} O_{A5} (XP Team A)

R O_{B1} X_{B1} O_{B2} X_{B2} O_{B3} X_{B3} O_{B4} X_{B4} O_{B5} (XP Team B)

Multiple groups (XP teams A and B) are shown in two lines.

O_{ji} denotes an observation in the experiment (i.e. detection of conformance violations during one development iteration)

X_{ji} denotes a treatment or intervention (in this case conformance violations were reported to developers or the manager)

R denotes randomization: students were assigned randomly to the two development groups

5.4.3 Study 3 (Long term study): Multiple Group Time Series Designs

The long term case study was conducted with professional developers. Several projects (multiple groups) and processes were investigated for violations. This is a quasi—experimental design. Reports of the violations were presented to the developers at different times depending on the project and process (therefore the time samples are not equivalent). Analysis methods consisted of a mix of quantitative and qualitative (i.e. interviews and questionnaires) analysis.

Multiple Group Time Series Design:

$$O_{A1}\ O_{A2}\ O_{A3}\ X_{A1}\ O_{A4}\ O_{A5}\ O_{A6}\ \text{(Project A)}$$

$$O_{B1}\ O_{B2}\ O_{B3}\ O_{B4}\ X_{B2}\ O_{B5}\ O_{B6}\ \text{(Project B)}$$

O_{ji} denotes an observation in the experiment (i.e. detection of conformance violations during one development iteration)

X_{ji} denotes a treatment or intervention (in this case conformance violations were reported to developers or the manager)

5.5 Contribution of Proposed Work

The proposed work contributes in several ways to the scientific body of knowledge. There are four main contributions given below:

Contribution 1: A step by step approach to define, detect, and investigate process violations as a measure of process non-conformance issues is presented. This approach uses a combination of techniques, such as interviews, information visualization and data mining.

Contribution 2: The work investigates whether identifying process violations is of value, i.e., if it offers some insights into how developers perceive violations (e.g. if developers are aware of them), and how managers can use them to earlier detect problems in a project.

Contribution 3: The work gives insight into (1) the kind of violations that actually appear in the set of software development processes investigated in the studies and (2) how well developers can follow a specific process in the given environment. At this point in time, the following processes have been investigated (see also Chapter 6):

1. Adherence to a design and development plan (Waterfall/professional)
2. Adherence to a Test and Review process (Waterfall/professional)
3. Test Driven Development (XP/classroom)
4. Continuous Refactoring (XP/classroom)
5. Pair Switching (XP/classroom)
6. Collaborative test case development (Agile/professional)
7. Adherence to architecture conformance (Agile/professional)
8. Continuous Refactoring (Agile/professional)
9. Adherence to communication processes in distributed development (XP/classroom)

Contribution 4: Last, the work will result in a reusable set of process templates and detection algorithms that can be used as a basis for other projects and studies.

6 The Four Studies

To validate the different research hypotheses a series of four studies has been conducted in classroom and professional environments. I will refer to the studies the following way:

Study *FEASIBILITY*: pre-experimental feasibility study applied on data from a large scale project with professionals
Study *CROOM1*: the first classroom study following a quasi-experimental design investigating XP practices
Study *CROOM2*: the second classroom study following a quasi-experimental design investigating distributed XP practice.
Study *PROF*: the long term study with professionals in a realistic company setting

As described in the previous chapter, the studies follow different experimental designs.

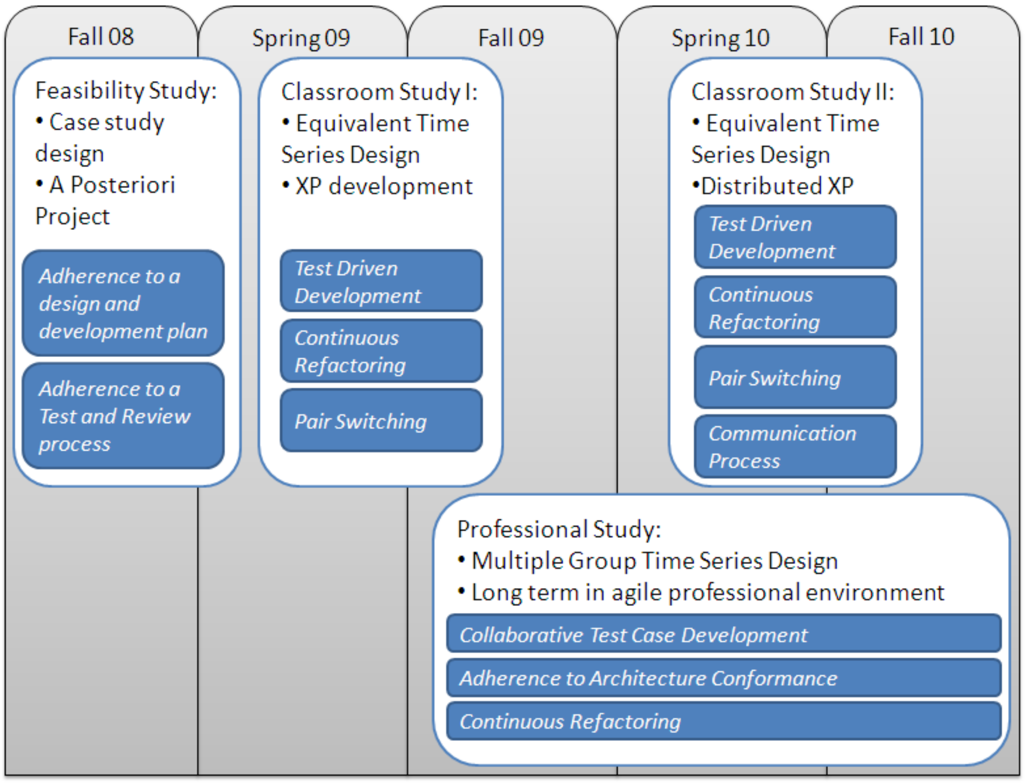


Figure 19: Plan of Studies Chronological Overview

6.1 *Overview of Studies*

A chronological overview of studies is given in Figure 19. Three of the four studies have been completed. The study with professionals (*PROF*) is an ongoing effort at a customer of the Fraunhofer Center Maryland. This study will continue to run even after completion of this thesis. The studies build evidence for different sets of hypotheses; later studies investigate the more complicated ones. A comparison of key facts about the studies is listed in Table 3.

Study	*FAESIBILITY*	*PROF*	*CROOM1*	*CROOM2*
Date of Study	March 2008	Sept 2009 – Dec 2010	May 2009 (5 development days)	May 2010 (5 development days)
Publications	ESEM 2009: Technical Paper (Acceptance Rate: 40%) (Zazworka, Basili and Shull, Tool Supported Detection and Judgement of Nonconformance in Process Execution)	ESEM 2010: Technical Paper (Acceptance Rate: 29%) (Schumacher, Zazworka and Shull)	ESEM 2010: Technical Paper (Acceptance Rate: 29%) ***Best Paper Award*** (Zazworka, Stapel and Knauss)	
Environment	Large Company in Aerospace Domain	Mid Sized Web Development Company in Washington D.C.	XP Course 2009 at University of Hanover	XP Course 2010 at University of Hanover
Number of investigated processes	2	3	3	4
Project Size	1 large project: 83kLOC	2 medium projects: each 15kLOC	2 small projects: each 2.3kLOC	2 small projects: each 5.2kLOC
Developers	7	4	14	15
Project Duration	3 years	1-2 years	3 months (XP course): 4 development days	3 months (XP course): 5 development days
Main Characteristics	• A posteriori analysis • Feasibility study • Limited execution of steps	• Long term study • Mid-size web development projects • Developer interaction • Integrated into CMMI and Agile Lifecycle	• Classroom study • Empirical investigation of XP practices • Timely reports of non-conformance each development day	• Similar to *CS1* but with one distributed team • One new distributed practice

Table 3: Study Characteristics

Chapter 6 is arranged the following way: firstly, in Section 6.2 I will give some background on the processes that were applied by the subjects, i.e. the process enactors. Afterwards, I will present in four subsections (Sections 6.3 - 6.6) the study environments, study designs, conformance rules, and finally the data that was collected in each of the four studies. The next chapter (Chapter 7) will describe how the data answers the research questions and hypotheses.

6.2 Investigated Processes

A variety of software processes and practices were investigated in the studies, ranging from formally defined ones to practices that are given in natural language. Table 4 summarizes the processes and studies. Some processes, e.g. *Completion Process*, were investigated in only one study. For other processes, e.g. *Continuous Refactoring*, more data could be collected through application in multiple studies and study environments.

Process	Short Name	Process Aim	*FEASIBILITY*	*PROF*	*CROOM 1*	*CROOM 2*
Adherence to a development plan in waterfall development	*Completion Process (ComP)*	Project progress traceability	X			
Adherence to a test and review plan	*Correctness Process (CorP)*		X			
Continuous Refactoring	*Continuous Refactoring (CR)*	Avoidance of degrading code design		X	X	X
Pair Switching	*Pair Switching (PS)*	Improved collective code ownership			X	X
Test Driven Development	*TDD*	Improve Program Correctness			X	X
Collective Test Case Development	*CTCD*	Developer training and improved program correctness		X		
Architecture Conformance	*AC*	Improved maintainability		X		
Communication Practice: Broadcast of Story Card and Name	*Communication Practice*	Communication, increased productivity, shared knowledge				X

Table 4: Summary of Processes and Studies

6.3 *FEASIBILITY: Feasibility Study*

The initial feasibility study was performed on data captured during a software development project from an industrial software application in the aerospace domain. On the one hand, the study demonstrates that there is a sufficient amount of non-conformance in the execution of processes in real world examples. On the other hand, it shows that the approach is applicable and powerful enough to uncover real process violations in such projects. However, since the process violation detection was performed after the project's lifetime it was not possible to influence the process executions, such as changing processes (step four of the approach) and reevaluating the impact of the changes (iterative characteristic of the approach).

6.3.1 Study Environment

The development time of the target application was two years and split into four phases. Seven programmers worked on developing the software following the waterfall model and were required to deliver a running and tested version at the end of each phase. The final size of the application was about 83,000 lines of code distributed over nearly 2000 components (i.e. Java classes). The following analysis focuses on the first two phases (version 1 and 2) of the project.

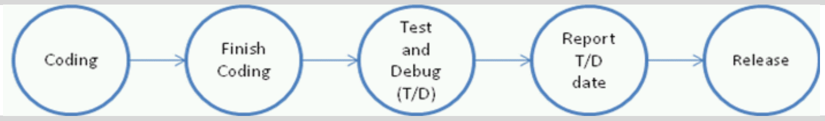
Process Name	***Correctness Process***
Process Focus	Process improves correctness on unit / class level.
Process Description	 Coding Finish Coding Test and Debug (T/D) Report T/D date Release
Collected Data	Automatically (existing): • Code repository Manually: • End of unit testing • End of code review
Process Violations	**Temporal**: **V1:** Modifications to components after finished testing and review date, detected by using change data from repository and reported finish dates. **Qualitative**: none

Table 5: Process Conformance Rule for Correctness Process

6.3.2 Step 1: Conformance Rules

Two processes were inspected that were planned to track the project's progress (*Completion Process*) and to increase correctness of the code (*Correctness Process*). The *Completion Process* defined a time frame for each component that described the start time and end time of development. The process definition was given in form of a list (i.e. an Excel spreadsheet). The *Correctness Process* included a plan for testing (i.e. unit testing) and code review activities for each component at the end of the component's development time.

Automatically collected data was gathered through the version control system (i.e. CVS). Programmers had to fill in weekly information about when code review and testing activities (including bug fixing) were completed. Both of these mechanisms were part of the normal work environment at this organization.

Conformance rules for both processes were created. Table 5 shows the conformance rule for the *Correctness Process*. Table 6 md graddoes the same for the *Completion Process*.

Process Name	*Completion Process*
Process Focus	Process improves traceability and predictability of project progress.
Process Description	Each developed component, given by its expected java class name, should be developed between its **start coding** and **end coding** date. A list (Excel spreadsheet) defines these dates.
Collected Data	Automatically (implicitly): • Code repository Excel spreadsheet defining start and end coding dates
Process Violations	Temporal: Various items can be detected. At a specific time t each class from the plan is in one of the three states: • **before start** of coding • **in coding** (after start of coding, before end of coding) • **after end** of coding Further each component in the repository can be assigned one of the two states: • **existent** in the repository • **nonexistent** in the repository Process violations are the following combinations: **V1:** {before start, existent}: a class that is **too early** in the repository **V2:** {in coding, nonexistent}: a class that should be in coding phase but cannot be found in the repository: **slightly delayed** **V3:** {after end, nonexistent}: a class that should be finished with coding and cannot be found in the repository: **delayed** **V4:** {undefined, existent}: a class in the repository that cannot be found in the plan: **unexpected** Semantic: none

Table 6: Process Conformance Rule for Correctness Process

6.3.3 Step 2: Process Violation Detection

The algorithms implementing the violation detection for these process violations were implemented into CodeVizard. For demonstration, the number of detected violations for both processes is plotted in Figure 20 and Figure 21.

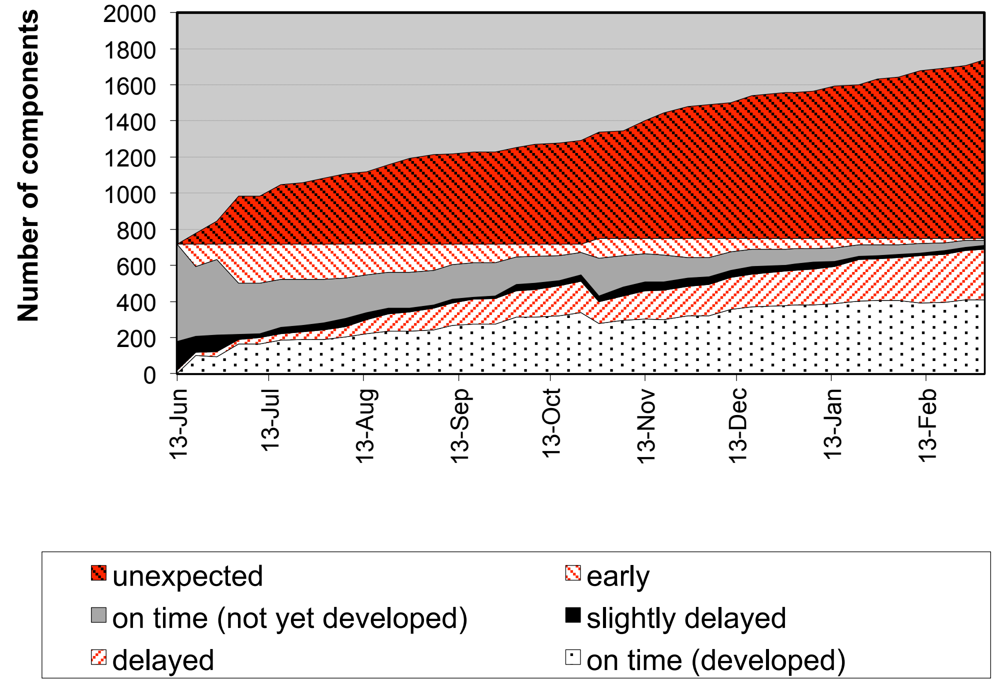


Figure 20: Detected Violations for *Completion Process*

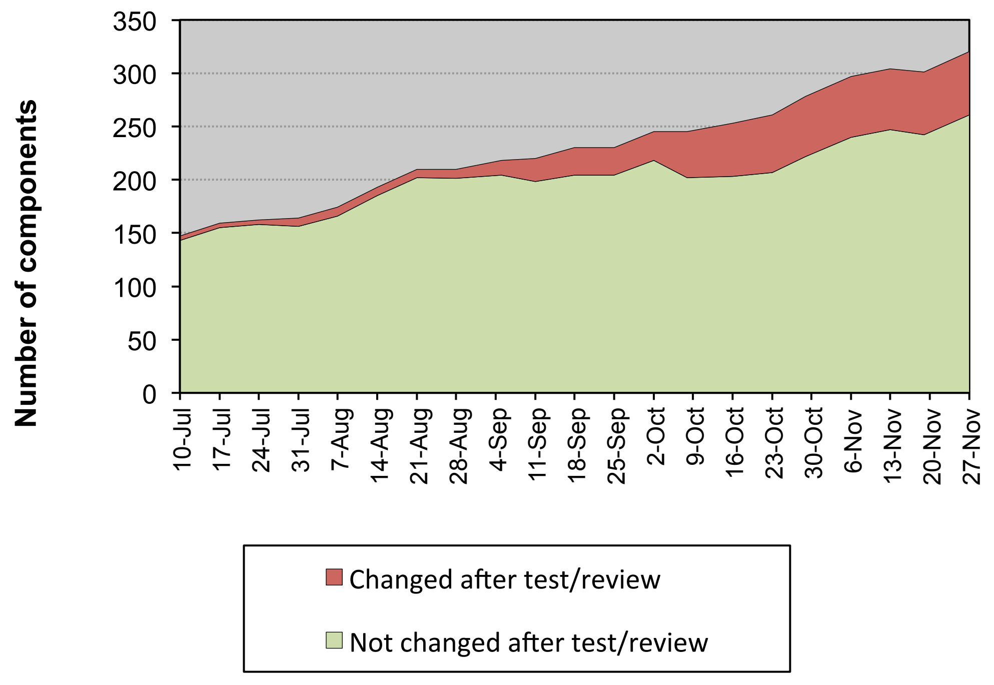


Figure 21: Detected Violations for *Correctness Process*

Both graphs show an increasing number of process violations over time during the first months of development. In the case of *Completion Process* (Figure 20) the number of delayed classes (violation V3) increases from the beginning on. Further, the amount of unexpected classes (i.e. classes not defined in the plan: violation V4) is very high. At any time, the repository contains more unexpected classes than actually planned and developed classes. The number of classes being developed too early is high in the beginning and then decreases; this is logical since these classes fall into the "on time" category once their planned start date is reached.

As for the *Correctness Process* (Figure 21), the number of modified components after testing/review increases steadily from September. In the end of the plotted time period, 50 classes are marked to have been modified after the testing phase.

To provide better insight into the severity of the detected items it is necessary to investigate the data closer. This is done on a recurring basis, e.g. once a week. As example for this work, I have selected two fixed dates for demonstration, as shown in the next subsection.

6.3.4 Step 3: Gathering Additional Information

To get a better understanding about the large number of violations in this project, CodeVizard was used to inspect the violations in detail. In particular, I used CodeVizard's System View to gain insight into when and where violations occurred.

My initial hypothesis, by looking at Figure 20, was that the developers were falling more and more behind plan (based on the increase in the number of delayed components) and that the high number of unexpected files can be explained by the import of external libraries that were not defined in the plan.

However, the visual analysis of the four categories through CodeVizard showed that all the process violations were distributed uniformly over the number of developers and the parts (i.e. Java packages and classes) of the software system. Further, components marked as unexpected were modified heavily and could be found in almost any of the packages. An example package is visualized in Figure 22. It shows two sudden increases (September and October) of unexpected components developed by two programmers.

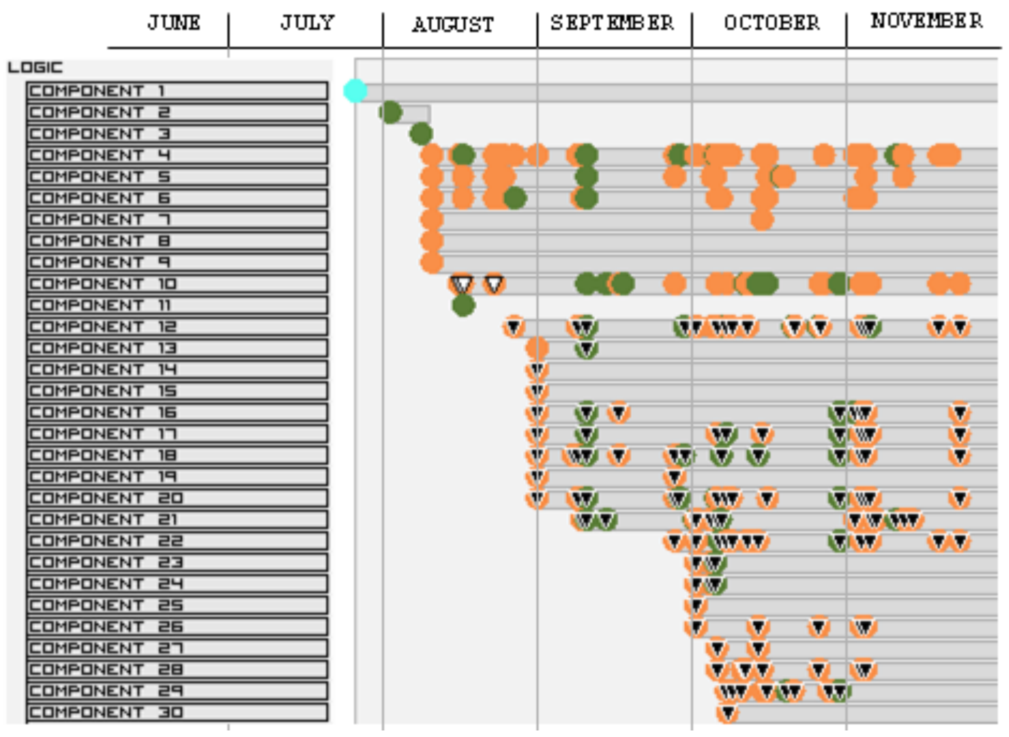


Figure 22: One package (LOGIC) with 30 java source files. The yellow (light grey) and green (dark grey) authors mainly worked on these files. Each circle represents one commit to the repository. A black triangle indicates that the component is unexpected (not defined in the project plan). A white triangle shows that the component is too early in the repository.

At this time I was able to interview a project participant with our results. The participant explained that the static design of the application (developed in the design phase down to class level) was changed by the programmers during the development. In many cases, bigger classes were broken down into multiple smaller classes. This can explain the amount of delayed classes (big classes) and unexpected classes (smaller classes). The developers did not report those modifications, because the process did not implement this step. Hence, the components in the project plan were never updated with this information.

One might now ask which risks this divergence between the project plan and the actual development implies for the process goal. Remembering the focus of the process (traceability and predictability of project progress), one can argue that the plan can no longer provide a precise trace and prediction of the projects progress, because it differs significantly from the system developed in reality.

A second question a project manager would certainly be interested in is: will my project be delayed? This question cannot be answered directly. The developers claim to have implemented the necessary functionality into the split classes of the system. The project plan however, is not defined in terms of functionality – it is therefore impossible to check if the functionality in the unexpected classes sums up to the functionalities in the delayed classes.

It is worthwhile mentioning that, in reality, the first phase of the project was delayed by two weeks.

For the second process (*Correctness Process),* Figure 21 indicates that the number of components modified after testing/review increases significantly around October 8th. For each of the 24 violating source components, CodeVizard can be used to gain more detailed insights. To demonstrate, I used CodeVizard's CodeView (see Section 4.6.2) to distinguish six kinds of changes. I assigned two different severity levels based on the impact the change can have on program correctness (see also Table 7):

- changing documentation (**d**): low severity (updating code documentation does not require one to update and rerun any test cases)
- code formatting, e.g. changing code indent, deleting blank lines (no syntactical change) (**cf**) - low severity
- code rewriting (syntax change, but no semantic change) (**cr**) - low severity
- add/delete of debugging (e.g. system.out.print) statements (**so**) - low severity

- semantic code changes (**sc**) - high severity
- addition of new functionality (**af**) - high severity
- deletion of functionality (**df**) - high severity

The last three categories pose a threat to correctness since these kinds of changes require retesting and re-reviewing the component. After identifying violations with a high severity, the manager might be interested in the reasons for these late modifications. Therefore, the analysis keeps track of the names of the programmers performing the changes to guide interview sessions.

In cases where a complete manual inspection of all affected files is too costly, the analyst might either want to draw a random sample from the set of affected components in order to estimate the total number of high risk items, or first focus on the ones that promise to pose a high risk. In later case, the relative code churn measured after the testing/review date can be a helpful guide to these components. Code churn (Nagappan and Ball) is a measure that describes how much of a component's code was changed over time. Table 7 shows an excerpt of the risk judgment for October 8th.

Component	d	c f	cr	so	**sc**	**af**	**df**	Churn (%)
Comp_a	+	+	+	+	+			30
Comp_b	+	+		+	+	+	+	698
Comp_c	+							4
Comp_d	+		+					2
Comp_e	+							2
Comp_f	+	+				+		12
Comp_g	+				+	+		3
Comp_h					+	+		35

Table 7: Gathering additional insights for a random selection of process violations. For each component the types of changes are listed ("+" meaning the type is present). Five components (a,b,f,g,h) include change types with high severity. The churn measure shows how many lines of code were changed relative to the test/review date, e.g. 50% means that half of the lines were changed.

The analysis showed that more than half of the process violations included dangerous changes. The risk that the correctness process will not achieve its optimal performance is certainly elevated by these items.

6.3.5 Step 4: Process / Rule Improvement

Since the investigated project was not observed at development time (but a-posteriori) I did not have the chance to further investigate the research questions and hypotheses that require giving advice directly to the manager and process enactors. However, if I would have the chance then I would advise them to tailor the *Completion Process* to account for design changes during the development time of the project. Further, I would advise them to retest and review the detected and analyzed classes that pose a risk to correctness in later states of the development phase.

As for rule improvements, one may think of further optimizing the detection algorithms for the correctness process: the detection algorithm could eliminate more false positives by automatically checking for the type of changes in some cases (documentation changes (d), code formatting changes (cf), and debugging changes (so)).

6.3.6 Investigated Research Questions and Conclusion

In light of the research questions given earlier, the first study showed that the step by step approach was able to define violations for two processes and that tools could be built to extract violations from the collected data. This is evidence for the first research question: *"Can the approach be used to find process violations using minimal intrusive methods?"* Further, the study showed how assessment and investigation can be performed by using visualization techniques and interviewing developers. The found violations were classified as risk for the project's success and therefore evidence that: *"The found process violations give useful insights and match the perceived conformance of the developer." (Research Question 2)*

6.4 CROOM1: First Classroom Study

The second study took place in a classroom environment and followed a quasi-experimental design. More specifically, it followed a two group - four equivalent time samples design. The two groups were formed by students learning new software development practices as part of their studies at the University of Hannover in

Germany. The four equivalent time samples correspond to the four development iterations. The iterations took one day each. After each of the iterations, conformance analysis was performed and feedback was given to the process managers and students.

The first study mainly served as feasibility study to provide evidence for research questions one and two, whereas the second study investigates the full range of research questions presented in Chapter 5, except for Research Question 5 (rule transfer).

To address the research questions I chose to investigate three popular XP practices in a classroom setting:

1. Test Driven Development (TDD)
2. Continuous Refactoring (CR)
3. Pair Switching (PS)

Research Question 2 (useful agreed upon insights) was tested by a comparison of the perceived conformance (of the students) versus the measured one. For Research Question 3 (rule improvement) I show how detection is tailored towards the classroom environment. Lastly, Research Question 4 was evaluated using one instance where developers were actively advised to improve conformance to a practice during project runtime. Data for Research Question 6 (cost of the approach) is presented in a later chapter (Chapter 7.1.7).

6.4.1 Study Design

The study took place as part of an XP class taught at the Leibniz Universität Hannover, Germany (LUH). Conformance analysis was performed remotely at the University of Maryland, USA (UMD). In the first theoretical part of the course student developers received lectures about agile development and XP basics. All but one of the XP practices were taught in this lecture on a theoretical level. The XP practice Test Driven Development was taught separately in a practical exercise. The second part of the course was a five day (eight hours per day) development project where the developers worked on building a software product in an - as close as possible - industrial environment. On the first day, the two customers introduced their visions, an initial technical spike was conducted, and the XP specific story cards were created. The following 4 days were development iterations, each with a duration of one day. The 14 developers, 11 graduate students and 3 undergraduate students, were randomly split into two groups with seven developers each. Both groups developed a different product; in the following I will refer to them as team *Zeit* and team *KlaRa* in accordance with the names of the two products. The implementation language was Java. The course was not the first of its kind. It was already in its 5th iteration. More details about the course design can be found in (Stapel, Lübke and Knauss).

Process Name	**Test Driven Development**
Process Focus	Improved correctness.
Process Definition	For each component (i.e. Java class) developers are supposed to create a JUnit test class (collection of test cases) prior to the development of the component.

Collected Data	Subversion code history. Developers are advised to use the following file naming scheme for implementation and test classes:
	Implementation class:
	`SomeName.java`
	Test class:
	`SomeNameTest.java`

Process Violations	**Temporal**: (1) Implementation classes (but not interface classes) without test classes. Violation detection: Implementation class is checked into the Subversion repository before its according test class. **Qualitative:** (1) The line coverage of the test cases is below 70% (2) The branch coverage of the test cases is below 70%

Table 8: Process Conformance Rule for Test-Driven Development

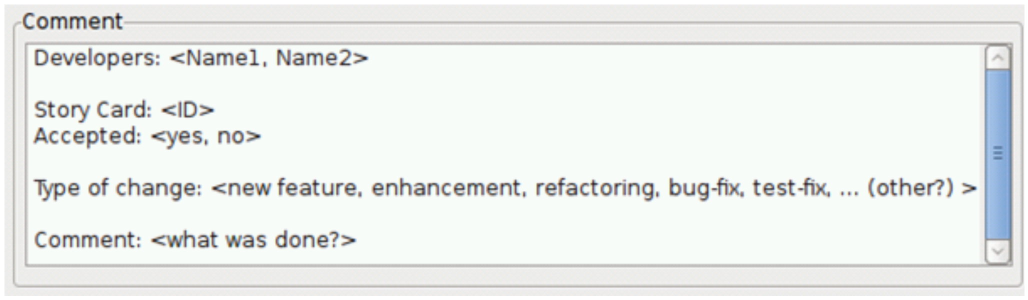
Process Name	**Continuous Refactoring**
Process Focus	Improved maintainability (extendibility).
Process Description	Refactoring activities should be a continuous part of code development.
Collected Data	• Manually: SVN commit template includes change type (e.g. refactoring) • Implicitly: SVN data provides us with information about changes of architecture. Further Code Metrics /Code Smells can provide insight into decay of code.
Process Violations	**Temporal:** (1) No refactoring activities in the commit template at all (during whole project) (2) Large refactoring only in a single stage (e.g. at the end of the project) **Qualitative:** (3) Increasing amount of code smells

Table 9: Process Conformance Rule for Continuous Refactoring

Process Name	**Pair Switching**
Process Focus	Code is collectively owned, high Truck Factor
Process Description	Pair Switching: subjects are supposed to switch their pair programming partner with each new story card and between iterations.
Collected Data	Manually: SVN commit template include name of programmers and story card number
Process Violations	**Temporal**: (1) The same developer pair working together on two consecutive *story cards* (2) The same developer pair working together on two consecutive *iterations* **Qualitative:** (1) The projects Truck Factor is low

Table 10: Process Conformance Rule for Pair Switching

Before the start of the programming project the researcher teams from LUH and UMD agreed to investigate the conformance of the three XP practices. Each of the three practices was translated into a process conformance template (Tables 8 to 10). Further, they agreed on the type of data to collect. Automatically and existing data was derived from the Subversion code repository that the subjects used to coordinate their work.


Comment

```
Developers: <Name1, Name2>

Story Card: <ID>
Accepted: <yes, no>

Type of change: <new feature, enhancement, refactoring, bug-fix, test-fix, ... (other?) >

Comment: <what was done?>
```

Figure 23: Subversion Commit Template for Manual Data Collection

Additionally, a small amount of manually collected data was captured. The researchers provided the developers with a special Subversion commit template[10] that had to be filled in every time developers committed new code to the repository. As shown in Figure 23 the following manually collected data was provided by the developers:

- The names of the two programmers in a pair
- The story card id that was implemented or changed by the commit
- The type(s) of change(s) from the set: {new feature, enhancement, refactoring, bug-fix, test-fix, other}

After each iteration of the XP project, the researchers at UMD created a report with the results of steps two and three of the presented approach (Figure 10). The report was sent to the researchers on site before the start of the next iteration. There is a time shift of 6 hours between UMD and LUH. The researchers specifically planned to use this time to create the report and thus benefit from the global distribution of the two sites. From the German perspective, analysis was done overnight.

The report included quantitative analysis describing how many violations occurred (Figure 10: step 2), as well as visualizations to give better insight into which components are affected (e.g. Java classes not being developed according to the Test Driven Development practice) and/or which developer violated the practice (e.g. for Pair Switching). Further, the report included descriptions of how the violation

[10] Subversion provides the developer with a text field every time new code is uploaded. Usually this commit message is used to describe the changes made to the code base.

detection rules were tailored over time (Figure 10: step 4). Optimizing the rules of the templates was done by a manual in depth analysis of false negatives and false positives. A typical example of a false positive was the Java Interface classes that were wrongly marked as violations in the first version of the Test Driven Development template.

It was up to the researchers at LUH how to use the reports to intervene with the ongoing projects. They discussed the violations that were found in the Test Driven Development practice with the subject groups *before the third iteration* and advised them to better adhere to the practice.

After the last iteration the developers received an end-of-study questionnaire that asked how well they followed the different XP practices. To increase the chance of receiving the most honest answers developers neither had to provide their name nor the project they were working on.

6.4.2 Study Results

The following paragraphs summarize the data that was collected during the study, the violations that were found, and the self reported data the developers provided through the end-of-study questionnaire.

	Zeit			KlaRa		
Iteration	New Classes	Test First Follow.	**Conf. Level (%)**	New Classes	Test First Follow.	**Conf. Level (%)**
1	11	3	**27.3**	9	5	**55.6**
2	7	1	**14.3**	4	3	**75.0**
3	5	3	**60.0**	2	1	**50.0**
4	3	2	**66.7**	6	5	**83.3**
Totals	26	9	**34.6**	21	14	**66.7**
Combined Conf. Level (%)				**48.9**		

Table 11: Test-Driven Development Results

Test Driven Development (TDD)

Table 11 shows the results for the two groups (*Zeit* and *KlaRa*). The conformance level (in the Table abbreviated with "Conf. Level") for TDD was calculated as follows: for each of the four iterations the newly developed Java classes (in Table "New Classes") were considered, and it was checked whether unit test classes were created according to the practice. The conformance level then describes in how many cases the developers followed the test first practice. As example, if no process violations could be identified the conformance level would be 100%, if violations can be found in half of the cases the level would be 50%, and so on.

The data shows that the developers of project *Zeit* followed the practice in only 27.3% of the cases in the first iteration and scored even lower (14.3%) in the second iteration. The developers were made aware of their rather poor performance at the beginning of the third iteration, in a stand up meeting, and improved their conformance to 60% after iteration three, and to 66% after the fourth and last iteration. The *KlaRa* team shows better and more stable conformance levels. They scored between 50% (iteration 3) and 83% (iteration 4) conformance level.

The combined level of both teams is calculated by the total number of classes developed divided by the ones developed according to TDD. The combined level of 48.9% indicates that study subjects followed the practice on average in about half of the cases.

How often did you write the test case before the implementation?	*Instances*	*Percentage*
Never	2	14%
Sometimes	8	57%
Most of the time	4	29%
Always	0	0%

Table 12: Questionnaire Answers for Test-Driven Development

The end-of-study questionnaire data shows a similar result. The developers were asked how often they wrote a test case before the implementation. Subjects could answer on a scale from “Never”, “Sometimes”, “Most of the time”, and “Always”. Table 12 shows the results. No subject said the practice was followed all the time, and only 29% of all developers said that they followed it most of the time. The majority said they followed it sometimes (57%) or never (14%).

Continuous Refactoring

The second practice under investigation was Continuous Refactoring. In comparison to the other investigated practices, the process violations were formulated rather weakly (see Table 9). The reason for this was that no good description could be found that describes how much or with what frequency refactoring should be done according to the XP practice. Developers are asked to refactor code whenever they feel it is necessary to adapt the design to new requirements or to improve maintainability. Therefore, I measured the number of times the developer teams indicated in the Subversion template that they refactored. The objective was to find out if subjects refactor at all and if there were differences in the amount of refactorings between the two groups.

	Zeit			**KlaRa**		
Iterat.	**Changes**	**Refac.**	**Ratio**	**Changes**	**Refac.**	**Ratio**
1	11	4	36%	4	1	25%
2	7	2	29%	8	0	**0%**
3	4	0	**0%**	9	5	56%
4	15	1	7%	8	1	13%
Totals	**37**	**7**	**19%**	**29**	**7**	**24%**

Table 13: Continuous Refactoring Results

The data in Table 13 shows that developers reported to have performed refactoring activities at a constant frequency. Both projects show about the same refactoring ratio: 19% (*Zeit*) and 24% (*KlaRa*) of all changes included the desired activity. Only two iterations did not include any refactoring activities (iteration three for team *Zeit*, and iteration two for team *KlaRa*). Therefore, violations of the practice as defined in the process conformance rule (Table 9) could not be detected. Even if the presented analysis could not find any violations, it helps to build a stronger baseline: the refactoring ratios from this study can be used to detect violations when used as thresholds in a future study.

How often did you refactor?	*Instances*	*Percentage*
Never	3	21%
One time	4	29%
Few times	6	43%
With every new story card	1	7%

Table 14: Questionnaire Results for Continuous Refactoring

Further, the self-reported data can help give the numbers more meaning. From the post-study questionnaire (Table 14) one can see that seven developers said that they either "*never*" refactored or that they refactored only "*one time*". The other seven subjects indicated to have done refactorings "*few times*" or "*with every new story card*". The answers indicate that the practice was not followed by all developers (at least three subjects did not refactor as often as the practice recommends); therefore the computed refactoring ratios of 19% and 24% might still be below an optimal, desired ratio.

Pair Switching and Collective Code Ownership

The third XP practice under investigation was Pair Switching and Collective Code Ownership. The goal of Collective Code Ownership is to ensure that all developers collectively own the code to be able to make changes and that a loss of a small set of programmers does not lead to project failure. The practice is not defined as a set of activities that have to be followed; it rather is a goal, i.e. a desirable state, which is reached through two other XP practices: Pair Programming and Pair Switching (particularly switching pairs regularly during iterations).

To detect non-conformance in Collective Code Ownership two measures were investigated:

1. Temporal: Adherence to the activities defined by **Pair Switching**.
2. Qualitative: Assessment of the project's **truck factor**

As for Pair Switching, I note that the study conductors required that programming pairs were reshuffled at the beginning of each development day (i.e. each iteration). That means that the process managers partly *enforced* the Pair Switching practice.

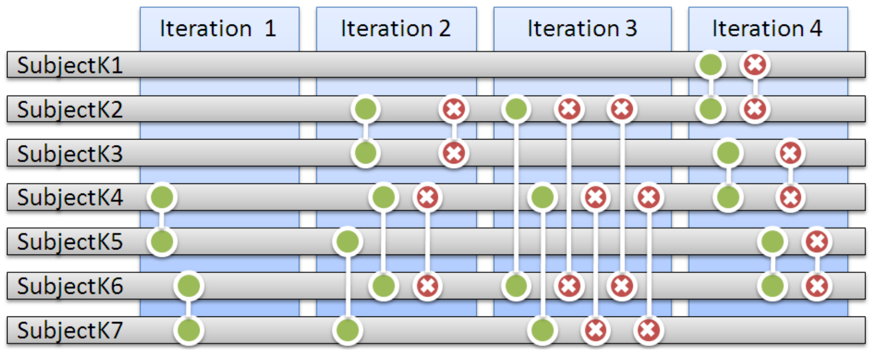


Figure 24: Pair Switching for Team KlaRa

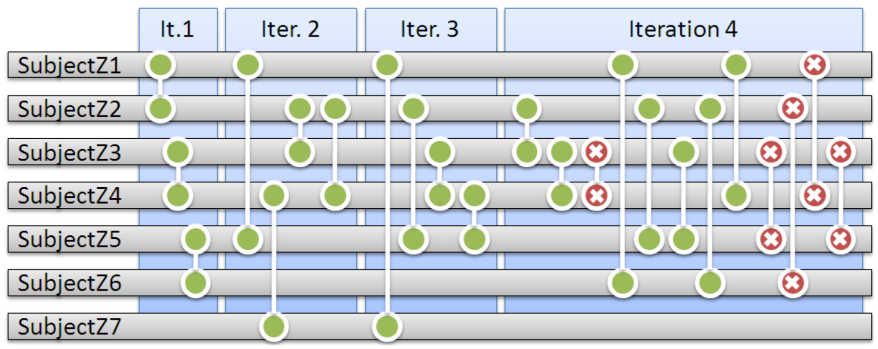


Figure 25: Pair Switching for Team Zeit

Pair Switching showed a significant amount of violations. Figure 24 visualizes the pairs working together on story cards for each of the four iterations in project KlaRa. A paired point in the figure represents a programmer pair working on one new story card. The points are ordered along the x-axis by time and day. Points with a cross mark indicate that the same pair worked on more than one story card consecutively (i.e. a violation against the process definition). From the second iteration on, violations indicate that developers did not switch their teammates as they were supposed to, between two story cards. During the second, third and fourth iteration they generated nine violations against the practice. For example, SubjectK2 and SubjectK3 worked on two story cards in a row during the second iteration, and so did SubjectK4 and SubjectK6 during the same iteration. The graph for KlaRa further shows that the pairs *never* change during an iteration (i.e. one development day): the subjects only switched their partners at the beginning of each day (which was enforced by the study conductors).

For project *Zeit* (Figure 25) the Pair Switching was followed the first three iterations without violations. Developers switched with every new story card. Only during the last iteration, where they worked on a larger amount of story cards, five violations against the practice could be detected.

How often did you switch pairs according to the pair switching practice?	*Instances*	*Percentage*
Never	1	7%
Sometimes	3	21%
Often	9	65%
Always	1	7%

Table 15: Questionnaire Results for Pair Switching

Again, the reported conformance from the questionnaire shows a similar result as before (Table 15). Only one developer agreed to have followed the practice all the time (this is also true for the data in Figure 24 and Figure 25: SubjectZ7 is the only one without violations).

The **Truck Factor Analysis** gives insight into how well the code is collectively owned at the end of the projects. For this I defined (to my knowledge for the first time) an analysis technique that builds upon the data collected through the code repository to assess the *Truck Factor*. The definition and an example of the *Truck Factor Metric* are given in the Appendix 9.1 . As pointed out in earlier chapters one might not always have a clear understanding as to what the expected measures should look like in such cases (i.e. which truck factor measure the practice should produce when followed). Therefore, the data was analyzed with two objectives. The first objective was to compare the two projects to see if their truck factors differ. The second objective was to compare the numbers to three non-XP projects that do not specifically focus on introducing processes to improve Collective Code Ownership.

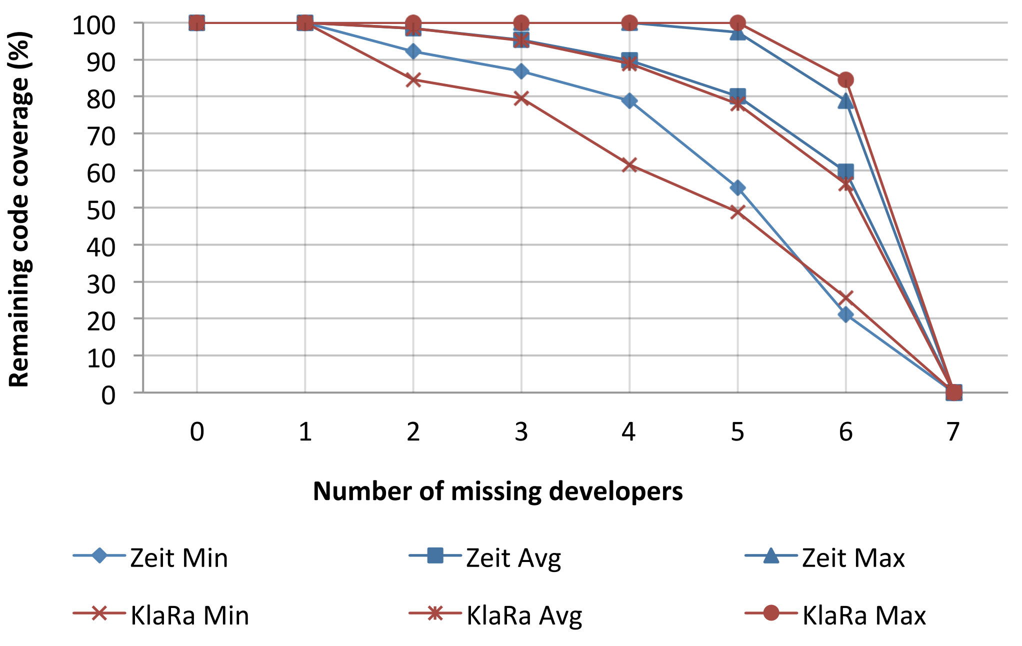


Figure 26: Truck Factor Characteristics for both Projects

Figure 26 shows the according truck factor characteristics for both XP projects. The worst case (i.e. Min), average case, and best case (i.e. Max) scenarios for Zeit and KlaRa are plotted. The graph shows that Zeit has better worst case performance than KlaRa: assuming a required code coverage of 80% Zeit can lose four out of seven programmers, where KlaRa can only lose three developers. The average case performance is almost equal with a slight advantage for Zeit. Figure 26 also shows the impact of pair programming: the loss of one programmer can always be covered by the programmers she/he worked with in a pair. The code coverage for a truck number of one is in both projects 100% (in worst, average, and best case).

The second question is how these graphs compare to conventional non-XP projects. The motivation for this analysis was the theory that if the goal of the XP practice is reached the collective ownership should be improved compared to projects not performing such processes. Our non-XP candidates were a large scale 2 year development project using the Waterfall lifecycle (i.e. from study *FEASIBILITY*) that I am describing in more detail in (Zazworka, Basili and Shull, Tool Supported Detection and Judgement of Nonconformance in Process Execution), and the development of two research tools developed at the two participating universities: CodeVizard and HeRa (a requirements editor mostly developed by one programmer).

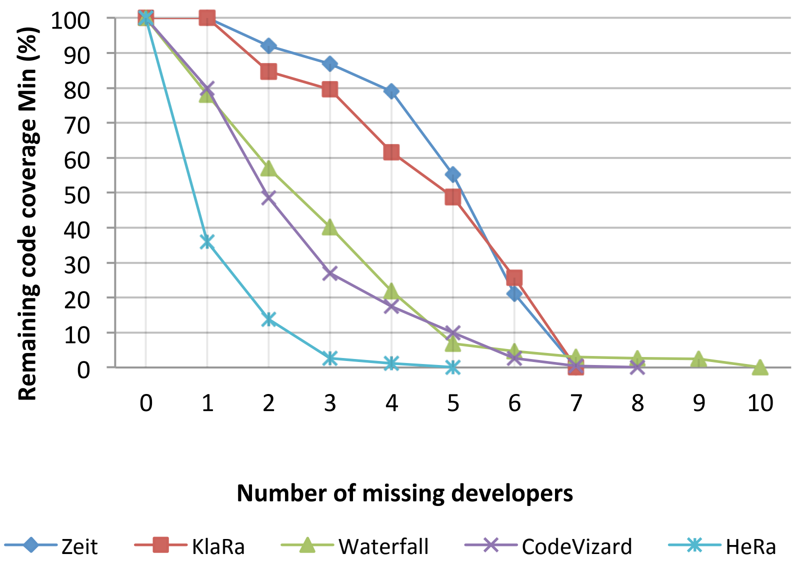


Figure 27: Worst Case Truck Factor for 5 Projects

Figure 27 shows the worst case scenario for all five projects and provides the first evidence that the three non-XP projects have significant lower (i.e. worse) truck factors: the loss of two developers leads in all three non-XP projects to a loss of at least 40% (and up to 85%) of code knowledge, whereas the XP projects would still preserve 85% (KlaRa) and 92% (Zeit) of knowledge.

How much percent of the system have you been working on?	*Instances*	*Percentage*
<25 %	1	7%
25-50%	5	36%
>50-75%	5	36%
>75%, <100%	2	14%
100%	1	7%

Table 16: Questionnaire Results for Collective Code Ownership, Question 1

Are there parts you have worked on alone (with your partner)?	*Instances*	*Percentage*
Yes	6	43%
No	8	57%

Table 17: Questionnaire Results for Collective Code Ownership, Question 2

In the end-of-study questionnaire, subjects were asked how much percentage of the final system they have worked on, and if they think there are parts that they have worked on alone with their partner. The results are summarized in Table 16 and Table 17.

6.4.3 Discussion of Results

The results of the study show that there were many process conformance violations in the process execution in the studied environment. Developers especially had problems following the Test Driven Development practice and one group performed poorly in following Pair Switching.

The results from the end-of-study questionnaire show that subjects are aware of not following a particular practice. When they were asked later why they did not follow Test Driven Development they answered that "*the implementation of new features to satisfy customer needs had a higher priority than following the steps of the process*".

Overall, conclusions for this study can be summarized as followed:

The results show that it was possible to translate three XP practices into the suggested scheme, to collect data non-intrusively with minimal manual effort, and to formulate and detect violations against the defined practices (Research Question 1). For most of the qualitative violations, thresholds and measures were found and tailored during the execution of the processes. Qualitative violations seem to be harder to define upfront. However, once found they can potentially be used in study replications or future projects.

The perceived conformance of the subjects fits the measured one to some extent. For two practices one could find a significant amount of violations and subjects admitted to not following the practice at all times (Research Question R2B).

Subjects were advised to improve their conformance to Test Driven Development one time before iteration three. The impact is visible in the conformance level (Table 11).

The number of violations could be lowered, but they still occurred after this feedback (Research Question 4).

The study shows that it was possible to improve and adjust the rules to the environment and practices. For all the rules, I did not have a good understanding of the qualitative levels before the study but was able to derive measures and thresholds during the execution to some extent. Further, I was able to catch some special cases (i.e. Java interface classes) to improve the automated detection of violations (Research Question 3).

So far, I was unable to find relationships between the adherence to a process and the resulting quality attributes of the product. However, the truck factor analysis gave insight into how a practice can help to reduce risks in a project. The KlaRa team violated the Pair Switching practice more often than Zeit and achieved a lower worst case truck factor. The major finding related to the truck factor risk is that the XP practices Pair Programming and Pair Switching appear to be linked to a better truck factor, when compared to conventional projects.

6.5 CROOM2: Classroom Study II

The second classroom study took place in a similar classroom environment as the first one and followed the same quasi-experimental design, i.e., an equivalent time samples design. Again, two groups of students developed a small sized software application following the XP development methodology. A difference from the first study was that one of the teams was distributed across two locations (four developers at Hanover, and four developers at Clausthal). Both locations (in Germany) are about 60 miles (100 kilometers) apart so that development teams had to make use of electronic communication channels (e.g., Skype[11] calls). The researchers (i.e. process managers) were interested in how distributed XP development compares to regular (one site) XP development. Specifically, they were curious to see if the XP practices applied differs in a distributed development and if the provided communication tools are efficient in distributed XP environments.

As in the first classroom study, three XP practices (TDD, Pair Switching, and Continuous Refactoring) and one additional Communication Practice: broadcast of story card and names, were investigated with respect to process conformance. Therefore, this study could make use of the previously defined process conformance templates for the three XP practices applied in *CROOM1*.

6.5.1 Study Design

The study followed the same equivalent time samples design as the first one, with the following differences:

- Instead of four development days, subjects developed for five days.
- The target applications were developed for Java Android mobile phones. All study subjects did not have any previous experience in developing Java applications on that platform. The target application of the distributed team helps cell phone users in medical emergency situations and was called *Notfallplan* (English translation: emergency plan). The application of the non-

[11] Skype is a proprietary application that allows voice calls over the Internet. It further allows video calls, and text message chats. More information about Skype can be found on: http://www.skype.com or http://en.wikipedia.org/wiki/Skype

distributed team was a game for simulating the blood alcohol concentration after drinking alcoholic beverages. It was simply called *Spiel* (English translation: game).

- The distributed team had two XP coaches present, one at each location. The customer was located at the Hanover location. Developers at Clausthal could communicate with the customer via the various electronic communication channels. Figure 28 shows how the roles of the conformance approach were distributed across the two locations.
- The end-of-study questionnaire was modified to ask more specifically for possible shortcomings and improvements of the investigated XP and communication practices.

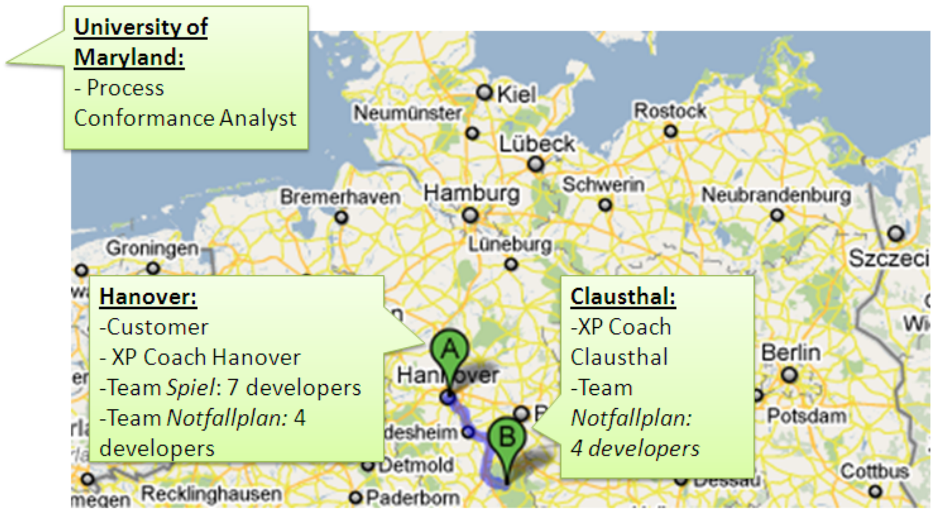


Figure 28: Distributed study setup for the second study in classroom

6.5.2 Step 1: Defining Process Conformance Rules

The previously defined process rules for the three XP practices were initially not changed and used as described in Table 8, Table 9, and Table 10. A rule change became necessary for Test Driven Development (Table 19) because one of the teams did not follow the suggested naming convention. For the communication practice "Broadcast story card and names" a new process conformance template (Table 18) was developed prior to the start of the study.

Process Name	**Communication Practice: Broadcast of story card and name**
Process Focus	Communication and increased productivity.
Process Definition	Developers should use Skype status messages to broadcast who is working on which story card in a timely manner.
Collected Data	Skype Status Log containing: • online, offline timestamp • changes in status message Subversion commits
Process Violations	**Temporal**: • empty status message for more than 1 hour • Subversion commit does not fit SC# or developer names • Developers seem to work in two teams at the same time **Qualitative:** • Incomplete information in Skype status

Table 18: Process Conformance Rule for the communication practice: Broadcast of story card and name

Process Name	**Test Driven Development**	
Process Focus	Improved correctness.	
Process Definition	For each component (i.e. Java class) developers are supposed to create a JUnit test class (collection of test cases) prior to the development of the component.	

Collected Data	Subversion code history. Developers are advised to use following file naming convention for implementation and test classes:
	Implementation class:
	`SomeName.java`
	Test class:
	`SomeNameTest.java`	

Process Violations	**Version 1:** **Temporal**: (1) Implementation classes (but not interface classes) without test classes. Violation detection: Implementation class is checked into the Subversion repository before its according test class.	**Version 2:** **Temporal:** (1) Implementation classes (but not interface classes) that are not tested by any test classes. Violation detection: Implementation class is checked into the Subversion repository but no test class accesses this implementation class.

Table 19: Adjusted Conformance Rule for Test Driven Development

The communication practice required that subjects, i.e. the developers, maintained the name of the current pair developers and story card by using Skype status messages. Skype allows for each user (i.e. machine it is installed on) to provide a status message (in Skype terminology: mood message) that is shown to all befriended[12] Skype users. For the study, Skype accounts were created for the four workstations that development pairs worked on, the two XP coaches, and the customer. All of the accounts were then befriended, i.e. added to each other's contact lists. The

[12] Befriended users are the users that are shown in the contact list of the Skype application. To friend another Skype user a request has to be sent, and confirmation to that request has to be given by the requested user.

communication practice should help to create a global understanding of which developers are working together on which workstation, and what story card the development pairs are currently working on. The overall goal is to create a fluent and transparent environment that decreases the amount of rework (i.e. duplicated work done on both sites) and increases productivity.

The process description in Table 18 recommends developers to maintain this status information in a "timely manner". More precisely, violations against the practice define, that Skype status messages are not allowed to be left blank for more than one hour (Table 18: first temporal violation) and that the posted information has to be complete (Table 18: first qualitative violation). Completeness of information requires that at least the names of the developers and the story card are maintained in the status message. To check for outdated or wrong status information the second temporal process violation defines that data from the subversion repository (i.e. the names and story card in the SVN commit message as shown in Figure 23) should map to the one in the Skype status message.

To instrument the Skype status message changes, a small tool, named *SkypeContactsStatusTracker* was developed. The tools allows for tracking of status changes by simulating a Skype client that is befriended with all other accounts. To do so *SkypeContactStatusTracker* reads unobtrusively every five seconds the status messages from all project participants and saves them in a log file[13]. The tool can be classified as automatically and supplementary data collection activity according to the

---

[13] More precisely, the log file contains only changes of the Skype status message and the Skype online status within a five second resolution.

classification scheme given in Chapter 3. It does not change the behavior of the subjects, but will require additional cost for installing and running it on an additional workstation. Further, some cost has to be spent in interpreting the log file results. In this study the tool was run on the workstation of the conformance analyst located at the University of Maryland.

```
(1)Status  changed,1274772259207/Tue  May  25  03:24:19
EDT  2010,pair4-c,  Mood  Message:   “”->””  ,OFFLINE  ->
ONLINE

(2)Status  changed,1274773314721/Tue  May  25  03:41:54
EDT  2010,pair3-c,  Mood  Message:  technik  lernen  mit
E*** ->”” ,ONLINE -> ONLINE

(3)Status  changed,1274774730428/Tue  May  25  04:05:30
EDT 2010,pair4-c, Mood Message:  “”-> Fe****/ Mo****:
Story Card #15  Startbildschirm,ONLINE -> ONLINE

(4)Status  changed,1274779411979/Tue  May  25  05:23:31
EDT  2010,pair2-h,  Mood  Message:  Story  Card:  29
(Notfallbutton), An***, Pa*** -> Story Card:  (Grund
Struktur überlegen), An***, Pa***,ONLINE -> ONLINE
```

Figure 29: Log file content of the SkypeContactsStatusTracker tool (developer names are anonymized using *)**

To illustrate in more detail, a short excerpt of the log file illustrates the collected data in Figure 29. Four changes of status messages are displayed. The first one shows that on Tuesday, May 25th, the second pair in Clausthal (pair4-c) went from being OFFLINE to ONLINE at 3:24am EDT (that is 9:24am in German time zone). In other words, they logged into their workstation at that time. A status message is missing at this point in time. As a second example, the last log entry in Figure 29 (change number (4)) shows a change of status message for pair2-h (the second pair in Hanover). The status message was changed from

“`Story Card: 29 (Notfallbutton), An***, Pa***`”

to

“`Story Card:  (Grund Struktur überlegen), An***, Pa***`”

which indicates that the same developers continued to work on a second story card. The number of the story card is not provided; the status message is therefore incomplete.

6.5.3 Step 2: Violation Detection

As in the first study, the violation detection was performed overnight at the University of Maryland. Reports of process conformance and according violations for each of the four practices were sent to the XP Coach in Hanover at the beginning of each of the development iterations. Discussion of the violations with the process enactors was done during the daily stand up meeting in the morning.

A violation detection algorithm and conformance analysis for the practice Test Driven Development and Continuous Refactoring was implemented into CodeVizard.

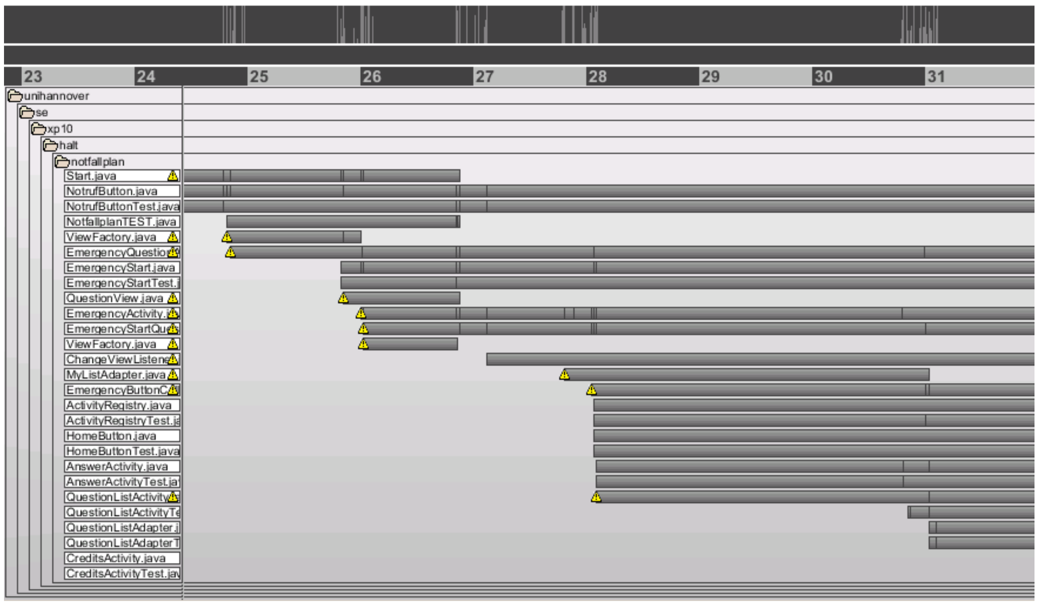


Figure 30: Process violations against TDD in project *Notfallplan*

Figure 30 shows the CodeVizard visualization for violations against TDD in one package of the project *Notfallplan*. The small yellow warning icons in the view indicate that the practice was violated. More precisely, the identified violations indicate that no test class was found when the source code files were checked into the repository. One can see in the figure that violations occur at check-in time of new classes, e.g. on May 24th, two classes were added (`ViewFactory` and `EmergencyQuestion`), and for both no test class were added.

For practice Continuous Refactoring, the subversion commit comments were extracted and it was counted how many times developers indicated to have refactored during each development iteration.

The violation detection for the other two practices was done in a more manual way. For Pair Switching, graphs as already presented in the previous section (Figure 24 and Figure 25) were created to identify process violations. For the new Communication

Practice, broadcast of the story card and name, a new graph was created that shows the relevant data, e.g. online status, status message, and SVN commit comments.

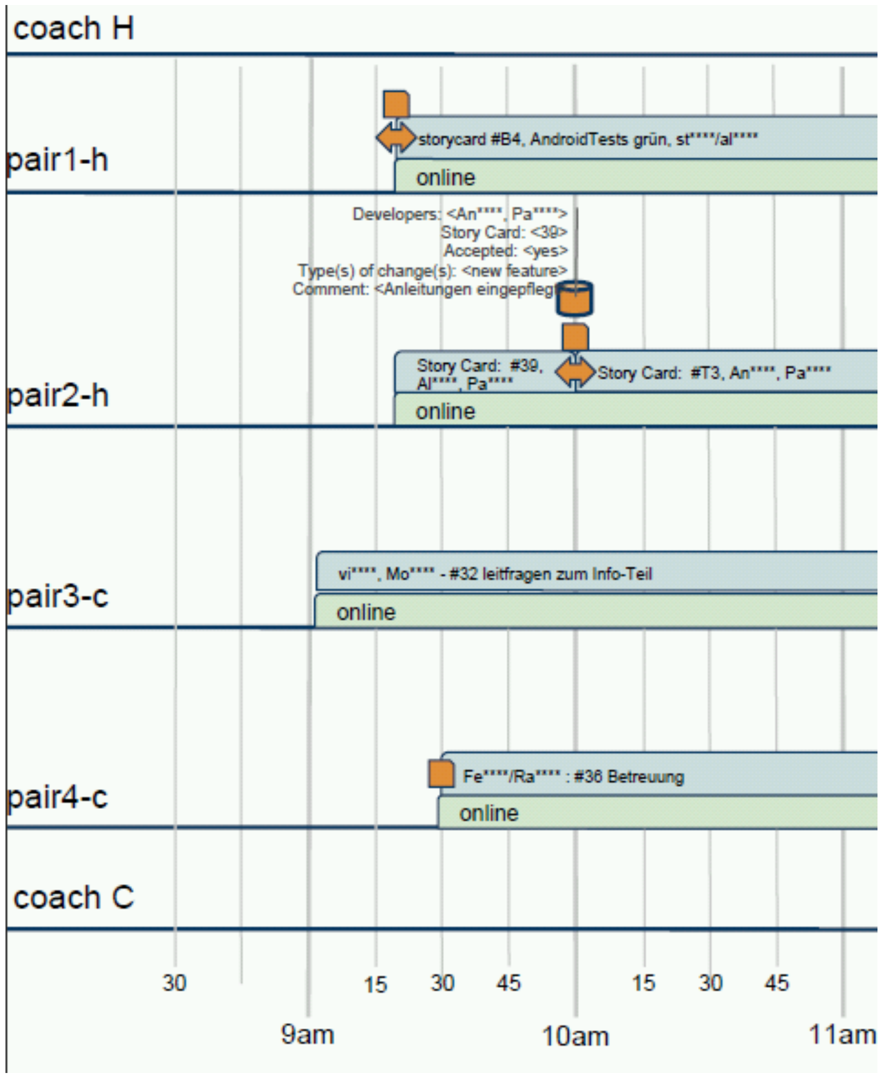


Figure 31: Skype Status graph to investigate violations against the communication practice: Broadcast story card and name

Figure 31 shows a part of the graph that was created to identify violations against the communication practice. The light green bars (labeled with online) for each distributed development pair indicates when they were logged into Skype (Skype status: online). From the figure one can conclude that pair3-c (the first pair in Clausthal) logged in earliest that day at around 9am. The other three teams started working between 9:15am and 9:30am. On top of the online status the current Skype status message is displayed on light blue background. For all developer pairs one can read the names of the developers working together and the current story card that they are working on. Additionally, some development pairs noted the description of the story card (which is not mandatory). If one pays closer attention to the developer names one will find the following inconsistency: developer "al****" worked in pair1-h and pair2-h at the same time at the beginning of the day, which is impossible. At 10am pair2-h changes their developer names. A logical explanation of this pattern is that in reality development pairs changed at the beginning of the day (pair switching is indicated by the double arrow). However, pair2-h forgot to update their status for the first 40 minutes and violated the practice of keeping the status up-to-date. Once they noticed this, they changed their developer names, and they also committed the code that was changed for that story card (#39). In the graph, Subversion commits are indicated by the cylinder symbol. The commit message is displayed above the symbol. Ideally, whenever a pair switching occurs one should see changes of status messages for *both* teams on one site at the same time.

6.5.4 Step 3: Gathering Additional Information

In this study, as in the last one (*CROOM1*), the process manager (i.e. the XP coach in Hanover) acted as bridge between the process enactors and the conformance analyst. That means that the analyst was not able to interview the enactors during the study.

However, an end-of-study questionnaire that was designed by the analyst focused on questioning why the enactors could not follow the practices.

During the study the analyst sent daily reports to the process managers, and these reports were used in the daily stand up meetings to point out process violations to the developers.

To gather additional insight, CodeVizard and email conversations with the process managers were used. For example, even before asking the subjects it could be understood (by looking at CodeVizard) that Test Driven Development was violated for components that reside in parts of the Android graphical user interface part of the code. Later investigation found that this was due to a lack of experience in how to test this code effectively.

6.5.5 Step 4: Process/Rule Improvement

Due to the short study duration no modifications on the processes itself were performed since the study design aimed on investigating process conformance and productivity of XP development, and not at tailoring the XP practices.

Overall, managers believed that process enactors (i.e. students) should follow the practices better. Thus, process enforcement was done by reminding the developers in the daily stand up meetings by presenting them with the analysis results of the conformance reports. Finally, an end-of-study questionnaire asked study enactors about what they would change on the process.

6.5.6 Results

Test Driven Development

Table 20 shows the number of newly introduced components for each of the development iterations and how often test cases that satisfied version 1 (with the naming convention) and version 2 (without the naming convention) as described in Table 19 (conformance rule for TDD) could be found. This statistic gives insight into the amount of temporal violations as defined in Table 19.

Iteration (Day)	Distributed Team: *Notfallplan*						Non-distributed Team: *Spiel*					
	Detection V1			Detection V2			Detection V1			Detection V2		
	New classes	Test case in repository	Estimated % TDD followed	New classes	Test case in repository	Estimated % TDD followed	New Classes	Test case in repository	Estimated % TDD followed	New Classes	Test case in repository	Estimated % TDD followed
1	9	0	0%	9	2	22 %	7	2	29 %	7	3	50 %
	Manager Feedback: - had to learn how to test android components. - found out that some of the developers in Clausthal did not know Java and were very inexperienced. - We analyzed the tests Wednesday night and forced our developers to enhance them on Thursday. **Feedback on Conformance was given to Developers**						No feedback					
2	11	5	45%	11	6	54 %	14	1	7%	14	11	79%
	Manager Feedback: - Testfirst was my main topic for the Thursday morning stand-up meeting. We decided to improve the situation and stopped working on new customer stories until these issues were fixed. **Feedback on Conformance was given to Developers**						No feedback					
3	3	3	100 %	3	3	100 %	6	0	0%	6	1	17%
	Feedback on Conformance was given to Developers						**Feedback on Conformance was given to Developers**					
4	6	3	50%	6	3	50 %	15	0	0%	15	3	20%
	Feedback on Conformance was given to Developers						No feedback					
5	1	1	100 %	1	1	100 %	11	0	0%	11	5	45%
Total	30	12	40%	30	15	50 %	53	3	6%	53	23	43%

Table 20: Conformance Results for Test Driven Development

The data in Table 20 indicates that the distributed team *Notfallplan* followed TDD in half of the cases (50%) when using the second, optimized version of the rule. This conformance level[14] is within the range of the results from the last class room study (34% and 66%). Therefore, when comparing the differences in the environment, it seems that the distributed development team, when compared to the non-distributed development teams of the last study, seemed to have neither a positive nor a negative effect on the quality of TDD execution. This is not a surprising result, since TDD is performed locally within one development pair and does not require that development pairs communicate.

The *Notfallplan* team could also increase their conformance during the first three days from 0% to 100%. Reminding developers of the importance of TDD in the daily stand up meetings seemed to have the desired effect.

When using the tools (CodeVizard) to inspect the violations in detail one can see that test cases were sometimes developed after the implementation class. This rework was suggested to the developers by the managers in the daily meetings. Overall, for *Notfallplan*, 15 classes (50%) were developed in test first order, 6 classes (20%) were developed in test last order, and for 9 classes (30%) no test classes could be found.

The data for the non-distributed team *Spiel* suggests that conformance to the TDD practice was very low when using the first version of the rule for detecting violations. Only on the first two days test cases could be found and assigned by using the file

[14] Strictly speaking, the rate is the inverse of a violation rate. One cannot necessarily conclude that TDD was followed in 56% of the cases when implementation classes and test cases appear in the right order in the repository. Developers could still have developed the classes in the wrong order. However, one can say that in 1-56%=44% of the cases no test class could be found, and therefore TDD, as defined in Table 8, was violated.

naming convention (version 1). These results reveal that either the developers did not follow the practice as described in the conformance rule, or that the detection method for violations was not applicable. The third step of the approach addresses this question. The tool support that was developed as part of this thesis (CodeVizard) showed that in general test classes were developed, but that developers did not follow the mandatory naming convention: developers were supposed to give a test case the same name as the implementation class and to append the suffix "Test". For example, a class `Application.java` should have a test case named `ApplicationTest.java`.

Implementation Classes	Test Case Classes
BarActivity.java	AddScoreEntriesTest.java
BarActivityListener.java	AllTests.java
ChooseGameCharacterActivity.java	BackgroundColorTest.java
DeadActivity.java	BarDisplayTest.java
DeadActivityListener.java	ChooseGameCharacterTest.java
DiscoActivity.java	DeadActivityTest.java
DiscoActivityListener.java	DiscoBrawlTest.java
DiscoArbitrationActivity.java	DiscoDanceTest.java
DiscoBrawlActivity.java	DiscoDisplayTest.java
DiscoBrawlDialogActivity.java	DiscoHelpsituationBlameTest.java
DiscoBrawlDialogActivityListener.java	DiscoHelpsituationComplimentsTest.java
DiscoDanceActivity.java	DiscoHelpsituationDrunkenTest.java
DiscoDanceActivityListener.java	DiscoOfferDrinkTest.java
DiscoHelpsituationActivity.java	DiscoParticipationTest.java
DiscoHelpsituationBlameActivity.java	DogHitDecisionDisplayTest.java
DiscoHelpsituationComplimentsActivity.java	DrunkenPersonTest.java
DiscoHelpsituationDialogActivity.java	DrunkTooMuchForHelpTest.java
DiscoHelpsituationDialogActivityListener.java	FolkfestivalDisplayTest.java
DiscoOfferDrinkActivity.java	GameCharacterTest.java
DiscoOfferDrinkActivityListener.java	GameOverTest.java
DiscoParticipationActivity.java	HighScoreEntryTest.java
DogHitDecisionActivity.java	HighscoreTest.java
DogHitDecisionActivityListener.java	HousepartyDisplayTest.java
DrunkenPersonActivity.java	InstructionsManualTest.java
DrunkenPersonActivityListener.java	MainMenuTest.java
FolkfestivalActivity.java	MarketPlaceDisplayTest.java
FolkfestivalActivityListener.java	OfferDrinkTest.java
HighscoreActivity.java	ParkBenchDisplayTest.java
HousepartyActivity.java	TestHighscore.java
HousepartyActivityListener.java	WalkingActivityTest.java
HousepartyBarActivity.java	
HousepartyDrunkenPersonActivity.java	
HousepartySleepingRoomActivity.java	
illegalArgumentLengthException.java	
InstructionsManualActivity.java	
InstructionsManualActivityListener.java	
MainActivity.java	
MarketPlaceActivity.java	
MarketPlaceActivityListener.java	
OfferDrinkActivity.java	
OfferDrinkActivityListener.java	
ParkBenchActivity.java	
ParkBenchActivityListener.java	
Situation.java	
Start.java	
TestDrunkTooMuchActivity.java	
WalkingActivity.java	

Table 21: Implementation classes and test case classes for project *Spiel*: naming convention was not followed

Table 21 shows the test classes and implementation classes of the project, and illustrates that the naming convention was not followed.

The process manager indicated later that developers had problems testing the code for the mobile devices (Android platform). No developer had previous experience in developing this kind of test code. Therefore, developers created new testing solutions that did not follow the suggested naming scheme. In detail, their test cases were not pure unit tests anymore that solely test one class or function of the code. The developed test cases rather executed bigger parts of the system, e.g. a whole use case, or screen of the application.

This behavior can be seen as an instance of modification of a process to tailor it to a new environment. After tailoring, the current process rule and violation definition was not applicable anymore. It had to be changed.

For the changed rule, the question still remained whether the process enactors implemented these system test cases prior to the implementation classes. A new process conformance rule (see Table 19: version 2) was developed. The new rule does not assume a relationship expressed by file and class name, but assumes that if a test case “uses” an implementation class then implementation class is being tested. “Uses” means in this context that either objects of the class are instantiated within the test case, or that one of the class’ static methods or members is used. As with the previous process rule, one might falsely conclude that the usage of a class in a test case really tests the functionality of the class. In this case, one would miss violations (false negatives). However, if no usage pattern between an implementation class and a test case can be found, one can surely conclude that the implementation class is not being tested by this test case. If this holds for the relationship of one implementation class with all test cases, then TDD must be violated, because no test case exists for this class.

When applying the second rule to the collected subversion data conformance levels improve (see Table 20) when compared to the first version of the detection rule. On the first two days process conformance to TDD was above average with 50% (day 1) and 79% (day 2) of all cases followed. On day 3, conformance drops to 17% and increases afterwards stepwise to 20% (day 4) and 45% (day 5). Overall, team Zeit followed TDD in 43% of all cases. Again, this result falls into the range of the previous classroom study (*CROOM1*) with 34% and 66%.

How well did you follow the process?		
	Notfallplan	Spiel
1: Never	1	1
2: From time to time	1	2
3: Half of the time	2	2
4: Often	4	1
5: Always	0	0
No answer	0	1
How hard was the process to follow?		
	Notfallplan	Spiel
1: Very easy	0	0
2: Easy	1	0
3: Neither easy nor hard	3	1
4: Hard	3	2
5: Very hard	1	3
No answer	0	1

Table 22: End of study questionnaire results for both teams for Test Driven Development

When asked for conformance and difficulty in the end-of-study questionnaire answers of the two teams differed. Team *Notfallplan* indicated to have followed TDD on average about half of the time (answer score median: 3). Team *Spiel* said on average that they followed TDD between “from time to time” to “half of the time” (answer score median: 2). Further, when asked for the difficulty of the TDD practice, team *Notfallplan* said on average that it was “neither easy nor hard” and “hard” to follow the practice (answer score median: 3). Team *Spiel* indicated that it was between “hard” and “very hard” to adhere to the practice (answer score median 4).

Developers further indicated that writing test cases for the Android platform was the main problem of not being able to follow TDD. Developers said that they *“did not know how to test a particular behavior”* and *“the technology Android was unknown, therefore no architecture could be planned”*, and *“if you do not have a clear idea of the architecture, you cannot write any test cases”*. One developer said that *“[...] it was going better after some time”*. Developers also thought that some functionality was *“so simple, that one does not think about writing a test for it”*.

Pair Switching

The Pair Switching practice, as described in Table 10, requires the programming pairs to switch partners every time a story card has been completed. Goal of the practice is to encourage team work and to indirectly improve collective code ownership.

The practice was differently executed for the two teams due to the number and distribution of developers. For the non-distributed team *Spiel* seven developers worked on the code in three pairs plus one additional “free” developer. The free developer worked on the code by himself. Whenever one pair completed a story card the free developer was supposed to switch with one of the pair members. The distributed team *Notfallplan* had four developers in each location, which required them to switch all members at one location as soon as a story card was completed at that location. Developers were never switched across locations. Comparing the two project teams one could argue that the practice might more often disrupt the developers in the distributed environment than in the non-distributed one.

The results for pair switching are, as in the last study (Figure 24 and Figure 25), visualized in form of a graph.

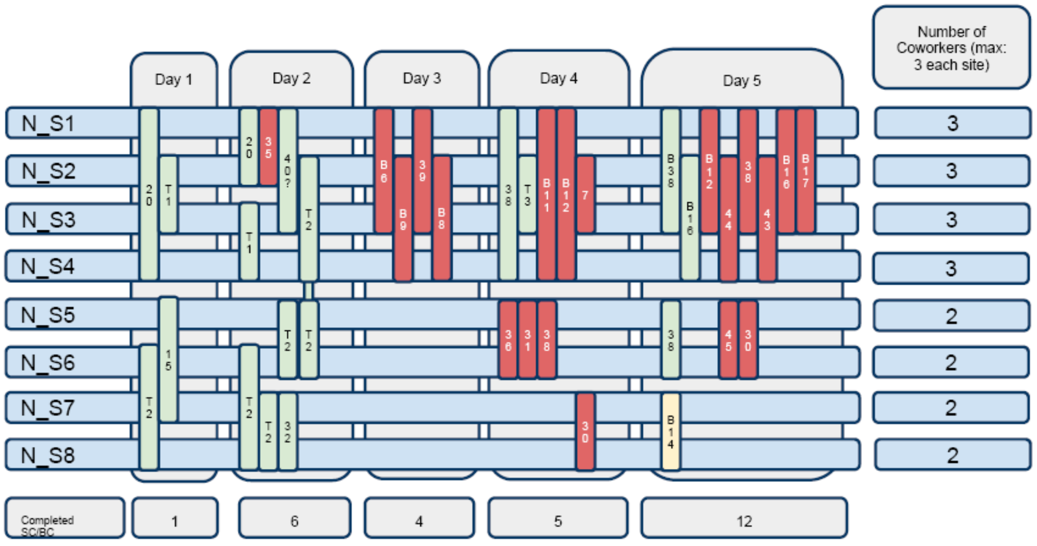


Figure 32: Pair Switching Graph for team *Notfallplan*

Figure 32 shows the pairs working on different story cards on the five development days. Each of the developers is visualized as a horizontal line in the graph (subjects N_S1 to N_S8). A connection, either in green or in red color, indicates that two developers worked together on a story card. The number inside the pair connection denotes the story card number. Red pair connections show that a particular pair has already worked together on the last story card, therefore violating the pair switching practice.

The sum row on the lower end of the picture shows how many story cards were completed on each day. One can see that the team increased their throughput of story cards for each day, from one to twelve cards.
The sum column on the right end of the picture shows how many developers the developer in that row has worked with. Considering the two teams of four programmers, one developer had the opportunity to work with three other developers. The data indicates that this was the case for the upper four developers (in Hanover) but not for the lower four ones (in Clausthal). Some combinations of developers never worked together in a pair, e.g. N_S5 and N_S8, N_S6 and N_S7. In the current definition of Pair Switching this was not defined as a violation. However, the process managers later indicated that they would have expected that all developers work together.

Concerning conformance to the Pair Switching practice, one can see that developers followed the practice better during the first two development days. Only one violation against Pair Switching can be identified: N_S1 and N_S2 work together on story card 20 and 35 in a row. On the last three days conformance was rather poor. One can see that developer pairs never switched *during* one of the last three days. Only at the beginning of days four and five, did developers change their programming partners.

The quantitative and qualitative data collected through the end-of-study questionnaire gave further insights into the reasons for the high amount of process violations. Developers were asked how well they have followed the process, and how hard it was for them to follow the process. Further, they could freely provide text as to why it was hard to follow.

How well did you follow the process?	
1: Never	0
2: From time to time	4
3: Half of the time	2
4: Often	2
5: Always	0
How hard was the process to follow?	
1: Very easy	1
2: Easy	3
3: Neither easy nor hard	3
4: Hard	1
5: Very hard	0

Table 23: End of study questionnaire results for the distributed development team (*Notfallplan*) for Pair Switching

The quantitative data for team *Notfallplan* in Table 23 indicates that developers were aware of violating the Pair Switching practice. Six out of eight developers said that they only followed the practice in half or less than half of all cases. When asked about the difficulty of applying the practice, seven out of eight developers said that it was "neither easy nor hard", "easy", or "very easy". Only one developer said that following the practice was "hard".

Further qualitative data in form of free text[15] indicates that developers thought that switching with every new story card was too frequent. Developers said that *"there were only few times were both pairs finished a card at the same time"* and that they needed more time to *"adjust to the new programming partner"*. Further, *"story cards*

[15] The answers were provided in German language. The author is a native German speaker and translated the questionnaire answers, as closely as possible, into English language.

were too short". One developer from the Hanover team said that the practice was changed towards less frequent switching of the development partner after each iteration, i.e. development day.

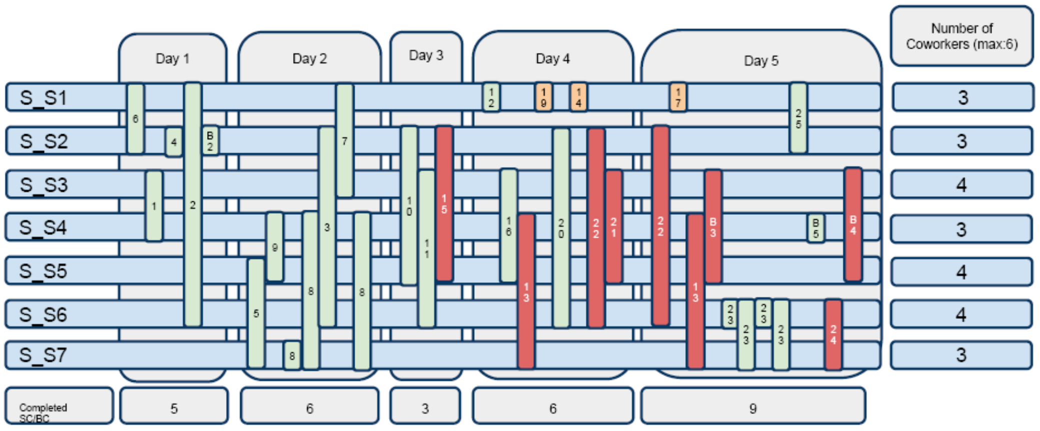


Figure 33: Pair Switching Graph for team *Spiel*

The pair switching graph for the non-distributed team (*Spiel*) shows a very similar picture as for the distributed team. During the first three days pair switching was violated only once. During the last two days violations can be found more frequently. Developer pairs did not switch on the fourth day even though one developer (S_S1) was available to switch with. Overall, developers worked together with three or four (out of possible 6) different partners during the five days. Five out of seven developers took the role of the free developer one time during the project.

How well did you follow the process?	
1: Never	0
2: From time to time	0
3: Half of the time	0
4: Often	6
5: Always	0
No answer	1
How hard was the process to follow?	
1: Very easy	0
2: Easy	2
3: Neither easy nor hard	4
4: Hard	0
5: Very hard	0
No answer	1

Table 24: End of study questionnaire results for the non-distributed development team (*Spiel*) for Pair Switching

When asked for their conformance to the process all six developers who answered the questionnaire (one developer did not fill in the questionnaire) said that they followed it “often”. When asked for the difficulty of the practice, all developers said that it was “neither easy nor hard”, or “easy” to follow.

The answers given in the qualitative part of the questionnaire once again show that developers believed that switching with every story card was too frequent and interrupting. One developer said that, *“when working together on a story card for a long time, it is hard to instruct somebody new [after switching]”*. Another developer said that *“Switching [during a story card] is not a pleasant activity, one wants to finish what one has started”*.

The second part of the violations defined in the conformance template aims at the truck factor. Pair Switching should improve this measure that describes how well code knowledge is uniformly distributed over the number of developers. Especially in this study setup one is interested as to whether the difference in environment (distributed XP vs. non-distributed XP) has an impact on the distribution of code knowledge.

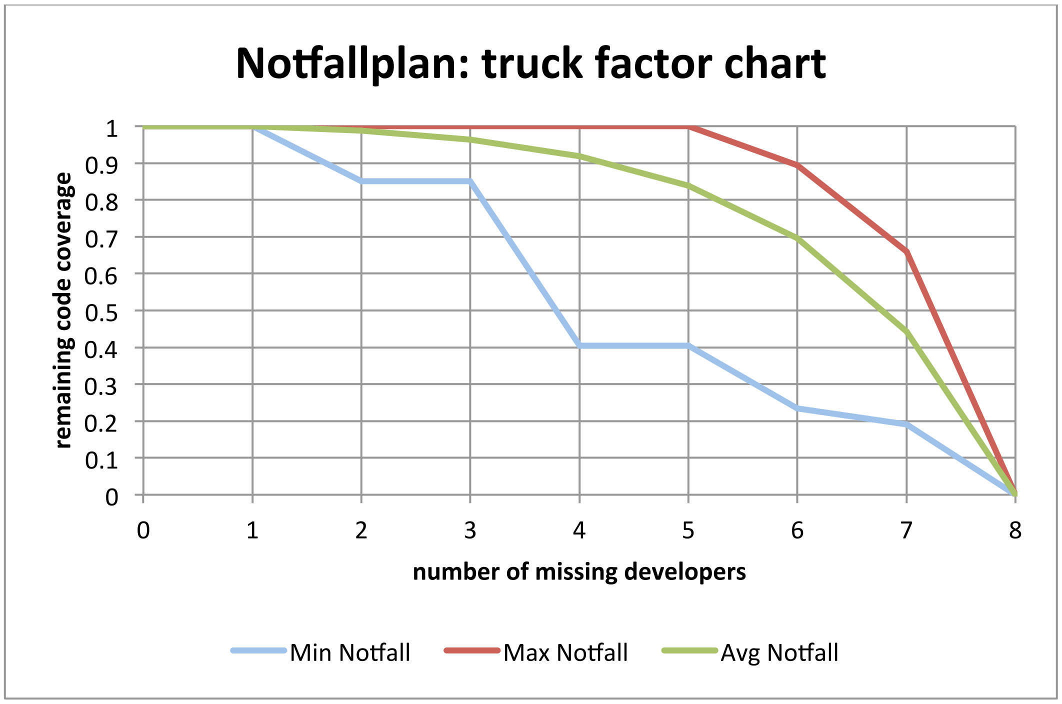
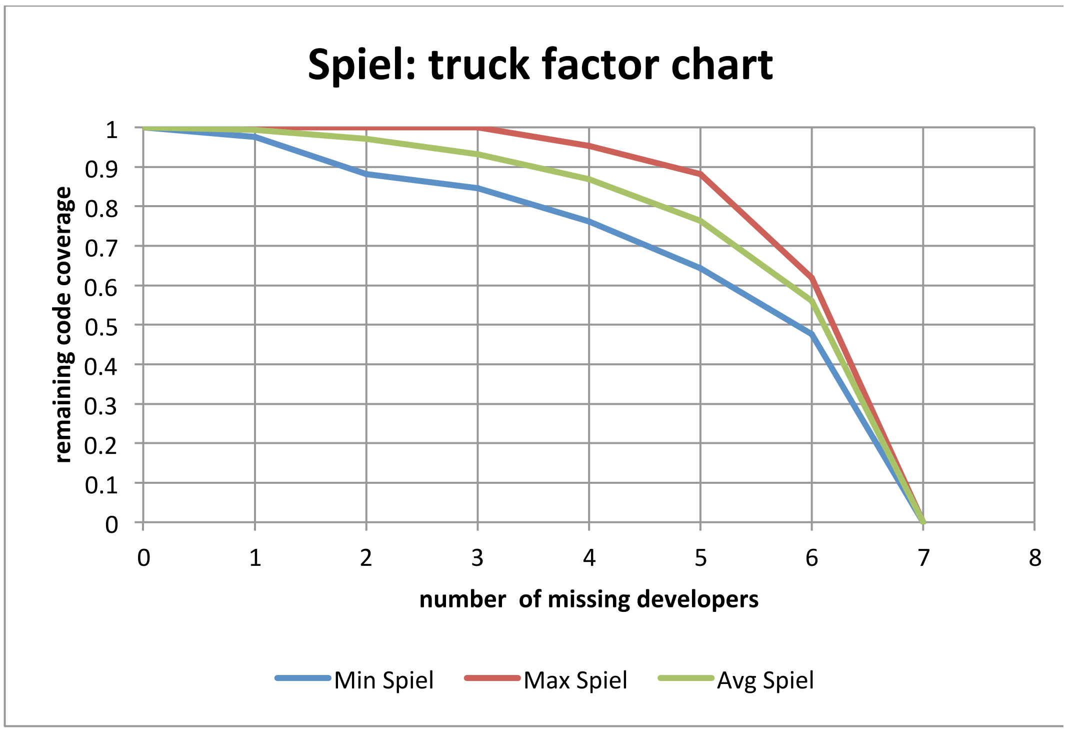


Figure 34: Truck Factor chart for team Notfallplan



The truck factor characteristics for both projects are plotted in Figure 34 and Figure 35. The three lines in each graph show how many developers a project could lose in best, average, and worst case and how much code the remaining developers would cover. For example, in project *Spiel*, even a loss of four out of seven developers (57% of all developers) would only lead to a situation where the remaining three developers would know between 75% and 95% of the code. When comparing the two graphs one can see that especially the worst case line (blue line) differs for both projects. If project *Notfallplan* would lose 4 out of 8 developers then the right combination of developers (i.e. the worst case combination) can lead to a situation where the remaining developers only know about 40% of the code. As it turns out, this worst case combination is the four developers that were working together in one of the locations (i.e. in Hanover). In other words, the Hanover group worked on 60% of the code base **exclusively**. *This finding supports the hypothesis that distributed development has an (negative) impact on how code knowledge is distributed.*

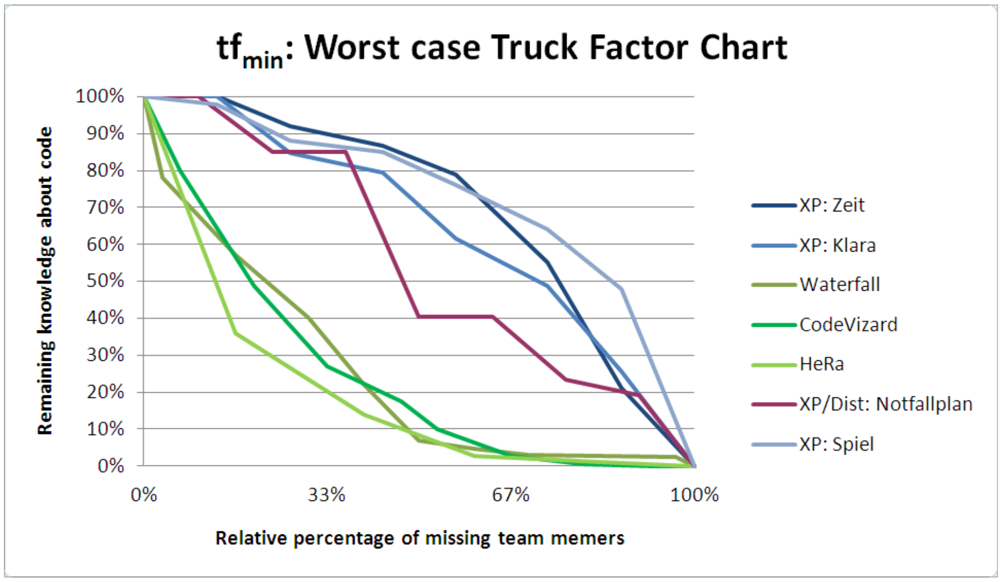


Figure 36: Worst case truck factor chart for seven projects

To make the difference more apparent,

Figure 36 visualizes the four XP projects from *CROOM1* and *CROOM2*, and three projects not applying XP practices (e.g. pair switching). The number of developers was normalized across all projects to compensate for different numbers of developers in each project. As explained earlier (Figure 27) the XP projects have significantly better truck factor characteristics. However, the distributed XP project (purple line: *Notfallplan*) falls short when compared to the non-distributed XP projects (blue lines).

Continuous Refactoring

The continuous refactoring practice advises developers to refactor code often, thereby avoiding postponing refactoring until code becomes hard to maintain. A violation against the practice is detected if developers either do not refactor at all, or in only one single stage of the project (see conformance rule in Table 9). As described in the last study, developers had to provide information about when refactorings were performed through the Subversion commit template (Figure 23). The information was self-reported.

	Notfallplan			**Spiel**		
Iterat.	**Changes**	**Refac.**	**Ratio**	**Changes**	**Refac.**	**Ratio**
1	7	1	14%	7	2	28%
2	8	1	13%	7	4	57%
3	6	2	33%	3	2	66%
4	8	2	25%	8	4	50%
5	15	2	13%	12	3	25%
Totals	**44**	**8**	**18%**	**37**	**15**	**41%**

Table 25: Results for Continuous Refactoring for both teams

The data in Table 25 shows for each iteration how many times developers indicated to have refactored. Both teams refactored their code with each iteration. No violations against the practice could be identified. The non-distributed team (*Spiel*) refactored code in 41% of all cases, the distributed team in 18%. When comparing these two numbers to the results from the last study (*CROOM1*: 19% and 24% refactoring changes), team *Spiel* refactored about twice as often as the other three teams.

How well did you follow the process?		
	Notfallplan	Spiel
1: Never	0	0
2: From time to time	2	1
3: Half of the time	0	0
4: Often	4	4
5: Always	2	1
No answer	0	1
How hard was the process to follow?		
	Notfallplan	Spiel
1: Very easy	1	0
2: Easy	4	4
3: Neither easy nor hard	1	1
4: Hard	2	1
5: Very hard	0	0
No answer	0	1

Table 26: Post Study Questionnaire answers for Continuous Refactoring

The answers from the post study questionnaire (Table 26) show that both teams followed the practice equally. Team *Notfallplan* said that, on average, that they followed the practice "often" (median score of answers: 4). Team *Spiel* said, on average, that they followed the practice "often" (median score of answers: 4). The same holds for the question asking about the difficulty of the process. Both teams indicate that it was "easy" to follow the process (median score *Notfallplan*: 2, median score *Spiel*: 2).

One might be surprised about the very similar questionnaire results since the refactoring ratio presented in Table 25 differed for both teams: 18% for *Notfallplan* vs. 41% for *Spiel*. It was not possible to investigate this difference further, but possible explanations are:

- Both teams followed the practice, but teams reported refactorings differently in the commit template. For example, one team might have reported every small micro refactoring (e.g. renaming a variable in a class), and the other might have reported only larger macro refactorings (e.g. refactorings affecting multiple classes). In a future study, one might want to distinguish micro and macro refactorings.

- Continuous Refactoring defines that software should be refactored *when* required. The software project of team *Spiel* might have required more refactorings than the one of team *Notfallplan*.
- Teams could have been dishonest when reporting refactorings, or when filling in the end-of-study questionnaire.

When asked for the reasons for difficulties with the practice, developers indicated different experiences. One developer said that *"[Refactoring] Changes lead to problems for other developers"*. This might indicate that developers ran into difficulties when using the version control system to synchronize their work. One developer said that refactoring *"was easy and fun to do with Eclipse"*. This shows that developers used built-in refactoring functionalities of the Eclipse IDE[16]. Another developer pointed out problems with language when using Eclipse: *"The XML is partly in German. [It] cannot be refactored by using [Eclipse's] refactoring menu"*.

[16] Eclipse is an open source integrated development environment (IDE) for Java development. It can be downloaded from: http://www.eclipse.org

Communication Practice: Broadcast of Story Card and Name

The new communication practice, as defined in Table 18, was only applied by the distributed team (*Notfallplan*). Violations against the practice are situations where developers either forget to maintain their Skype status message, or if the information is incomplete. The first type of violation (temporal violation) includes situations where developers do not have a status message for more than one hour, or if content of their status message (e.g. the story card number) does not fit the Subversion commit comment. Further, situations as shown in Figure 29, where developers forget to update their names in the status message (and therefore appear to be working in two teams at the same time), are considered as temporal violations. The second type of violation (qualitative violation) includes scenarios where developers post a Skype status, but the status is incomplete, e.g. does not contain a story card number and/or the names of the developers.

	Hanover Location			Clausthal Location		
Iteration	Temporal Violations	Qualitative Violations	Completed Story Cards	Temporal Violations	Qualitative Violations	Completed Story Cards
1	0	1	0	1	0	1
2	2	1	5	0	0	2
3	0	0	4	0	0	0
4	2	0	3	0	1	2
5	3	0	8	0	1	4
SUM	**7**	**2**	**20**	**1**	**2**	**9**

Table 27: Violations against the communication practice for both locations of project *Notfallplan*

Table 27 shows the temporal and qualitative violations for each of the five development iterations. The developers at Hanover violated the practice nine times, and the developers in Clausthal violated the practice four times. At first, it seems that the Hanover team did twice as bad as the Clausthal team in following the practice. However, if one considers the amount of completed story cards (the more story cards are completed the more often developers have to update their Skype status), both teams were following the practice in about the same number of cases: the Hanover team generated nine violations while completing 20 story cards (45% violation rate) and the team at Clausthal violated the practice three times while working on nine story cards (33% violation rate).

How well did you follow the process?		
	Location Hanover	Location Clausthal
1: Never	0	0
2: From time to time	0	0
3: Half of the time	0	0
4: Often	4	0
5: Always	0	4
How hard was the process to follow?		
	Location Hanover	Location Clausthal
1: Very easy	0	1
2: Easy	1	3
3: Neither easy nor hard	3	0
4: Hard	0	0
5: Very hard	0	0

Table 28: End of Study Questionnaire results for the communication process

When asked for their conformance, all developers in Hanover said they followed the practice "often". The developers in Clausthal believed to have followed the practice "always". One possible explanation that developers thought they never violated the practice is that they violated it only three times during five days. This might have fallen below the threshold of recognition.

Developers in Clausthal perceived the difficulty of the process easier than the developers in Hanover. When asked for problems with the process, developers in Hanover said that *"sometimes they forgot to do it"*, especially in situations *"where something unexpected happened".*

6.6 PROF: Long Term Study in Professional Environment

After the approach was tuned and initial tool support was built during the initial feasibility study (*FEASIBILITY*) and the first classroom study (*CROOM1*) it was time to apply it in a realistic professional environment. The chosen environment was provided by a customer of the Fraunhofer Center for Experimental Software Engineering[17] where the author worked part time during this thesis. The Fraunhofer Center supports the customer by providing them with CMMI consulting. CMMI (Ahern, Clouse and Turner) is a process framework developed by the Software Engineering Institute (SEI) that helps to assess how mature a company is in developing software products. Different CMMI maturity levels (one to five) distinguish between different levels of maturity. The Fraunhofer Center has helped the company reaching CMMI Maturity Level three in 2007.

The customer can roughly be described as a software development company[18] focusing on web based software systems for government contractors. The company employs 36 people, of which about one third are serving as developers, one third are serving as web designers, and one third are serving as other staff. Multiple applications are developed at a time (about 5) using an agile development lifecycle (similar to SCRUM (Schwaber and Beedle)). Process conformance analysis was focused on two of their projects: Project J and Project F. The primary programming language used in the environment is C#.

[17] The Fraunhofer Center is an affiliate of the University of Maryland. Its mission is to transfer technology from research to practice. The study with professionals helped to contribute to this mission. More information about the work of the center can be found on: http://www.fc-md.umd.edu

[18] The name of the company, their projects, and their developers are sanitized for security reasons. Whenever particular projects or developers need to be pointed out terms such as "Project A" or "Developer C" will serve as replacement.

The three investigated software processes can best be described as guidelines. Guidelines are rules that developers should follow during development in order to improve quality characteristics of the software product. A typical guideline that was inspected is Architecture Conformance. This guideline recommends developers to adhere to one common project architecture by providing them with a set of architectural rules. The guideline does not define specific steps, or an order of steps. It requires the guideline to be followed throughout the software development life cycle.

	CTCD	Cont. Refactoring	Architecture Conf.
Step 1: Defining Conformance Templates	6.6.1:step1	6.6.2:step1	6.6.3:step1
Step 2: Violation Detection	6.6.1:step2	6.6.2:step2	6.6.3:step2
Step 3: Gathering Additional Insights	6.6.1:step3	6.6.2:step3	6.6.3:step3
Step 4: Process and Rule Improvement	6.6.1:step4	6.6.2:step4	6.6.3:step4

Table 29: Organization of Section 1.6

The following sections and subsections describe, for each of the practices, how the four steps of the conformance approach were performed. The reader can read these in two different ways (see Table 29):

To follow a particular practice the sections should be read in the order presented here.

To follow the four steps of the model the reader can go over the sub sections separately.

6.6.1 Collaborative Test Case Development (CTCD)

CTCD: Defining Conformance Templates

The guideline Collaborative Test Case Development (CTCD) requires that all developers in a project contribute to test case development. For the company, this guideline is important since it ties into one of their organizational goals. The goal defines that developers should continuously be trained in all core technologies. One of these core technologies is the ability to develop unit test cases for web applications.

Process Name	Collaborative Test Case Development (CTCD)	
Process Definition	All developers in a project should contribute continuously to the test case development.	
Process Focus	Training of personnel, increased program correctness	
Collected Data	SVN data provides us with information about which developers are actively involved in test case development (create and modify source files in a specific test directory).	
Version	**V1** (Sep 2009 – Feb 2010)	**V2** (Feb 2010 – today)
Violations	An active developer that has not contributed to test case development for a longer time: A developer who has changed at least 1 source code file (suffix:.cs) in the last 30 days but has not changed any test case files (files in folder tests) in the same time period.	An active developer that has not contributed to test case development for a longer time. A developer who has changed at least 10 source code file (suffix:.cs) in the namespace Core.* in the last 30 days but has not changed any test case files (files in folder tests) in the same time period.

Table 30: Process Conformance Template (with different versions) for CTCD. Differences in the versions are highlighted in yellow.

The complete process conformance template can be found in Table 30. It includes two versions of violation detection that show how the rule was tailored over time.

To detect violations against the guideline the already collected data in the Subversion repository can be used. The data stored in the repository provides information on which parts of the system have been changed and who changed them. The structure of each of the companies' development projects demands that the developers store test cases in a particular folder (in the companies' terminology: "the Tests namespace"). Hence, it can be concluded that only developers adding or modifying files in this folder work on developing test cases for the application.

CTCD: Violation Detection

CodeVizard was used by implementing an extension for detecting the violations as described in Table 30. The extension allows printing a list of authors and the number of changes they made to test and implementation classes in the last 30 days.

```
Sun Apr 04 04:13:59 EDT 2010
Sensor CTCD (Collective Test Case Development) - V2 -
Violation 1

Results:

(!) Author: dm  scChanges:43 tcChanges:0

    Author: jj  scChanges:27 tcChanges:11

    Author: jb  scChanges:0  tcChanges:0
```

Figure 37: Violation detection results as printed by CodeVizard for Project J on 4th, April 2010

An example output from the CodeVizard extension is shown in Figure 37. One can read from the figure that three developers worked on the codebase in a 30 day period ending on April, 4th 2010. The first developer (dm) made a total of 43 source code (sc) changes (i.e. changes on implementation classes) and no test case (tc) changes. Thus, the developer violates the rule of continuously developing test cases. The second developer follows the guideline by making source code and test case changes. The last developer (jb) made no changes to test cases, and did not make any changes to source code files. The developer does therefore not violate the guideline[19].

Developer	Nov09	Dec09	Jan10	Feb10	Mar10	Apr10	May10	Jun10	Jul10
dm	31/34	7/0	10/3	1/1	43/0	38/5	1/3	20/3	7/0
jj	31/8	4/4	1/0	1/0	27/11	0/2	3/4	65/34	7/3
kb	-/-	-/-	-/-	-/-	-/-	-/-	0/0	0/0	0/0
jb	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
de	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-	0/0
al	0/0	0/0	0/0	-/-	-/-	-/-	-/-	0/0	0/0
cn	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-	0/0
af	0/0	-/-	-/-	-/-	-/-	-/-	0/0	-/-	-/-
rf	-/-	0/0	-/-	-/-	-/-	0/0	-/-	-/-	-/-
ms	18/1	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-

Table 31: Example results for CTCD for project J from November 2009 to July2010: underlined figures are violations against version 1, red figures are

The results for project J over time can be read from Table 31. The cells show if a particular developer (rows) violates the CTCD guideline in a given time frame of 30 days. The 30 days are counted from the end of a month (columns) on. Therefore, this table shows approximately if a developer violated the rule in each of the months

[19] One might wonder why the developer shows up in this list. This is because this developer changed code unrelated files that were also stored in the repository, e.g. documentation files and requirements specifications.

ranging from November 2009 to July 2010. The first figure in each cell states the number of source code changes. The second figure shows the number of test case changes. If a developer did not make any changes during the timeframe in the repository (e.g. no source code, test case, and any other file change) two dashes are shown (“-/-“). One can see that only three developers (dm, jj, ms) changed code parts. In most cases developers followed the rule, only in one instance (marked in red: March 2010, developer: dm) a violation could be detected.

Further the figure shows how rule tailoring affects the results of the violation detection. The violations according to the first version of the rule (see Table 30: V1) are underlined in the table.

CTCD: Gathering Additional Insights

The issues identified in the second step were brought up in monthly meetings with the project leads for the two projects and the process manager. Since there were not many violations this simple feedback process was used to judge the violations.

CTCD: Rule and Process Improvement

The process rule was changed one time (see Table 30: version 1 and version 2) to adjust for the number of false positives detected. The guideline itself was left unchanged.

Reason for changing the rule was that the first version of the rule was too strict since it required changing test cases even if only *one* single source code file was changed. These violations were judged by the process manager and developers as false positives because these changes are too small to necessarily require test case changes. The threshold for source code changes was readjusted in the second version: a violation will only be detected if developers are changing at least ten source code files, and no test case file.

Project	Identified violations when using V1 (last 12 months)	Identified violations when using V2 (last 12 months)
Project J	5 (4 false positives: 20% precision)	1 (100% precision)
Project F	3 (2 false positives: 33% precision)	1 (100% precision)

Table 32: Overall results of violation detection for CTCD and impact of rule improvement step.

Overall, two projects were monitored for a time frame of 12 months. The identified violations and precision measures for the two versions of the rules are presented in Table 32. The data shows that the second version of the conformance rule has better precision.

6.6.2 Continuous Refactoring (CR)

CR: Defining Conformance Templates

The second practice under investigation was a flavor of an already known one from the classroom studies: Continuous Refactoring. In this environment the technical lead of the company requires developers to follow object oriented (OO) design rules and to refactor code as soon as it becomes necessary. The rule definition in this case is rather vague since it is often up to judgment of an expert as to whether or not a system follows OO rules, such as information hiding or encapsulation.

One way to identify potential violations against the practice is to search for symptoms of process violations. In this case, if design rules are not followed and refactoring is not done then code might exhibit certain negative features, also known as code smells. To identify code smells in this work I could make use of previous research efforts. Code smells, first introduced by Fowler and Beck (Fowler and Beck) , are indicators for the misuse of, or flaws in object oriented design. Code smells point to refactoring opportunities. Metric based approaches to automatically detect code smells in software systems (Lanza and Marinescu) have been proposed and partly validated.

Process Name	**OO Rules and Continuous Refactoring**	
Process Definition	The design and implementation should follow the principles of good object oriented design. Refactoring should be performed continuously to adapt design to new requirements. Thus, the number of components with design flaws (e.g. code smells) in a system should be hold small.	
Process Focus	Maintainability, Understandability, Extendibility	
Collected Data	SVN data provides with code that can be used to detect Code Smells (indicators for bad object oriented design). At the moment we are able to identify God Classes with high precision and recall (confirmed through code smell study)	
Version	**V1** (Mar 2010)	**V2** (Apr 2010 - today)
Violations	1. A new true positive identified/verified God Class. 2. A God Class ratio (#God Classes / # All Classes) higher than 10%.	1. A new true positive identified/verified God Class. 2. A God Class ratio (#God Classes / # All Classes) higher than 5%.

Table 33: Process Rule for Continuous Refactoring

My assumption for defining violations against the process definition (see Table 33) is that if developers are not following OO rules and if they do not regularly refactor their code, then they will introduce new code smells (violation 1). In other words measuring a raise of code smells in a system can point to violations of the rule. This is an example of a violation using an indirect measure (e.g. a quality measure of the product) for inferring that a process has not been followed.

In order to detect code smells in this new environment one specific code smell was selected that seemed most promising for detecting a lack of refactoring activities. The code smell *God Class* describes classes that implement too much functionality and responsibility in a software system. God Classes are usually among the larger classes of the system and typically originate when developers are adding more and more functionality to one class. A typical refactoring strategy for resolving God Classes is to split the class up into multiple ones.

In an initial study it was evaluated that it was feasible to detect God Classes with high precision (71%) and recall (100%), based on the metrics approach by Lanza and Marinescu (Lanza and Marinescu). In detail, the judgments for about 80 classes by four developers of the company were compared to the results of the automatic approach. Specific details of the study can be found in the according conference publication (Schumacher, Zazworka and Shull).

The first violation states that whenever a new God Class is introduced a violation is detected. The second violation was defined as projects having more than 5% of God Classes (in the first version of the rule: 10%). This violation describes a maximum threshold that no project should exceed.

CR: Violation Detection

CodeVizard was used for identifying God Classes in projects J and F. CodeVizard allows to compute a wide range of object oriented metrics and allows composing them into code smell detection. To exemplify the results, the three figures below show parts of the violation detection process:

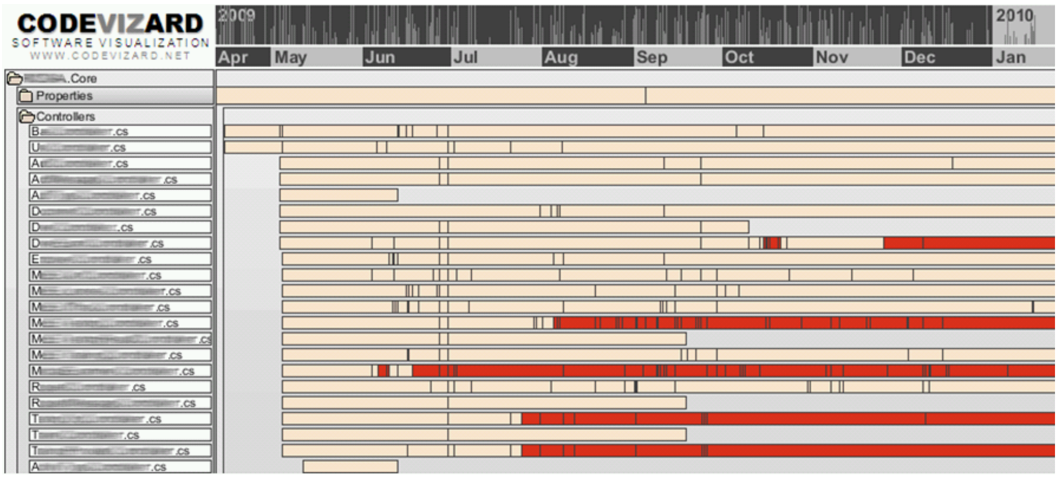


Figure 38: God Classes as indicators for violating continuous refactoring in project F

The above screen shot of CodeVizard (Figure 38) visualizes classes in namespace Controllers over a time period of 10 months (Apr 2009 – Jan 2010). The red sections in each of the classes' life lines show when a class became a God Class. For example, the top most class in the picture (class names were anonymized) became a God Class in October 2009 for a short period of time, and again in November 2009.

To inspect the overall trend of God Classes CodeVizard allows printing the number and percentage of God Classes over time.

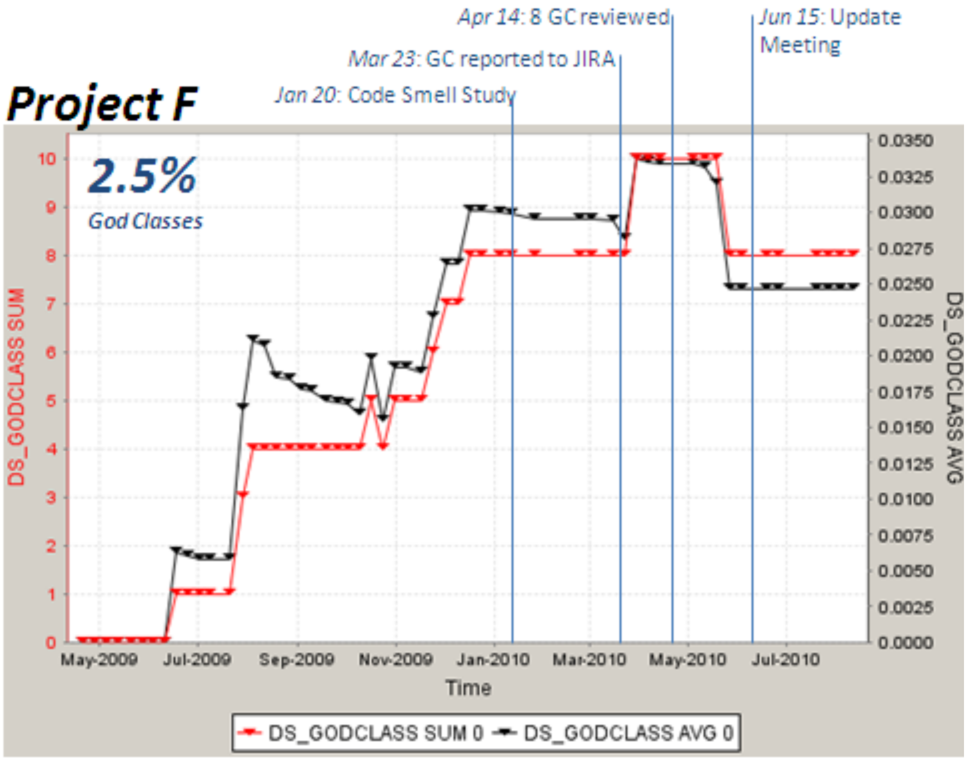


Figure 39: Number and percentage of God Classes in project F

Figure 39 shows the trend of God Classes in project F. The red line and scale on the left side of the graph show the total number of God Classes. The data shows that, from the beginning of the project until April 2010, the number of God Classes grows to a total of ten. Afterwards the number decreases again to a total of eight classes. Every time the number of God Classes increases a violation is generated according to the conformance rule. The black line and scale on the right side indicate what percentage of classes is affected. At the end of the analysis period (August 2010) about 0.025 (=2.5%) of all classes have the smell. One can immediately see that the second part of the conformance rule (violation 2: a God Class rate of greater than 5%) is never violated in project F.

Additionally, in this graph, events (such as meetings with the developers) are overlaid to illustrate the impact of reporting violations against the practice. The first intervention that focused on code smells was the initial study performed to validate the feasibility of the automated God Class classifier. All developers of projects J and F were subjects of this study. One can see that the linear growth of God Classes (from July 2009 to Jan 2010) stopped at this point, and that it was more or less stable from this point on. This behavior is not necessarily evidence of a causal relationship between raising the awareness of developers and the introduction of God Classes, but shows some amount of correlation between the two.

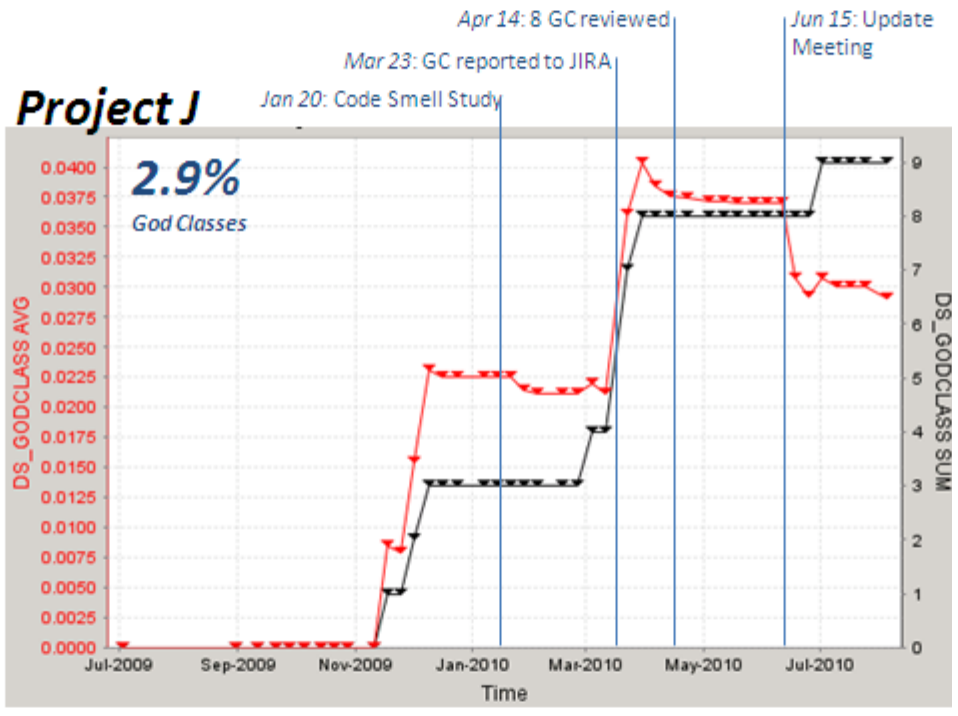


Figure 40: Number and percentage of God Classes in project J

The same graph is shown for project J in Figure 40. The total number of God Classes increases from November 2009 on to a total of nine God Classes. The percentage of

God Classes first raises to a peak of 4.1% and declines, from April 2010 on, to 2.9%. This shows that more and more classes were added to the system and these new classes do not have the code smell. One can argue that this trend is a positive one (even if the total number increases slightly).

The impact of interventions is not as visible as in project F. After the initial study that trained developers in detecting God Classes a steep raise of classes with the smell is visible in March 2010. The percentage of God Classes decreases after the second intervention. This intervention reported the existing God Classes to the developers using their companywide bug tracking system (JIRA[20]). These classes were then individually reviewed by the developers (the review process is given in later sub section). The decrease of percentage of God Classes might be a delayed effect of the interventions that reminded developers of the importance of the practice.

CR: Gathering Additional Insights

In order to inform developers of new God Classes in their project and to get feedback on the validity of the classes a feedback process was created. The process is described in the following figure.

[20] www.atlassian.com/software/jira/

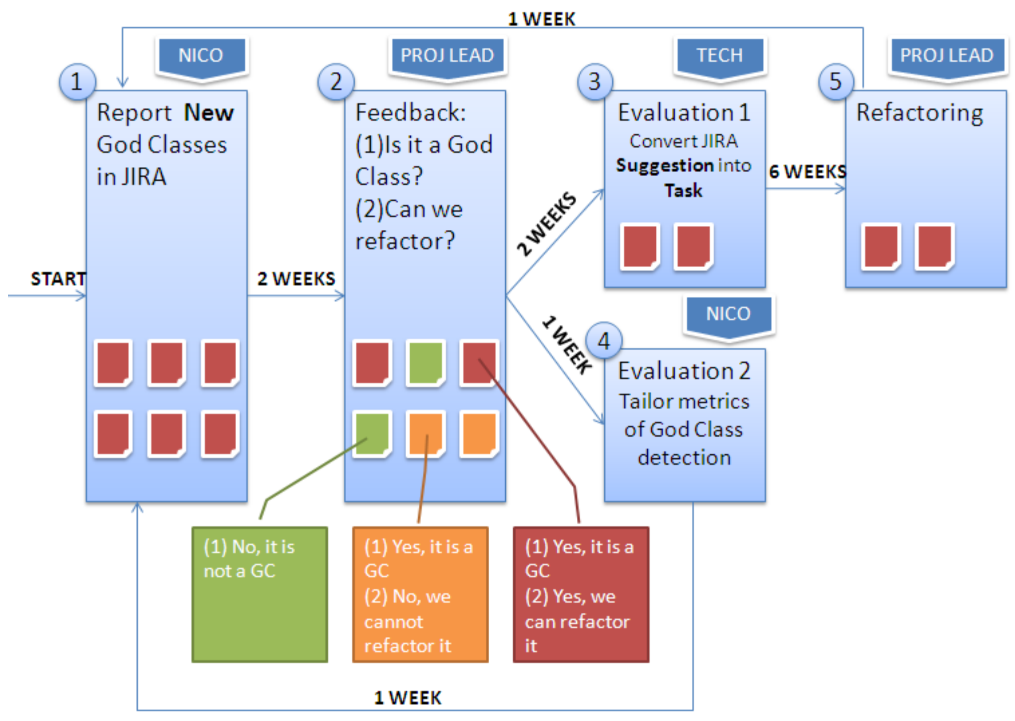

1 WEEK
NICO
PROJ LEAD
TECH
PROJ LEAD
1
Report New God Classes in JIRA
2
Feedback: (1)Is it a God Class? (2)Can we refactor?
3
Evaluation 1
Convert JIRA Suggestion into Task
5
Refactoring
6 WEEKS
2 WEEKS
START
2 WEEKS
1 WEEK
NICO
4
Evaluation 2
Tailor metrics of God Class detection
(1) No, it is not a GC
(1) Yes, it is a GC (2) No, we cannot refactor it
(1) Yes, it is a GC (2) Yes, we can refactor it
1 WEEK

Figure 41: Feedback process for new violations against the conformance rule

As explained earlier, to report new God Classes the companies' bug tracking system (JIRA) was used. As the first step of the process (Figure 41: step 1) all new God Classes were reported as separate issues in JIRA and were assigned to the project leads. The JIRA issue requires the project lead to review the class and to answer two questions:

1. Do you consider the class as a God Class?
2. Can it be refactored?

The three possible outcomes are shown on the bottom of the figure. The first question will help to understand if the right classes are identified by the automatic approach and will, in the long run, allow optimizing the code smell detection algorithms (e.g. the metrics and thresholds).

In the third step of the process, the technical lead of the company reviews the class and judgment of the project leads. He converts the issues into refactoring tasks that have to be completed by the project leads (or developers of the project) in step 5.

CR: Rule and Process Improvement

For Continuous Refactoring the accurate detection of refactoring opportunities (e.g. God Classes) is the primary objective of this step. For God Classes the initial study showed that it was possible to identify these classes in a subset of all classes of the system with 71% precision and 100% recall. In other words, all God Classes could be found, but some of the identified classes turned out to be false positives (29% false positive rate).

6.6.3 Architecture Conformance (AC)

AC: Conformance Rule Definition

One of the companies' goals is to employ a standardized architecture across all database driven web applications. The common architecture should help to increase maintainability and make it easier to switch developers across projects (e.g. decrease risks when loosing development personal). Further, it should help to identify parts that are used by all projects, and can be outsourced into a common companywide code library. The code library should help to increase the correctness of the application (since commonly used code can be tested more thoroughly), and the productivity within the project (since commonly used code does not have to be reinvented in each project).

At the time of executing Step 1 of the conformance approach, this architecture was not made explicit. That means it existed in the minds of the developers (mostly the project leads). Therefore, an initial effort had to be spent to make this knowledge explicit and to agree on the appropriate architecture. The latter was necessary because not all developers had the same mental model of the architecture.

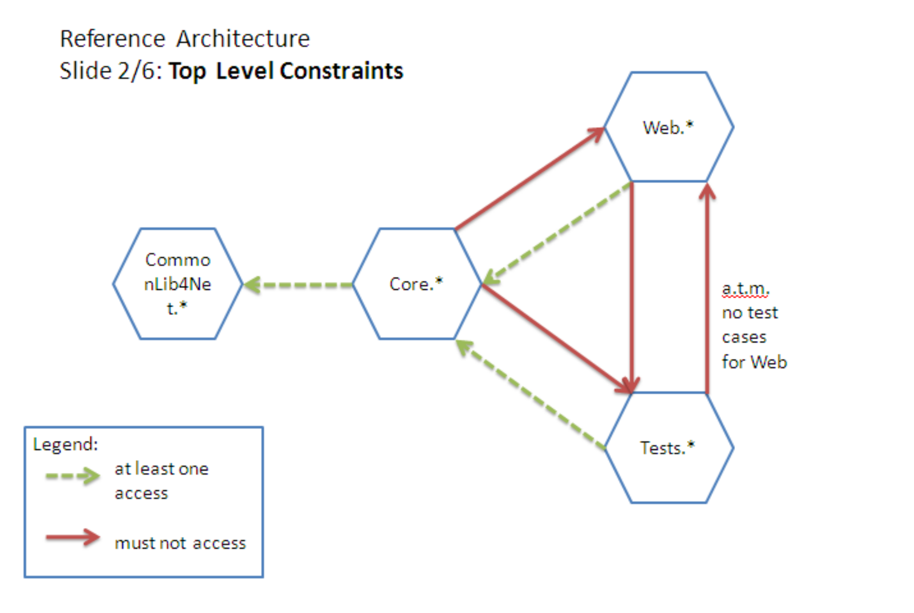


Figure 42: Excerpt of the agreed reference architecture. Arrows in the figure represent rules that a project has to adhere to.

In three meetings with the companies' technical lead and four project leads a list of 43 architecture rules was defined with the help of software dependency graphs (Zimmermann and Nagappan). An example graph is given in Figure 42. The hexagons in the picture show the main components (i.e. C# namespaces) of a project: Core (contains the data model and DB access), Web (contains logic for web sites), Tests (contains test cases), and CommonLib4Net (refers to the common library used by all projects). The arrows between the components define access relationships. Green arrows suggest that at least one class in the namespace has to access the namespace on the other end of the arrow. For example, classes in the Core namespace should access one the common library (CommonLib4Net) at least one time. Red arrows indicated that classes within a namespace are not allowed to access the namespace on the other end of the arrow. For example, classes in Core are not allowed to access classes in the Web namespace.

These relations were further refined for all sub namespaces. Additionally, rules were created that express "existence requirements". Existence requirements state that a project is required to have certain namespaces. For example, each project following the standard architecture has to have Core, Web, and Tests namespaces. A selection of all architecture rules is presented in the conformance template in Table 34.

Process Name	**Architecture conformance (AC)**	
Process Definition	Database driven web applications should follow common project architecture and use common libraries. The architecture rules are given in the violation section of the conformance template.	
Process Focus	Maintainability (e.g. decrease of code duplicates) , Correctness (e.g. common architecture features are well tested), lower truck factor risk (e.g. avoids new, not understandable architectures and designs)	
Collected Data	SVN data provides us with information about file and directory names, as well as used features of the common architecture.	
Version	**V1** (Sep 2009 – Dec 2009)	**V2** (Dec 2009 – today)

Violations	Project not having the following namespaces:
	`(1) Web`
	`(2) Web.Util`
	`...`
	`(11)Test.Models`
	`(12)Test.Properties`
	`(13)Test.Util`
	Project not satisfying the following access relationships (“1+” means: at least one access; “0!” no access allowed):
	`ID FROM TO ACCESS`
	`(14)Web.* Core.* 1+`
	`(15)Web.* Test.* 0!`
	`...`
	`(41)Core.Util Core.Controllers 0!`
	`(42)Core.Util Core.Models 0!`
	`(43)Core.Util Core 0!`
		Project not having the following namespaces:
		`(1) Web`
		`(2) Web.Util`
		`...`
		`(11)Test.Models`
		`(12)Test.Properties`
		`(13)Test.Util`
		Project not satisfying the following access relationships (“1+” means: at least one access; “0!” no access allowed):
		`ID FROM TO ACCESS`
		`(14)Web.* Core.* 1+`
		`(15)Web.* Test.* 0!`
		`...`
		`(32)Core.Utils Core.Models 0!`
		`(33)Core * 0!`
		`(34)* Core 0!`

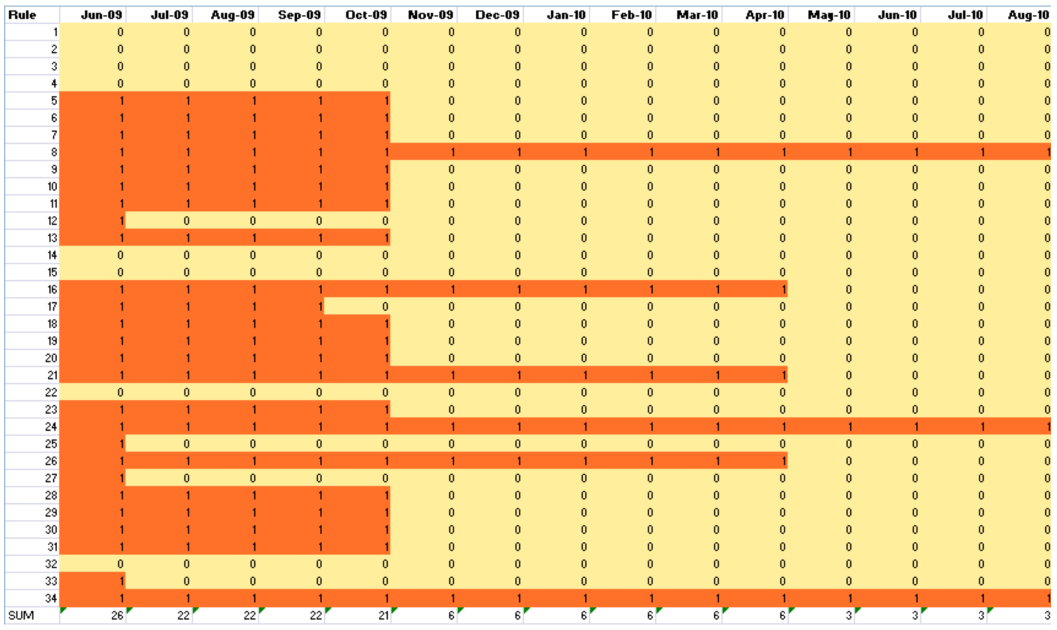
Table 34: Process Conformance Rule for Architecture Conformance.

AC: Violation Detection

In order to detect violations against the set of architecture rules, defined in the conformance rule in Table 34, another extension was implemented into CodeVizard. This extensions is responsible for extracting access relationships from C# code that is stored in the Subversion repository. In detail, it iterates over the C# classes of one version and decides for each pair of classes C1 and C2 if the classes access each other (and the direction of access). Accesses include: the use of a class (instanciation), method calls, inheritance relationships, and use of parameters of a class (including static parameters). Accesses are restricted to static relationships, e.g. dynamic bindings (through reflection) are not considered.

The second responsibility of the extension is to apply the ruleset on the extracted relationships. For rules that express a desired access relationship (in Table 34: “1+” rules) the extension checks whether at least one subclass of a namespace access the desired namespace. If this is not the case, a violation against the rule is detected. For rules that express undesired access relationships (in Table 34: “0!” rules) the extension checks if no sub class of the first namespace accesses the namespace of the second namespace. If at least one access realtionship can be found then a violaton is detected.

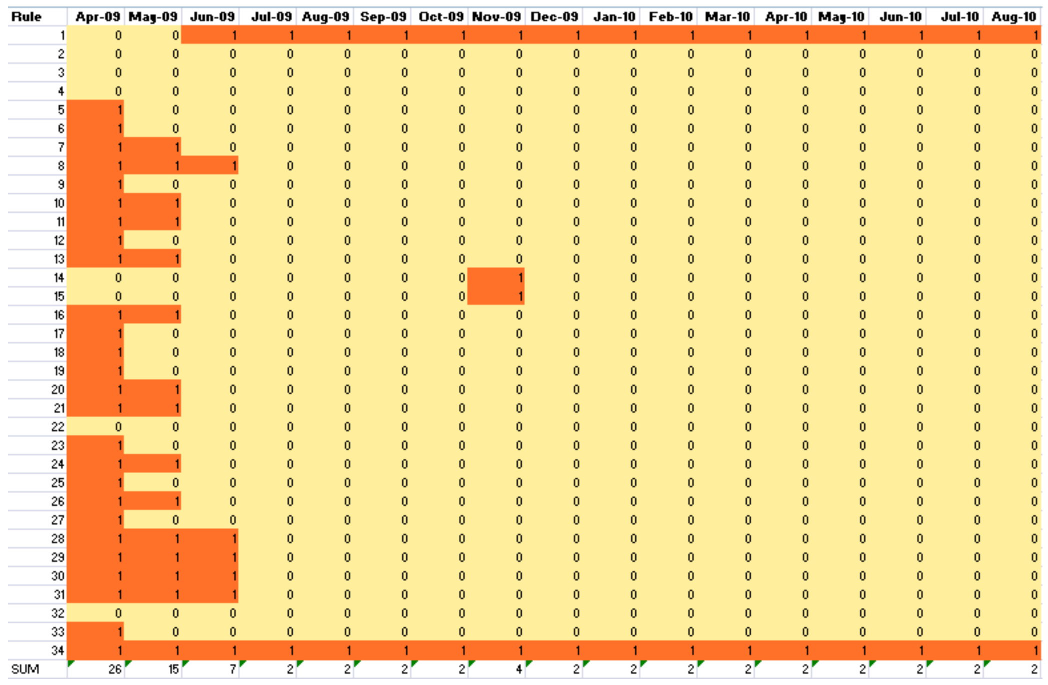
The extension further allows to print the results in csv format for Excel import.



Rule	Jun-09	Jul-09	Aug-09	Sep-09	Oct-09	Nov-09	Dec-09	Jan-10	Feb-10	Mar-10	Apr-10	May-10	Jun-10	Jul-10	Aug-10
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
6	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
7	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
8	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
9	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
10	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
11	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
12	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
17	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
18	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
19	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
20	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
21	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
24	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
25	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
27	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
29	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
30	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
31	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
33	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
34	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
SUM	26	22	22	22	21	6	6	6	6	6	6	3	3	3	3

Figure 43: Process Violations against AC for Project J

The identified violations can then be visualized over time as shown in Figure 43. Each of the rows in the figure represents one of the AC rules (from V2 in Table 34). The columns show months. Each cell indicates if a process violation at the beginning of the month could be found (dark red cells with figure "1") or if no violation was detected (light cells with figure "0"). The sum of violations can be found in the last row (SUM). The data from project J indicates that early in the project a lot of violations were detected. 26 out of 34 rules were violated. In November 2009 most of these violations were resolved. Additional three violations were resolved in April 2010. At the last point of measurement the project violated only three of the formulated rules.



Rule	Apr-09	May-09	Jun-09	Jul-09	Aug-09	Sep-09	Oct-09	Nov-09	Dec-09	Jan-10	Feb-10	Mar-10	Apr-10	May-10	Jun-10	Jul-10	Aug-10
1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
16	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
33	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
34	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
SUM	26	15	7	2	2	2	2	4	2	2	2	2	2	2	2	2	2

Figure 44: Process Violations against AC for Project F

The indentified violations for project F are visualized in the same manner in Figure 44. At the beginning of the project (April 2009) most rules are violated (26 out of 34).

The violations are resolved very early in the project. Four months into development, only two violations still exist. Only in November 2009, for a short time, two more violations can be identified. At the end of the analysis period, two architectural violations reside in the code.

AC: Gathering Additional Insights

The identified violations were discussed with the technical lead and the two project leads of project J and project F in meetings. First, the amount of initial violations in the projects could be explained by the project leads. They said that since no or only little code is present at the beginning of a project, all rules that either expect certain namespace or access relationships to exist are violated. Therefore, the severity of violations at project start can be judged as negligible.

The meetings further helped to inform the project leads of violations that could be resolved in later versions of their software systems. In particular, for project J, three violations were resolved in April 2010 that were reported in meetings. For project F, two violations were reported (in November 2009) and could be resolved immediately.

AC: Rule and Process Improvement

The process rules for architecture conformance were tailored one time (from V1 to V2). This became necessary since not all project leads agreed after a first round of analysis to the defined rule set of 43 architecture rules. In particular, once a violation was detected they argued that some of these rules might be applicable to only a subset of projects, but should not be part of the common rule set. Therefore, rules that turned out not to be applicable in general, were deleted from the list (the second version contains a smaller set of 34 architecture rules).

This change is an instance of tailoring a process, since the process definition was changed. The detection of the rules was accurate in all cases (precision 100%). In future, the company considers applying two rule sets to each of the projects. One general rule set as presented here, and one customized for each of their projects that encodes specifics of the project's architecture.

Overall, for both inspected processes the number of violations could be reduced over time from initial 26 violations to 2 (project F) and 3 (project J) violations.

7 Validation of Research Questions and Hypotheses

The four studies presented in the previous chapter provide different levels of supporting evidence for the six research questions and four hypotheses. This chapter elaborates on these findings. First, in section 7.1, evidence for the research questions is summarized. Second, in section 7.2, evidence for the research hypothesis is presented. After discussing the evidence found, section 7.3, will explain the threats to internal and external validity. Finally, in section 7.4, open questions and future work will be presented.

7.1 Validation of Research Questions

	RQ 1: Feasibility	RQ 2a: Useful insights	RQ 2b: Agreement	RQ 3: Rule Improve-ment	RQ 4: Process Enactment Improve-ment	RQ 5: Rule Transfer	RQ 6: Overhead Cost
FEAS	ComP: +	ComP:+	ComP: N/A	ComP: N/A	ComP: N/A	ComP: N/A	N/A
	CorP: +	CorP:+	CorP: N/A	CorP: N/A	CorP: N/A	CorP: N/A	
CROOM1	PS: +	PS:+	PS: +	PS: 0	PS: N/A	PS:N/A	12.1%/ 6.25%
	CR: 0	CR: 0	CR: +	CR: -	CR: N/A	CR:N/A	
	TDD: +	TDD: +	TDD: +	TDD: +	TDD: +	TDD:N/A	
CROOM2	PS: +	PS: +	PS: +	PS: +	PS: -	PS: +	9.1%/ 6.26%
	CR: 0	CR: 0	CR: +	CR: -	CR: N/A	CR: +	
	TDD: +	TDD: +	TDD: +	TDD: +	TDD: +	TDD: +	
	CP: +	CP: 0	CP: 0	CP: 0	CP: -	CP: N/A	
PROF	CTCD: +	CTCD: +	CTCD: +	CTCD: +	CTCD: +	CTCD: N/A	3.4%/ 1.01%
	CR: +	CR: +	CR: 0	CR: 0	CR: 0	CR: 0	
	AC: +	AC: +	AC: +	AC: +	AC: +	AC: N/A	

Table 35: Overview of evidence for the six research questions.
The indicators should be read the following way:
“-“ negative evidence;
“0” neither negative nor positive evidence;
“-/+” mixed evidence;
“+” positive evidence;
“N/A“ no evidence collected

Evidence for the six research questions is summarized in Table 35. The table gives details on how each study (rows) can support the different questions (columns). When necessary, processes are further given in each cell of the table. A “+” in a cell indicates that positive evidence could be found for in a specific study for a specific research question. A “0” indicates that neither positive nor negative evidence could be found (no evidence), or that the results are pointing in both directions. In this case the data provides no clear support for a “yes” or “no” answer. A “-“ sign in the cell indicates that the evidence supports a “no” answer to the question. The “N/A” value represents cases were no evidence could be collected in the given study.

7.1.1 RQ 1: Feasibility

The first research question asked if the approach can be used to find process violations using minimal intrusive methods. Almost all studies and processes provide positive evidence for this research questions. In all cases the process conformance approach and template could be used to translate existing, realistic software processes into the template and to define a set of violations using mostly existing project data. In most cases this data was stored in software repositories that were used to coordinate development activities among a group of developers. In studies *CROOM1* and *CROOM2* a small amount of manual data was collected through subversion commit templates. Further, in almost all cases the method could help to identify real process violations. That means, for each process (except Continuous Refactoring in the classroom studies) at least one true positive violation could be identified. Therefore, the studies provide a large body of evidence that the approach is feasible and promises to be applicable to a large set of software development processes that are applied in practice in classroom and in professional environments. For the Continuous Refactoring practice baselines could be build that describe how often refactoring activities are expected but no violations could be identified so far in the studies. Overall this can help to define stricter violation rules in future.

It should be again noted that the processes were not picked for investigation, but were the ones available and chosen by others (e.g. the researchers designing the XP course and the software managers in the professional environments). To some extend it can be argued that this process comes close to a random selection from the real population of applied processes in the field.

7.1.2 RQ 2a: Useful Insights

The first part of the second research questions asks if the detecting process violations is actually useful and provides valuable insights. As stated in the earlier chapter, valuable insights contain information on problems with the process definition, the application of the process, the characteristics of the violations, and the measures of those violations. These insights can even contain valuable information on how to design potential changes to the process.

For two of the processes one can argue that this goal could not be reached fully. For the Continuous Refactoring practice (CR) in the classroom environment baselines could be build that to some extend describe how often code refactoring should be part of change activities. However, many of the above described insights are missing. For example, even if it could be shown that refactoring ratios of 20% are below a desired ratio it was not yet possible to identify problems with the definition or application of the process, or to design potential changes to improve conformance. One valuable insight gained was that the initial violation definition (see Table 9: no refactoring during the whole project, or only one refactoring in a single stage) is too weak. The same holds for the Communication Practice. Violations showed that developers forgot to update their Skype statuses, or that they provide incomplete information. Again, identification of causes and solutions to improve conformance are missing.

For the other seven processes useful insights could be gained from the violations. For the Completion Process (ComP) I could show that developers deviated from the plan and that there were steps missing that require documenting these deviations. For the Correctness Process (CorP) I showed that an additional step in the code review phases could help to retest changed components to lower the risk of faulty code. For Pair Switching (PS) insights could be gained that show that the frequency of switching pairs is an essential variable, and that switching pairs too often bears conformance problems.

For Test Driven Development (TDD) the results indicate that novice developers perceive this practice as very difficult, especially when they work with previously unknown technologies (i.e. Android). The classroom teams had to be reminded and forced in some situations to develop test cases. Even if this result might not hold in more mature environments, results indicate that TDD requires discipline and control to be followed.

For the Collective Test Case Development (CTCD) practice violations could be identified that helped providing developers with instant feedback. In the two identified cases of process violation these developers adhered to the process after violations were reported to them. Due to the low number of violations it was not necessary to change the definition of the practice.

For Continuous Refactoring (CR), when applied in PROF, it was shown that identifying code smells in the professional environment indeed points in many cases to missed refactoring opportunities. Developers perceived these insights as useful and based on these observations a new process could be defined that includes the identification and report of code smells. When compared to the results in the classroom studies, code smells are more promising in identifying violations against CR than measuring refactoring ratios (i.e. number of refactoring per number of changes). However, when I applied the code smell detection to the small classroom applications, then no code smells could be identified in the rather short term of development. One explanation for this is that the specific code smell used, i.e. God Classes, is less apparent in small application than in larger ones. Therefore, identifying violations against CR in small applications will either require a different set of code smells or a different method overall.

For Architecture Conformance (AC), rules could be effectively built and applied that point to violations against standard architecture rules. The identified violations provided insights for managers and developers. Analysis of violations over time helped managers understand that, at the beginning of a project, many rules are violated. But this is understandable because many parts of the architecture were not built yet. When developers were informed of violations they could effectively correct most of them.

7.1.3 RQ 2b: Agreement

The second part of the second research questions asks if the measured conformance matches the perceived conformance of the developers. In the classroom study, end-of-study questionnaires were used to assess this question. In almost all cases results from the questionnaire matched the ratio of identified violations. For example, for often violated practices such as TDD, developers said that they only followed the practice in half or less of the time. One exception for perceived conformance was the Communication Practice. Developers believed that they followed this practice in most or all times. However, violations were identified for both development groups that show that the practice was violated in 9 out of 20 times for the first group and 3 out of 9 times for the second group. One possible explanation could be that developers were not aware of violating the practice, e.g. when they forgot to maintain they Skype status message.

In the professional environments I was able to receive positive feedback from the developers and manager on reported violations. This provides to evidence that that reported conformance issues correlate with actual occurring ones.

7.1.4 RQ 3: Rule Improvement

The third research question asks whether the rules for detecting process violations can be iteratively improved and tailored to the environment. In 5 out of 10 cases rules had to be tailored and therefore satisfy the iterative approach of the model.

For TDD several technical process issues that were not understood initially required tailoring. For example, some specifics, such as which java compilation units were required to be tested, but were not understood completely at the necessary level of detail at the beginning of the project. The benefit of the iterative character of the approach was demonstrated by the analysis of false positives leading to an improved detection of violations.

For Architecture Conformance rules describing the expected common architecture needed tailoring after applying them to a set of projects. In this case the violation detection was accurate but the process definition needed to be improved to fit the collective mental model of the companies' common software architecture. This instance of tailoring provided evidence that the approach allows for tailoring through evolving the process definition.

Collective Test Case Development required tailoring to account for developers that are only changing an insufficient number of components during an analysis period. As with TDD, the tailoring affected the way how violations are detected.

The remaining 5 processes that did not yet go through the improvement step can be categorized in the following way. For three processes (Pair Switching, Communication Practice and Continuous Refactoring in PROF) the initial process violation detection strategies proved effective from the beginning on. That means the initial guess how to identify violations did not require to be changed. Therefore these processes are not necessarily a "no" answer to this research question since they did not require tailoring. For the two processes investigated in the first study (*FEASIBILITY*) no tailoring was done due to the nature of the a posteriori study. The last process (Continuous Refactoring in *CROOM1* and *CROOM2*) was not yet tailored because insufficient evidence was collected to support that developers in the two projects did not refactor often enough. However, the results show that the initial violation definitions (see Table 9: no refactoring during the whole project, or only one refactoring in a single stage) are too weak and need tailoring.

7.1.5 RQ 4: Conformance Improvement

The fourth research question asks if process enactors improve their conformance when provided with feedback on process violations. In 7 out of 10 instances, process enactors were educated about violations at least one time during the time of each of the studies. Overall, the response to the feedback on process conformance varies.

For TDD, conformance could be improved after providing feedback. In *CROOM1* developers of team *Zeit* were made aware of their poor conformance and an improvement (increase of conformance level of 46%) could be measured in the next development iteration. However, even after being reminded, developers did not follow the process all the time. This also holds for *CROOM2* were the process managers made TDD a top priority after observing low conformance levels on the first day (see Table 20). The TDD observations suggest that if a process is considered as very important by the process manager it can be enforced to some extent on the enactors, even if it is hard to execute for them initially.

For Pair Switching in *CROOM1* and *CROOM2* the number of process violations increased towards the end of the study. Subjects argued in *CROOM2* that this process bears a problem because switching (and breaking up pairs) was done too often. Additionally, the process manager indicated that this was a problem of the process. The results suggest that process enactors will intentionally violate a process if they see a problem with the steps of the process (or the frequency of executing the steps).

This is an important insight that supports the theory that process enactors will tailor processes automatically and intentionally if the effectiveness and applicability of the process is questioned by the enactors. In the case of Pair Switching applied in non-distributed environments it could be further shown that the tailoring towards less switching did not have a negative effect on one of the process' goals: providing good collective code ownership. Therefore, as an insight in defining the process for Pair Switching, one can recommend tailoring the practice towards less frequent switching of programming partners in the given classroom environment.

The communication practice was violated steadily and developers indicated later that they believed they followed this practice. This instance might indicate that process enactors are not always aware of violating the process. In other words, they did not intentionally modify the steps of the process. In the questionnaires developers indicated that they forgot to execute the process when unexpected events occurred. Therefore, one might conclude that some processes require better support (e.g. tool support) to remind developers to execute the process steps.

Collective Test Case Development in the study PROF was violated only twice during the analysis period of twelve months for projects F and J. In both cases, feedback was provided to the two developers violating the practice and in the following iteration they did not violate the practice. Therefore, reminding developers could have caused the change in behavior.

Continuous Refactoring (CR) was reported to the process enactors by identifying missed refactoring opportunities twice during the project. The first time enactors performed a code review to identify God Class code smells. The second time they were presented with a pre-selection of classes that were potentially infected with the smell (based on the automatic classification). Enactors had two weeks to review and comment on these potential violations. In the timeframe between the initial study and the end of the review phase the relative number of God Classes increased in both systems (see Figure 39 and Figure 40). Increasing the awareness of violations against CR did not show an *immediate* effect. After the end of the review, in both systems, the relative number of God Classes decreased. This points to the fact that the newly developed code contained less God Classes. This can be an effect of increasing awareness of God Classes in the system: developers are introducing less God Classes than before. Overall the results are mixed and future analysis is necessary to provide more insight whether refactoring opportunities are missed less often than before and therefore that CR is followed more closely.

For Architecture Conformance, process enactors were made aware of their violations by providing details on the violations through the project's bug tracking system (i.e. JIRA). Developers resolved the outstanding issues or commented on the validity of the architectural rules. Overall this feedback mechanism helped to decrease the number of architectural violations and newly introduced violations in both projects over time. This positive impact could be caused by the feedback provided to the developers.

7.1.6 RQ 5: Rule Transfer

The fifth research questions asked if a new project in either the same or a different environment can make use of previously defined and tailored process templates and violation detection mechanisms.

For the three processes that were investigated in the second classroom study (*CROOM2*) the tailored rules from the first study (*CROOM1*) could be indeed used as a starting point. For Pair Switching the rules did not require further tailoring in the second study. Continuous refactoring was applied as in the first study. Last, TDD was used as in the first study and further tailored towards the changed application technology (i.e. Android) and developer behavior (i.e. not following the recommended naming convention) in the second study.

For transferring rules from one environment to another environment (e.g. from classroom to professional) no evidence can be yet presented that shows this to be feasible. The one practice that was investigated in two environments (i.e. classroom and professional environments) was Continuous Refactoring. However, data collection methods differed in both scenarios significantly. Whereas the process enactors in the classroom setting reported on refactoring activities, the enactors in the professional environment did not report on it, but refactoring violations were measured indirectly through a product measure: code smells. The difference in measurement techniques also required a change in how violations were defined and identified. Thus, conformance rules were not just tailored versions of the previously applied versions. When comparing the two approaches of identifying violations against CR one can still learn important properties for future rule application. On the one hand, the code smell idea (i.e. in particular the God Classes) worked especially well in identifying violations in the mid-sized professional project. However, it could not help to identify violations in small projects, because no God Classes could be identified. Thus, project size is a key variable when identifying God Classes. On the positive side, the code smell approach did not rely on additional manual data collection (i.e. through commit templates) and is therefore less expensive and less prone to falsely reported data. On the other hand, measuring refactoring ratios (i.e. in *CROOM1* and *CROOM2*) could help to build support that the practice is executed by the developers. However, setting a fixed threshold for an expected minimal refactoring ratio turned out to be complicated. More research is necessary to understand if this model for detection is feasible.

7.1.7 RQ 6: Overhead Cost

The sixth research question asks about the overhead cost for the different roles of the approach. To answer this question an estimate of the costs was generated after the studies.

CROOM1

This four day development effort included overhead cost for process enactors through filling in subversion commit templates. Further, managers and enactors spent time discussing process violations in stand up meetings. The highest cost was spent by the process analyst since initial analysis models had to be created. Table 36 summarizes the estimated effort data.

Role	Estimated hours spent (and activities) per day on conformance issues /analysis	Estimated total hours spent during 4 development days on conformance issues/ analysis	Total work hours	Percentage of time spend with conformance issues/analysis
Process Analyst (1)	**8h**: creating models to detect violations, creating reports on violations, sending reports to process manager	**32h**	**32h**	**100%**
Process Manager (1)	**0.5h**: reading conformance reports (20 minutes) and discussing violations (10 minutes) in daily stand up meetings	**2h**	**32h**	**6.25%**
Process Enactors (14)	**0.5h**: ca. 10 times filling in svn template a day (10 * 2 minutes=20 minutes); 10 minutes discussing violations in daily stand up meeting	**2h**	**32h**	**6.25%**
Total (all actors: 1 analyst, 1 manager, 14 developers)	**15.5h/day**	**62h**	**512h**	**12.1%**
Total without Process Analyst	**7.5h/day**	**30h**	**480**	**6.25%**

Table 36: Effort Estimation for CROOM1

The cost measures presented in above table show the effort that was spent by the enactors during the studies, but does not include the cost of developing the tools used for data analysis (e.g. CodeVizard). However, the cost for the analyst included effort that was required to adapt tools. For example, a adaptation for the detection of violations against Test Driven Development is included in the cost.

Overall the data shows that about 12.1% of time of all enactors and 6.25% of developers and managers was spent to perform the conformance analysis in this study. Most time was spent by the analyst who had to build and adapt the model for violation detection and had to create reports (i.e. Word documents) that were send to the manager at the end of each development day.

CROOM2

For *CROOM2* the cost of the conformance analysis spent by the process analyst could be lowered since most (3 out of 4) conformance templates and detections mechanisms could be reused from the first study. Effort estimates are shown in Table 37.

Role	Estimated hours spent (and activities) per day on conformance issues /analysis	Estimated total hours spent during 5 development days on conformance issues/ analysis	Total work hours	Percentage of time spend with conformance issues/analysis
Process Analyst (1)	**4h**: creating models to detect violations, creating reports on violations, sending reports to process manager	**20h**	**20h**	**100%**
Process Manager (1)	**0.5h**: reading conformance reports (20 minutes) and discussing violations (10 minutes) in daily stand up meetings	**2.5h**	**40h**	**6.25%**
Process Enactors (15)	**0.5h**: ca. 10 times filling in svn template a day (10 * 2 minutes=20 minutes); 10 minutes discussing violations in daily stand up meeting	**2.5h**	**40h**	**6.25%**
Total (all actors: 1 analyst, 1 manager, 15 developers)	**12h/day**	**60h**	**660h**	**9.1%**
Total without Process Analyst	**8h/day**	**40h**	**640h**	**6.25%**

Table 37: Effort Estimation for CROOM2

The relative time spent in conformance activities could be lowered in this study to about 9% due to the existing analysis models. For process enactors and managers the effort spent was the same (i.e. 6.25%) as in the first study.

PROF

For the professional study effort estimates were created for the time period between January 2010 and August 2010. The data was derived from timesheets and meeting notes that were created during the analysis period. The number of process enactors was reduced to the ones that were mainly engaged in the development and the processes. Developers that were not fully included in the analysis cycle (e.g. developers who sat in meetings, but their project was not checked for violations) were excluded from the analysis. The timeframe was chosen since in that period all developers worked on a single project that was analyzed for conformance. Table 38 summarizes the effort data.

Role	Estimated total hours spent during from Jan 2010 to Aug 2010 on conformance issues/ analysis	Total work hours	Percentage of time spent with conformance issues/analysis
Process Analyst (1)	**16h/month=128h**: creating models to detect violations, creating reports on violations, sending reports to process manager and enactors, meeting with process manager and enactors	**128h**	**100%**
Process Manager (1)	**2h/month=16h**: defining process conformance templates, discussing violations, implementing improvements	**ca. 1280h**	**1.25%**
Process Enactors (3 most involved developers)	**1.5h/month=12h**: providing feedback on potential violations, meeting with process manager and analyst, performing code smell study	**Ca. 1280h**	**0.9%**
Total (all actors: 1 analyst, 1 manager, 3 developers)	**128h+ 16h + (3*12h)=180h**	**5248h**	**3.4%**
Total without Process Analyst	**52h**	**5120h**	**1.01%**

Table 38: Effort Estimation for PROF

The effort data shows that in the professional environment overhead cost was less than in the classroom study. A total of estimated 3.4% was spent including the process analyst, and 1.01% when excluding the process analyst, in process conformance activities.

7.2 Validation of Research Hypothesis

	Hypothesis 1: Precision > 50%	Hypothesis 2: Recall > 50%	Hypothesis 3: Precision Improvement	Hypothesis 4: Conformance Improvement
FEAS	ComP: N/A CorP: N/A	ComP: N/A CorP: N/A	ComP: N/A CorP: N/A	ComP: N/A CorP: N/A
CROOM1	PS: N/A CR: N/A TDD: N/A	PS: N/A CR: N/A TDD: N/A	PS: N/A CR: N/A TDD: N/A	PS: N/A CR: N/A TDD: +
CROOM2	PS: N/A CR: N/A TDD: N/A CP: N/A	PS: N/A CR: N/A TDD: N/A CP: N/A	PS: N/A CR: N/A TDD: N/A CP: N/A	PS: - CR: N/A TDD: + CP: -/+
PROF	CTCD: + (100%) CR: + (71%) AC: + (100%)	CTCD: N/A CR: + (100%) AC: N/A	CTCD: + (+73%) CR: + (+29%) AC: 0 (+0%)	CTCD: + CR: -/+ AC: +

Table 39: Results overview for the four research hypotheses
The indicators should be read the following way:
"-" negative evidence;
"0" neither negative nor positive evidence;
"-/+" mixed evidence;
"+" positive evidence;
"N/A" no evidence collected

7.2.1 H1: Precision > 50%

The first hypothesis states that for a given project and process it is possible to tailor the process violation detection mechanisms towards a precision of at least 50%. That means, in worst case the detection will report a maximum of 50% false positives (i.e. potential violations that turn out not be real violations).

Data for the hypothesis was collected in the professional environment the following way. Each violation was rechecked by either the process manager, or the process enactors, or both parties to make the final judgment whether the identified violations are indeed valid. For Collective Test Case Development and the second version of the violation detection strategy (see Table 30) the identified two violations were judged as valid violations. Therefore precision was in this case 100%. For Continuous Refactoring the classes that were marked as missed refactoring opportunities (i.e. classes with the God Class code smell) were reevaluated in the review process described in Figure 41. A precision of 71% was reached in successfully identifying the classes that require refactoring. For Architecture Conformance all reported violations were indeed violations against the defined architecture rules. The precision of the approach is therefore 100%. Overall, all processes and violation detections in the professional environment could be tailored towards having a precision of 71%.

When testing this hypothesis based on the three data points gathered in the professional environment one can formulate the null hypothesis as:

H0: true precision median <= 50%,

Let X be a random variable that indicates whether a detection for an arbitrary process is less or equal than 50% (i.e. X=0) or greater than 50% (i.e. X=1). X is then binomially distributed. The claim of the null hypothesis is that, overall, more processes will reach a precision of less or equal 50% than there are processes reaching more than 50% precision. In other words, P(X=0) >= P(X=1).

To calculate the probability (p-value) that given the data (3 data points indicating X=1) one is falsely rejecting the null hypothesis (e.g. error type I) one can use a binominal test:

P(3 observations X=1 | P(X=1)=0.5=P(X=0)) = 0.5 * 0.5 * 0.5 = 0.125

This means that with a probability of 12.5% one would reject H0 even if the true precision median is 50%.

Depending on the chosen significance level (α-level) one can or cannot reject the null hypothesis: when choosing an α-level of 0.2, as typically used in exploratory studies, one can reject the null hypothesis (i.e. 0.125 < 0.2). When using the more commonly used α-level of 0.05 one cannot reject the null hypothesis. Therefore the data does not provide statistical significant evidence for H1 on a statistical significant level of 0.05.

7.2.2 H2: Recall > 50%

The second research hypothesis states that for a given process and project, violations detection can be optimized to identify at least 50% of the real violations for one specific violation type. As explained in the original hypothesis, the recall of the approach can be estimated by taking a subset of items and manually identifying violations on them. Since this is an effort intensive task it could be only performed for one process in the professional environment. In this environment the process enactors examined a subset of all classes in a system for being God Classes (i.e. having missed to be refactored according to the Continuous Refactoring process). Thus, the specific violation type was the identification of God Classes, which is one out of many ways to identify missed refactoring opportunities. For two projects subsets of about 40 files were randomly chosen and examined. Detailed information on the experimental designs is presented in (Schumacher, Zazworka and Shull). The classes that were identified by the enactors were compared to the set of classes picked by the automatic solution (i.e. the God Class classifier). In this study all classes that were found to be God Classes by the enactors were also identified by the algorithm. Therefore, this experiment provides evidence for the hypothesis since a recall of 100% could be reached for identifying God Classes.

As with precision in section 7.2.1 this result does not hold when trying to reject the null hypothesis with a binominal test on a significance level of 0.05. The p-value of the test is 0.5. Therefore, the evidence presented for H2 is not statistically significant.

7.2.3 Precision Improvement

The third research hypothesis states that the four step iterative model will help improve the precision of the violation detection over time. Precision measures were assessed in study *PROF* and a positive change can be recognized throughout the set of processes.

For the first process, Collective Test Case Development (CTCD), the initial precision (see Table 30: version 1) was relatively low with 20% and 33% (see Table 32 for measurement results). For the tailored version (v2) precision could be improved to 100%. This is an average gain of +73% (out of a maximum of 100%).

For the second process Continuous Refactoring the initial precision of the approach was 71%. This is the precision of the God Class classifier identifying the right classes as God Classes. The classifier works on a set of software metrics as described in (Lanza and Marinescu) and (Schumacher, Zazworka and Shull) (i.e. a complexity metric: weighted method count; a coupling metric: access to foreign data; and a cohesion metric: tight class cohesion). If the three metrics are out of certain bounds (defined through thresholds) a God Class is detected. In (Schumacher, Zazworka and

Shull) it is shown that one can tailor the metric thresholds to achieve a precision of 100% for the data collected in the code smell study (while holding the recall constant at 100%). Therefore precision can be raised in this case from 71% to 100%.

For the last process, Architecture Conformance, all identified violations were indeed true positives according to the defined rule set from the beginning on. Thus, the initial precision was 100% and could not be further improved. The later change of the architectural rule set did not affect this behavior (see conformance rules in Table 34: V1 and V2). As explained in 0, the change of architecture rules is an instance of tailoring the process, and not a tailoring of how violations are detected.

As with precision in section 7.2.1 this result does not hold when trying to reject the null hypothesis with a binominal test on a significance level of 0.05. The p value of the test is 0.125 assuming improving and not improving precision is equally distributed with each of the events having a probability of 0.5. Therefore, the evidence presented for H3 is not statistically significant.

7.2.4 Conformance Improvement

The fourth hypothesis investigates in detail the impact of feedback on process violations; the H0 hypothesis is that feedback has no impact on future process conformance. The null hypothesis can be rejected if process enactors are improving their process conformance whenever violations are reported to them in the previous analysis cycle.

As already discussed for Research Question 4 in Section 7.1.5 the results depend on the process. For each process I analyzed whether after feedback to the process enactor's conformance was improved (positive evidence) or worsened (negative evidence).

Process Name	Study and Team	Time of Feedback	# Violations (violation rate) before	# Violations (violation rate) after	Evidence Based on # violations
TDD	CROOM1 Zeit	Before 3rd iteration	6 (85.7%)	2 (40%)	positive
	CROOM2 Notfallplan	Before 2nd iteration	9 (100%)	6 (55%)	positive
	CROOM2 Notfallplan	Before 3rd iteration	6(55%)	0 (0%)	positive
Pair Switching	CROOM2 Notfallplan	Before 3rd iteration	2 (25%)	4 (100%)	negative
	CROOM2 Notfallplan	Before 4th iteration	4 (100%)	7 (77%)	negative
	CROOM2 Notfallplan	Before 5th iteration	7(77%)	8(73%)	negative
	CROOM2 Spiel	Before 4rd iteration	1(33%)	5(63%)	negative
Communication Practice	CROOM2 Notfallplan	Before 2nd iteration	2	3	negative
	CROOM2 Notfallplan	Before 3rd iteration	3	0	positive
	CROOM2 Notfallplan	Before 4th iteration	0	3	negative
	CROOM2 Notfallplan	Before 5th iteration	3	4	negative
Continuous Refactoring	PROF Project J	Jan 20 2010	%GC:2.25%	%GC:2.75%	negative
	PROF Project J	March 23 2010	%GC:2.75%	%GC:3.75%	negative
	PROF Project J	Apr 14 2010	%GC:3.75%	%GC:3.70%	positive
	PROF Project J	Jun 15 2010	%GC:3.70%	%GC:2.9%	positive
	PROF Project F	Jan 20 2010	%GC:2.9%	%GC:2.8%	positive
	PROF Project F	March 23 2010	%GC:2.8%	%GC:3.45%	negative
	PROF Project F	Apr 14 2010	%GC:3.45%	%GC:2.5%	positive
	PROF Project F	Jun 15 2010	%GC:2.5%	%GC:2.5%	-
CTCD	PROF Project J	March 2010	1	0	positive
	PROF Project F	Dec 2009	1	0	positive
AC	PROF Project J	Apr 2010	6	3	positive

Table 40: Detailed results on conformance change after feedback to process enactors

In Table 40 results are presented for this analysis. Out of six processes in three cases conformance could indeed be improved all the time in 6 instances of providing feedback. The processes are Test Driven Development, Collective Test Case Development and Architecture Conformance. The probability of this happening by chance is p=0.015[21] assuming that negative and positive changes are equally distributed with having each a chance of 50%. Therefore, when combining the results of the three processes this result is significant on a level of 0.05. As a result one can reject the null hypothesis in this case. For two of the six processes (Communication Practice and Continuous Refactoring) the results varied. For Continuous Refactoring results got better (positive evidence) towards the end of the projects which might indicate a delayed effect of providing feedback. For the Communication Practice conformance got better one time but worsened in three cases. For one process, Pair Switching, feedback affected the results in a negative way, i.e., conformance declined in all cases after feedback. This is the process were enactors and managers identified micro process issues due to too frequent switching of pair programming partners. All these results are not statistical significant when tested with a binominal test on a significance level of 0.05.

Overall, one can conclude that the initial hypothesis that process conformance always improves when providing feedback was too simple and naive. When looking for reasons why for some processes this was the case and for others it was not, one can identify several explanations. For the processes that were considered as extremely

[21] P(all 6 instances are positive) = 0.5^6=0.015 (binominal test)

important and valid by the process managers, such as TDD, AC, and CTCD, positive change could be observed. Further for AC, developers helped to formulate the architectural rules and had therefore impact on the design of the practice. Processes that were either flawed (i.e. Pair Switching frequency) or not yet fully understood (i.e. Communication Practice) were in the set of processes that were not followed, even after feedback, and require being changed (or better supported) in future.

7.3 Threats to Validity

As with any study the presented four studies bear threats to internal and external validity. This section discussed these threats in detail.

7.3.1 Threads due to Internal Validity

Threats to internal validity describe problems with the experimental design that allow circumstances other than the treatment to influence the experimental results.

History

Historical events can change the outcomes of experiments independent of the applied treatment. In case of identifying process violations one can imagine that external events that were occurring in the different software projects will influenced the behavior of the process enactors. After all, software projects are typically executed in highly dynamic environments with changing parameters (e.g. changing requirements, changing deadlines, change of personal and task priorities). These dynamics are also reflected in the presented approach through its iterative character. Process definitions and violation detections might change over time with the dynamics of the project.

I investigated this threat in the last three studies by examining especially high amounts of violations in detail. Interviews with the managers and process enactors were used to do this. For example, in the classroom studies, process enactors tended to violate Pair Switching increasingly towards the last development day. My investigation of the cause through interviews with the process manager showed that the last day was usually the busiest one, where developers tried to complete a shippable product. Therefore this additional pressure could be identified as having an influence on process conformance. In the professional environment the high number of violations against architecture conformance could be explained by the initial build up of the software. In this early phase it was normal that not all architectural rules were adhered to yet.

In the professional environment analysis was performed less frequently than in the classroom environment. Several historical events did happen that may have influenced the behavior of the developers. For example, in between analysis reports developers had to finish releases of the software after six week development sprints. Further, both projects had phases, e.g. requirement elicitation phases, were only very little development was done: Table 31 shows months where all developers in the project (Project J) changed less than 10 source code files. Therefore, they could not violate Collective Test Case Development in that month.

In summary, dynamics in a software project will always be presents and cannot be ruled out as causes for changed behavior. The applied model reflects these dynamics through its iterative character.

Maturation

The threat of maturation describes the effects of subject behavior that changes over time due to learning effects. E.g. subjects might execute a process more precisely and effectively after some time because they increasingly learn and understand the process. As with the history threat this is behavior one will expect when process enactors, especially in classroom settings, execute a previously unknown process. One cannot rule out this threat for many of the studies and processes, especially the classroom studies that were of short duration. However, in some cases strategies were in place to limit the impact of maturing subjects. In the two classroom studies, subjects practiced Test Driven Development in a practical exercise during the course and before the actual study. This should lower the initial learning effect at the beginning of the experiment. Pair Switching and Continuous Refactoring were taught on a theoretical level before the beginning of the study. For Pair Switching, subjects tended to violate the practice towards the end of the study. Therefore, one can argue that a maturation effect did not affect the subject's conformance (i.e. one would expect poor conformance in the beginning of process application). Continuous Refactoring could have been influenced by maturation of subjects that learned over time how to refactor their own software.

In the professional environment the inspected processes were the ones that were already practiced for a long time in the organization. Therefore, learning effects are limited and should only be present for new development staff joining the organization. In this case new developers could increasingly become better at following a process and improve their conformance automatically without being impacted by conformance feedback.

Instrumentation

This threat describes changes in outcomes caused by changes in instrumentation, observers, or how scores are counted. For all of the studies and inspected processes instrumentation methods were held constant (i.e. did not change) during the process application. The only changes that were done during the studies are the changes to the methods being responsible for identifying process violations. These methods were tailored mainly based on false positives and it was shown that they were more effective in identifying the right violations. These changes were documented and are presented thought the various versions in the process conformance template.

One possible threat to the validity of the collected data was the partly self reported data used in the classroom studies. Refactoring activities were reported by the process enactors as part of the Subversion commit template. Subjects could have indicated that they did refactoring without really doing it, or they could have forgotten to indicate refactoring changes even when doing them. Evidence that this was not the case is presented through the background questionnaires that showed a general fit to the self reported data.

Statistical Regression and Selection of Subjects

The threats of statistical regression and selection of subjects describe biases caused the methods how subjects for the studies are recruited and assigned to the development groups. This threat was limited by the following actions. In the classroom study subjects were randomly assigned to the development teams. Further, subjects were regular students signing up for the XP course. In the professional environment the two chosen projects were the ones most active and important to the organizations at the time of analysis. There was no evidence that these projects are problematic (e.g. cause cost and time overruns). Therefore, the projects and developers represent a valid subset of the organizations projects and developers.

Experimental Mortality

Describes the threat of the loss of subjects and therefore biases towards characteristics of the group of subjects continuing in the study.

In the classroom experiment no subjects left the project during the study. In one instance one subject did not provide answers to the end-of-study questionnaire in the second classroom study. In the professional environment one developer left the organization during the time of the study. This could have impacted results in following way. An improvement in conformance could have been affected by the loss of a developer who conforms poorly to the process. For processes where developers could be tracked back to violations (e.g. CTDC) I could analyze that this developer was not responsible for the majority of violations. For the processes where this was not feasible (e.g. AC and CR) one cannot be sure if mortality could have affected the results.

7.3.2 Threats to External Validity

Threats to external validity are the treats that exist when generalizing the research results, or applying the methods in new environments (e.g. with a different population of developers, or different measurement variables such as different software processes).

One specific threat to external validity is the interaction effects of the selection of the subjects and the treatment. In other words, one needs to make sure that the subjects (i.e. the population) of the studies are representative. In Chapter 1 of this work it is outlined that the presented approach aims at investigating process conformance for two specific populations: the first population being students in classroom studies, the second one being professional developers in real world software projects. As for the first population, the studies conducted were using students (i.e. 29) on graduate levels in computer science in two universities in Germany. The population is in many ways representative for students used in typical classroom experiments. In particular the skills of the 29 students varied and the taught material differed from one university to the other. However, not all conclusions and behaviors might be reflected by the total population. In particular there could have been cultural influences that do affect the results and might not generalize. For example, observing students in other countries might lead to different results than the ones in Germany. Further, the studies do not include junior level students. Therefore not all of the results might generalize for the overall population of students used in classroom experiments.

As for the population in the professional environment, the two environments had a wide range of professional software developers with different levels of experience (ranging from less than one year to more than 7 years of development experience (Schumacher, Zazworka and Shull)). Further the development lifecycles were very different reaching from a very planned and static waterfall lifecycle in *FEASIBILITY* to a very iterative and agile lifecycle used in *PROF.* Further, the project size differed to a considerable amount from code in the 100kLOC range developed by ten developers (in *FEASIBILITY*) to smaller projects with two developers in the 10kLOC range in study *PROF*. Last, different programming languages were used for the different application types in the two studies with professionals.

Last a set of eight different and realistic software development processes has been investigated that share in common that they have not been set up for the purpose of conformance measurement but for producing software in classroom and professional environments.

In summary, the four studies conducted in the scope of this thesis might not rule out all threats to external validity but present evidence from valid and different enough environments to provide overall evidence that the presented approach is not limited to a specific environment or populations of processes and subjects.

7.4 Open Questions and Future Work

The presented work proposes and evaluates an approach to identify and inspect process violations for software processes. The four studies show that violations are apparent in software development processes and that various factors may affect process conformance. In some sense, this thesis presents a way to detect issues with a process definition and shows that issues exist, but provides only little support in how to resolve or prevent these issues.

Based on this observation several important research questions can be formulated that are worth investigating in future:

7.4.1 Impact of process variables on conformance

One of the important research questions is how can we help process designers in the future to create software processes that are one the one hand effective and on the other likely to be followed by process enactors. At this point in time, problems during the application can be identified but little is known about the "ingredients" of a successful process. Important process parameters to investigate in future are (and are not be limited to):

- The complexity of a process that might influence how well enactors can remember a process in detail.
- The likelihood of forgetting process steps due to a process design that allows skipping steps (e.g. if future steps do not depend on previous steps)
- The subjective perceived importance to the manager, the development team, or the individual developer.
- The influence of enactors helping to design the process, instead of *not* being involved in the process modeling.
- The way process descriptions are formulated and communicated across the development team.
- The role of tools that support process steps, or remind process enactors to execute process steps.

- How a process fits the environment.

7.4.2 Process tool support

One possible approach to improve conformance are tools that build upon this work and provide developers with more active feedback about process violations during process execution. Future work should investigate how to technically integrate these tools into existing software development environments (e.g. IDEs, such as Eclipse). For example, in an IDE such as Eclipse, TDD could be supported by guiding the developer through the steps of firstly developing the test case and secondly implementing the class itself. Further research is necessary to understand whether developer perceive these tools as useful (rather than interrupting) and if resulting process conformance is affected positively.

7.4.3 Relationship between process conformance and software quality

As explained in the first chapter, the motivation of this work builds on the assumption that not conforming to a process will also likely result in a product with decreased quality. For example, not following Test Driven Development will likely result in a product that is less correct, thus having more defects. Future research should therefore investigate two important questions. First, more evidence should be collected to confirm the assumption for a cause effect relationship between process conformance

and product quality. Second, it will be of interest how the two variables, conformance and quality, are connected: can we assume a linear relationship between the two of them? For example, does a decrease of 10% of process conformance lead to a, sometimes acceptable, decrease of 10% in product quality?

8 Conclusions

In this work I have presented a methodology for formulating, identifying and investigating process violations in the execution of software processes. The methodology consists of a four step iterative model, compromising templates and tools. A strong focus is set on identifying violations in a cost efficient and unobtrusive manner by utilizing automatically collected data gathered through commonly used software development tools, such as version control systems. The presented approach can be thought of as "*process testing*" and is powerful enough to show the presence of process violations but not their absence.

To evaluate the usefulness and correctness of the model a series of four studies have been conducted in both classroom and professional environments. A total of eight different software processes have been investigated and tested. The results of the studies show that the steps and iterative character of the methodology are useful for formulating and tailoring violation detection strategies and investigate violations in classroom study environments and professional environments, using minimal intrusive methods. The overhead cost of the approach is shown to be feasible with a 3.4% (professional environment) and 12.1% (classroom environment) overhead.

All the investigated processes were violated in some way, which emphasizes the importance of conformance measurement. This is especially important when running an empirical study to evaluate the effectiveness of a software process, as the experimenters want to make sure they are evaluating the specified process and not a variation of it.

Further, investigation of feedback about violations to the process enactors shows that conformance will not improve in all cases by merely presenting violations back to the process enactors. For example, if the process enactors see problems with the process, such as too frequent switching in the XP practice Pair Switching, they tend to continue violating the process. This is important feedback to the process designers.

And, some processes, such as Test Driven Development, sound simplistic in their definition but do require a fair amount of discipline to follow, at least by novice programmers. Test Driven Development was violated in four observed classroom projects at least 33.3% to 65.4% of the time and developers indicated this practice to be hard to follow.

For some processes potential improvement could be suggested: for Completion Process and Correctness Process, additional steps could have helped to keep the development plan up-to-date, or to retest and re-review components that were heavily modified after their being in the validation phase of the project.

Different approaches for detecting violations were presented for the agile Continuous Refactoring (CR) practice. It was shown that the identification of code smells, i.e. God Classes, led to very good precision and recall in identifying missed refactoring opportunities. Contrarily, measuring the relative amount of refactoring changes did not lead to a sufficiently strict enough measure for identifying violations against CR.

Future research is needed to investigate which aspects of a process promise to be “human compatible”, i.e. promise to be likely to be adhered to by enactors. Another future stream of research is to investigate what impact non-conformance has on the quality of the resulting software product, i.e. if not following a process leads indeed to decreased quality characteristics.

9 Appendix

9.1 Truck Factor Metric: Definition and Example

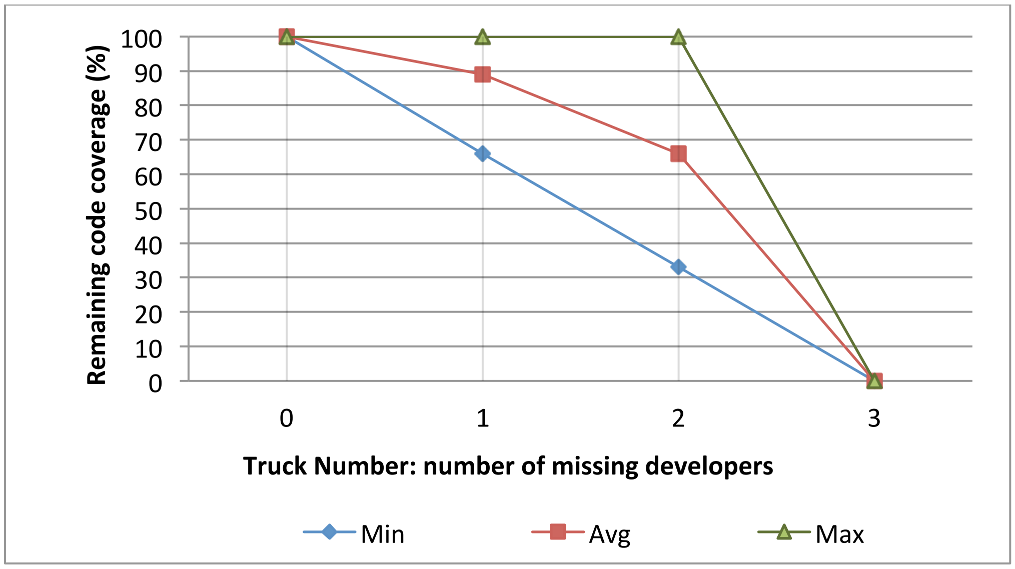
The truck factor has been defined by the eXtreme Programming Community as: “*The number of people on your team that have to be hit with a truck before the project is in serious trouble*”[22]. A high truck factor is desirable since it lowers the risk of project failure when losing personnel. Collective Code Ownership is the XP practice which helps in avoiding a low truck factor (Beck), situations where a small set of programmers owns a large part of the code base *exclusively*. To my knowledge, this measure has been proposed informally only so far and I am the first to derive this number by using information about code ownership from a code repository. The basic assumption of our analysis is that a source component (e.g. a Java file) in the repository is collectively owned by the developers who made changes to it (i.e. edited it).

		Truck Number: number of missing developers							
		0	1			2			3
File	Developer Set	{}	{A}	{B}	{C}	{A,B}	{A,C}	{B,C}	{A,B,C}
File 1	{A,B}	+	+	+	+	-	+	+	-
File 2	{B}	+	+	-	+	-	+	-	-
File 3	{A,B,C}	+	+	+	+	+	+	+	-
	Coverage (%)	100	100	66	100	33	100	66	0
	Min (%)	100	66			33			0
	Max (%)	100	100			100			0
	Average (%)	100	89			66			0

The table on top exemplifies a toy system with three developers (A,B,C) and three components (File 1, File 2, File 3). After extracting which developers modified which

[22] Truck Factor Definition: http://www.agileadvice.com/archives/2005/05/truck_factor.html, retrieved June 26th, 2010

components from the code repository data one can generate different scenarios where one can assume that a certain subset of developers has been "hit by a truck". For each component one can decide if the remaining developers have knowledge about it (light cells with "+" sign) or not (dark cells with "-" sign). A coverage number $cov_x(n)$ then describes the percentage of the components that would still be known by the remaining developers if n developers are absent. There are three types of coverage numbers: (1) the minimum (x = min), i.e. the worst case, is the remaining coverage when the set of developers with the most exclusive knowledge leaves, (2) the average (x = avg) coverage, and (3) maximum (x = max), i.e. the best case, is the coverage when the set of developers with the least exclusive knowledge leaves. The three coverage curves can be plotted as shown in the lower figure to visualize the *truck factor characteristics* of a project.



To define the truck factor (i.e. a single number) the manager has to define a threshold for code coverage. The truck factor can then be read from the chart by finding the intersection of the coverage number with one of the three curves. Typically, a project manager who wants to lower the risk of a project would be most interested in the worst case (i.e. x = min) curve since it shows the developers that are least dispensable.

Therefore the truck factor is defined as:

$$tf_{x,\,c} = \max \{n \mid cov_x(n) \geq c\}$$

For example, the worst case 60% coverage truck factor of our example would be:

$$tf_{min,\,60\%} = \max \{n \mid cov_{min}(n) \geq 60\%\} = 1$$

10 Bibliography

Zazworka, N. and C. Ackermann. “CodeVizard: a tool to aid the analysis of software evolution.” Poster Session, International Symposium on Empirical Software Engineering and Measurement. Bolzano-Bozen, Italy: AVM/IEEE, 2010.

Zazworka, N., V.R. Basili and F. Shull. “Tool Supported Detection and Judgement of Nonconformance in Process Execution.” 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM), 2009. 2009.

Zazworka, N., et al. “Are Developers Complying with the Process: An XP Study, ,.” 4th International Symposium on Empirical Software Engineering and Measurement (ESEM). Bolzano, Italy: ACM, New York, 2010. 1-10.

Zimmermann, T. and N. Nagappan. “Predicting defects using network analysis on dependency graphs.” 30th international conference on Software engineering. Leipzig: ACM, 2008. 531--540.

International Organization for Standardization. Information Technology - Software life cycle processes. 1995.

Williams, L.A. and L.A. Kessler. “All I really need to know about pair programming I learned in kindergarden.” Cummunications of the ACM 43.5 (2000): 108-114.

Ahern, D.M, A. Clouse and R. Turner. CMMI Distilled: A Practical Introduction to Integrated Process Improvement. Boston MA: Addison-Wesley, 2003.

Arisholm, E., et al. “Improving an Evolutionary Development Process.” EuroSPI. Pori, Finland, 1999. 940-950.

Bandinelli, S., et al. “SPADE: an environment for software process analysis, design, and enactment.” In Software Process Modelling and Technology (1994): 223-247.

Basili, V. R., F. Shull and F. Lanubile. “Building Knowledge through Families of Experiments.” IEEE Trans. Softw. Eng. 25, 4 (Jul. 1999), 456-473. (1999).

Beck, K. Extreme Programming Explained: Embrace Change. Addison Wesley, 1999.

Boudes, P. “Drug Compliance in Therapeutic Trials: A Review.” Controlled Clinical Trials, Volume 19, Issue 3, June 1998, Pages (1998): pp.257-268.

Broynooghe, R.F., J.M. Parker and J.S. Rowles. “PSS: A System for Process Enactment.” Proceedings of First International Conference on Software Process, IEEE Computer Society Press. 1991.

Brooks, F. P. “No Silver Bullet Essence and Accidents of Software Engineering.” Computer 20, 4 (Apr. 1987), 10-19. (1987).

Cook, J. E. and A. L. Wolf. “Discovering models of software processes from event-based data.” ACM Trans. Softw. Eng. Methodol. 7, 3 (July), 215–249., 1998.

—. “Software process validation: quantitatively measuring the correspondence of a process to a model.” ACM Trans. Softw. Eng. Methodol. 8, 2 (Apr. 1999), 147-176. (1999).

Cook, J. E. “Process discovery and validation through event-data analysis.” Tech. Rep.CU-CS-817-96. Department of Computer Science, University of Colorado at Boulder,Boulder, CO., 1996.

De Beer, H.T. and B. F. Van Dongen. “Process Mining and Verification of Properties: An Approach based on Temporal Logic.” On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005, volume 3760 of Lecture Notes in Computer Science. 2005.

Fowler, M. and K. Beck. Refactoring: improving the design of existing code. Addison Wesley Longman, Inc., 1999.

Gittins, R. and R. Hope. “A study of Human Solutions in eXtreme Programming.” in 13th Workshop of the Psychology of Programming Interest Group. 2001. Bournemouth: UK, 2001. 41-51.

Glass, R.L. “The Standish report: does it really describe a software crisis?” Commun. ACM 49, 8 2006 йил August: 15-16.

Humphrey, W. Managing the Software Process. Reading, Massachusetts: Addison-Wesley, 1989.

Huo, M., H. Zhang and R. Jeffery. “A Systematic Approach to Process Enactment Analysis as Input to Software Process Improvement or Tailoring.” Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific,pp 401-410, 2006.

—. “An exploratory study of process enactment as input to software process improvement. In.” Proceedings of the 2006 international Workshop on Software Quality (Shanghai, China, May 21 - 21, 2006). WoSQ '06. . New York: ACM, 2006.

Hochstein, L., et al. “Experiments to Understand HPC Time to Development.” CTWatch Quarterly, Vol 2, No. 4A (2006).

Johnson, P. M., et al. “Practical Automated Process and Product Metric Collection and Analysis in a Classroom Setting: Lessons Learned from Hackystat-UH.” In Proceedings of the 2004 international Symposium on Empirical Software Engineering (August 19 - 20, 2004). International Symposium on Empirical Software Engineering. IEEE Computer Society, Washington, DC, 136-144. 2004.

Kitchenham, B.A., et al. “Preliminary guidelines for empirical research in software engineering.” IEEE Transactions on Software Engineering 28 (2002) (8) (2002): 721–734.

Knight, J. C. and P. E. Ammann. “An experimental evaluation of simple methods for seeding program errors.” 8th international Conference on Software Engineering. London: IEEE Computer Society Press, Los Alamitos, CA, 1985. 337-342.

Kou, Hongbing, Johnson, Philip M. “Automated recognition of low-level process: A pilot validation study of Zorro for test-driven development.” In Proceedings of the 2006 International Workshop on Software Process. 2006.

Krishnan, M. S. and M. I. Kellner. “Measuring Process Consistency: Implications for Reducing Software Defects.” IEEE Trans. Softw. Eng. 25, 6 (Nov. 1999) (1999): 800-815.

Kroeger, T., D. Jacobs and C. Marlin. “Implementing Process Enactment within a Process-Centred Software Development Environment.” Proceedings of the Australian Software Engineering Conference. ASWEC. IEEE Computer Society, Washington, DC, 151., 1998.

Laitenberger, O., et al. “An experimental comparison of reading techniques for defect detection in UML design documents.” Journal of Systems and Software, Volume 53, Issue 2, 31 August 2000 (2000): 183-204.

Lanubile, F. and G. Visaggio. “Evaluating Defect Detection Techniques for Software Requirements Inspections.” ISERN Report no. 00-08. 2000.

Lanza, M. and R. Marinescu. Object Oriented Metrics in Practice. Berlin: Springer, 2006.

Leonhardt, U., J. Kramer and B. Nuseibeh. “Decentralised process enactment in a multi-perspective development environment.” In Proceedings of the 17th international Conference on Software Engineering. ACM, New York, NY, 255-264, 1995.

Nagappan, N. and T. Ball. “Use of relative code churn measures to predict system defect density.” ICSE. St. Louis, Missouri, USA: IEEE Computer Society Press, 2005. 284-292.

McCracken, M., et al. “A multi-national, multi-institutional study of assessment of programming skills of first-year CS students.” SIGCSE Bull. (2001): 125-180.

Mishali, O., Dubinsky, Y., Katz, S. “The TDD-Guide Training and Guidance Tool for Test-Driven Development.” Lecture Notes in Business Information Processing, Springer Berlin Heidelberg, pp 63-72, 2008.

Scacchi, W. and C. Jensen. “Data mining for software process discovery in open source software development communities.” In Proc. Workshop on Mining Software Repositories, page 96, . 2004.

Schumacher, J., et al. “Building Empirical Support for Automated Code Smell Detection.” International Symposium for Empirical Software Enginieering. Bolzano, 2010.

Schwaber, K. and M. Beedle. Agile Software Development with Scrum. Prentice Hall PTR, 2001.

Schramm, W., et al. “Software Process Enactment Based on an Object Oriented Description.” Proceedings of the 4th International Workshop Software Engineering & its Applications, Toulouse. 1991.

Silva, L. F. and G. H. Travassos. "Tool-Supported Unobtrusive Evaluation of Software Engineering Process Conformance." In Proceedings of the 2004 international Symposium on Empirical Software Engineering (August 19 - 20, 2004). International Symposium. 2004.

Shull, F., J. Carver and G. H. Travassos. "An empirical methodology for introducing software processes." SIGSOFT Softw. Eng. Notes 26, 5 (Sep. 2001), 288-296. 2001.

Sorumgard, S. "Verification of Process Conformance in Empirical Studies of Software Development." (1997).

Standardization, International Organization for. Quality systems - Model for quality assurance in design, development, production, installation and servicing. 1993.

Stapel, K., D. Lübke and E. Knauss. "Best Pratices in eXtreme Programming Course Design." 30th International Conference on Software Engineering. Leipzig, Germany: IEEE, 2008. 769-776.

Rubin, V., et al. "Process mining framework for software processes." In: Wang, Q., Pfahl, D., Raffo, D.M. (Eds.), Software Process Dynamics and Agility, ICSP 2007, Lecture Notes i 2007.

Rubinstein, D. "sdtimes.com." 2007 йил 01-March. Standish Group Report: There's Less Development Chaos Today. 2009 йил 10-November <http://www.sdtimes.com/article/story-20070301-01.html>.

Reis, C. A., et al. "Flexible Software Process Enactment Support in the APSEE Model." Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (Hcc'02). IEEE Computer Society, Washington, DC, 112, 2002.

Roethlisberger, F.J. and W.J. Dickson. Management and the Worker. Cambridge, Mass.: Harvard University Press, 1939.

Thompson, S., T. Torabi and P. Joshi. “A Framework to Detect Deviations During Process Enactment.” Computer and Information Science, 2007. ICIS 2007. 6th IEEE/ACIS International Conference on , vol., no., pp.1066-1073, 11-13 July 2007. 2007.