Are Developers Complying with the Process: An XP Study

Nico Zazworka[1,3], Kai Stapel[2], Eric Knauss[2], Forrest Shull[3,1], Victor R. Basili[1,3], Kurt Schneider[2]

[1]University of Maryland
College Park
Maryland, USA
+1 301 405 2668
{nico,basili}@cs.umd.edu

[2]Leibniz Universität
Hannover
Germany
+49 511 762 19 666
{kai.stapel,eric.knauss,
kurt.schneider}
@inf.uni-hannover.de

[3]Fraunhofer Center
College Park
Maryland, USA
+1 240 487 2904
fshull@fc-md.umd.edu

ABSTRACT

Adapting new software processes and practices in organizational and academic environments requires training the developers and validating the applicability of the newly introduced activities. Investigating process conformance during training and understanding if programmers are able and willing to follow the specific steps are crucial to evaluating whether the process improves various software product quality factors. In this paper we present a process model independent approach to detect process nonconformance. Our approach is based on non-intrusively collected data captured by a version control system and provides the project manager with timely updates. Further, we provide evidence of the applicability of our approach by investigating process conformance in a five day training class on eXtreme Programming (XP) practices at the Leibniz Universität Hannover. Our results show that the approach enabled researchers to formulate minimal intrusive methods to check for conformance and that for the majority of the investigated XP practices violations could be detected.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – *process metrics.*

General Terms

Management, Measurement, Human Factors, Verification.

Keywords

Process conformance, XP programming, process improvement

1. INTRODUCTION

Introducing new processes and practices in industrial environments is an essential component of improving workflow, reducing the cost of a project and accomplishing quality goals. Whenever a manager decides to introduce a new process, it raises questions of whether the developers can follow the specific steps and activities defined, and whether the process itself is applicable in the target environment. As a first step, the development team should be trained in the new process to guarantee a common understanding of the definition and to equip personnel with the necessary skills. In some cases the training might be delivered by a theoretical lecture with an application of the process to the target project, in others cases the training might include practical exercises to strengthen the understanding. In either case it is worth investigating if developers are complying with the specifics of the process to validate that the process is indeed executed as expected. If a process focuses on improving the quality or cost of the final product then not conforming with its definition will most likely produce a different result.

In order to uncover process conformance violations (i.e. situations not conforming to the process' definition), methods are needed that fulfill a range of requirements.

First, the methods should be general enough to investigate a wide range of processes, practices, methods, and techniques at different stages and levels of the software development lifecycle. A successful approach should be able to handle different definitions of process, ranging from informal ones given as text to stricter and more formal ones expressed through a process modeling language. Further, most processes need to be tailored to the specific environment. Numerous project variables, such as team size, system size and type, implementation language and environment can influence how the steps are carried out and the expected process results of the process. The method for assessing process conformance should also be tailorable as well.

Second, the cost and overhead of the data collection activities to investigate process conformance should be minimal and ideally should make use of data already collected by the development team. Examples for such data bases are version control systems that are primarily used to coordinate development efforts. Further, bug and issue tracking systems can provide insight into process relevant activities. One reason to minimize additional data collection is to avoid any interference with the studied process. For example, if developers need to collect too much data manually, they might be distracted from carrying out the expected steps of the process. Or, if developers are actively watched they might follow the steps more precisely during the time of observation but not afterwards. This effect is known as the Hawthorne effect [1] in experimental studies.

Another requirement is that the method should give insights into the different levels of process execution. We consider two layers as important. A well-defined process describes (a) *what* steps should be followed and in which order (syntactic layer) and (b) *how* these steps should be carried out qualitatively (semantic layer). For example, the first XP practice we investigated in our case study was *Test-Driven-Development.* The process requires developers to (a) implement one or more test cases prior to the associated implementation class and (b) to engineer useful and complete test cases that test the implementation thoroughly. Both the syntactic and the semantic layer can be adhered to independently: developers might follow the steps but still implement poor or even useless test cases, and they might implement good test cases but in the wrong order (i.e. write their test cases after the implementation). A good approach will be able to detect non conformance on both layers.

A last characteristic of a method to assess conformance is how timely the method is in reporting violations to the manager. Immediate feedback enables the recognition of risks to a successful process execution early, allowing the initiation of appropriate counter measures. As an example, developers might need to be reminded from time to time to follow certain steps, or a changing project environment (e.g. a close deadline) might affect the process execution quality.

Our approach was designed with these goals in mind. Related work is described in Section 2 followed by a step by step description of our method in Section 3. A practical example demonstrating the usefulness and limitations of our approach is presented in Section 4. The study was part of a practical XP programming course and investigates the conformance to three popular XP practices. We found significant violations of the practices as taught. Section 5 discusses the results and Section 6 offers concluding remarks.

2. RELATED WORK

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” [2] 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” [3].

There is further evidence that we cannot assume that processes are always executed the way they were intended to be. As one example, in an empirical study investigating reading techniques conducted by Lanubile and Vissagio [4] the researchers found that “ (...) less than one third of Checklist reviewers could be trusted to have used the checklist and one fifth of the PBR 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.”

One approach for assessing conformance was proposed by Cook and Wolf [4,6]. They use an event based framework that expresses the expected process as a finite state machine (FSM). An event stream representing the observed process can then be compared to the FSM and deviations can be found. Several string distance metrics can then be used to express the difference between the observed event sequence and the expected model. The work focuses on the syntactic level of conformance. Building upon this is the work of Huo, Zhang, and Jeffrey [7] that further introduces the idea of an iterative process refinement. Other approaches [20] use temporal logic to detect deviations on the syntactic layer.

A model that is able to give live feedback has been proposed by Thomson et al.[21]. Their work also uses conditions checked during process execution to detect violations. However, the approach presented here extends live conformance checking of a method for defining and evolving the process and rules to be checked. Further, this work provides evidence through a case study with real developers.

An approach to detect nonconformance on the semantic layer of process execution has been previously proposed in [8]. The idea presented in this work is to define upfront an expected time measure and an appropriate quality measure for process execution. These measures can then be compared to the actual, observed ones and a deviation vector can express the distance between both of them.

Our own work [9] has focused on applying our model to detect nonconformance in process execution in a large scale industrial project. Our approach places fewer restrictions on the formalism to define the expected process and, as we will show in this work, captures syntactic and semantic aspects of process execution.

2.1 Agile Conformance

Agile methods claim to have some distinguishing advantages over conventional software development methods [10,11], but little work exists about measuring conformance to agile practices. With risk management being the key motivation for using agile practices, measuring conformance is important for evaluating, whether these risks are really tackled.

Schwaber and Beedle give process related metrics that can be used for measuring conformance to Scrum [11]. They focus on “sprint signatures” in the shape of burndown charts. A burndown chart shows the remaining estimated workload. Nonconformance to agile practices can be deduced from these charts but it is not the primary focus and there is no support measurement. Cohn discusses measuring the performance in agile environments [12]. The focus is on estimating, planning, and tracking, rather than process conformance. In [10] Kent Beck discusses collective code ownership in detail. He claims the following effects of collective ownership:

- Complex code does not live very long, because people refactor code they cannot easily understand. Often, it does not even enter the system, because nobody would write complex code that could not be justified: Developers know that other team members will be reviewing their code in a very short amount of time.
- Knowledge is spread around the team, because it is unlikely that there is any part of the system only two people know about. This reduces project risk (which is the primary motivation for using XP in [10]).

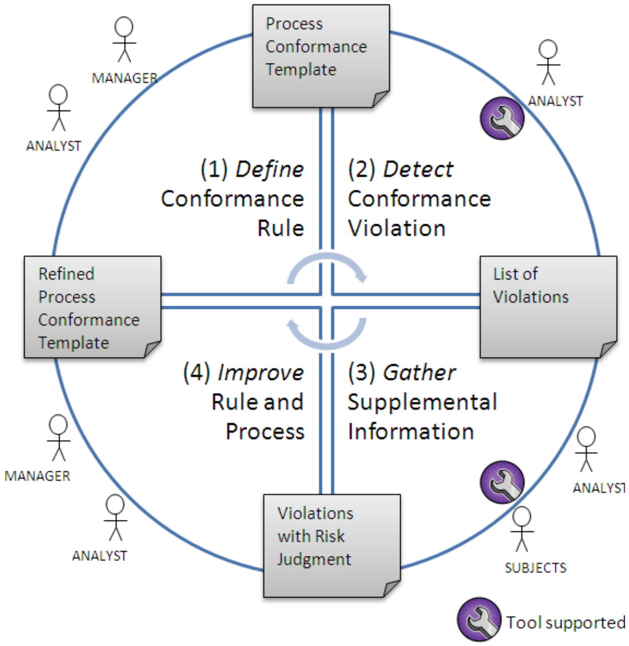
Beck does not give any support for these claims, apart from common sense arguments, and claims that all practices in XP should be “turned to ten,” i.e. that conformance to the textbook practices should be achieved [10]. But he does not provide objective measures on how to find out whether a practice actually is at level ten or not. Krebs and Williams define the conformance to agile practices through a maturity measure [13]. In their work, the maturity levels are defined relatively (i.e. 10 is more agile than 8). Resulting assessments of practice conformance will always be very subjective, because the questionnaire is based on subjective questions.

3. APPROACH

The approach presented in this paper follows the four step iterative model presented in Figure 1. In the following subsections we illuminate each step. We will explain the inputs and outputs of each step, and the roles that are involved. Primarily three different roles are important in our model:

1. Process manager: the person(s) interested in studying the practice/process.
2. Process enactors (developers): the person(s) performing the practice/process.
3. Conformance analyst: the person(s) investigating process conformance.

In small scale efforts the first and third role might be assigned to the same person. Ideally, to limit bias issues, the conformance analysis should be performed by a different person than the manager, or - even better - externally by a third party group.



Process Conformance Template
MANAGER
ANALYST
ANALYST
(1) *Define* Conformance Rule
(2) *Detect* Conformance Violation
Refined Process Conformance Template
List of Violations
(4) *Improve* Rule and Process
(3) *Gather* Supplemental Information
MANAGER
ANALYST
ANALYST
SUBJECTS
Violations with Risk Judgment
Tool supported

Figure 1: The nonconformance process at top level

3.1 Define Conformance Rule

The initial step to be performed aims at creating a common understanding of the details of the expected, planned process. The process manager makes this knowledge explicit by filling in the appropriate fields in the process conformance template (Table 1). How the process is represented (e.g. through a formal or verbal representation) is up to the manager. Therefore various formal process modeling languages, such as finite state machines or Petri Nets [18] are compatible with the template. Also, rather weakly formulated processes that include no specific steps can be expressed. As example, in a current study with an industrial partner the manager formulated that "*All developers should write test cases!*". The requirement to the definition is solely that it allows the conformance analyst to decide if certain patterns in the collected data represent a violation of the process definition or not. Secondly, the conformance analyst and the manager will list all data and measures that are already implicitly collected in the project. Typical examples are code management systems, issue-, bug-, and effort tracking systems. The third step is to *connect* the data and the definition in order to formulate process violations. Both roles have to think and decide about "*Which (temporal) patterns in the collected data will violate the process definition?*" and "*Which metric values are indicators of process violations?*" The first question aims at the syntactical aspects of the process (e.g. the process might be described as a partially ordered set of activities to be performed). The second question focuses on violations on the semantic level (i.e. how/with what quality the process is performed).

Table 1: Process Conformance Template

Process Name	A unique identifier.
Process Focus	Quality attributes that the process should improve: e.g. maintainability, correctness
Process Description	Formal or verbal description of the process.
Collected Data	List of implicitly, manually, and automatically collected data.
Process Violations	**Syntactic**: Which temporal patterns in the data violate the steps of the process? **Semantic:** Which measures and thresholds derived from the collected data indicating low quality of process execution?

The semantic definition of the process might not be always clear from the beginning on. For example, the investigated XP practice *Collective Code Ownership* claims that knowledge about the code should be collectively owned and the loss of a few programmers should not lead to project failure. However, the practice does not define in detail how resistant the project should be to loss of personnel, i.e. how many programmers are expendable exactly, and how much code should be covered by the remaining programmers. In such cases, the manager and conformance analyst will have to choose a first guess for those parameters and iteratively refine them if necessary. A second strategy is to let the practice run for a short time and derive the parameters from the observed data. This can be understood as the derivation of the process from its execution. Of course, this only works under the assumption that the process is performed appropriately.

If not enough process violations can be defined using the implicitly collected data then it is up to the manager to decide on additional data and metric collection activities, such as self-reported data. These can be goal and priority driven (e.g. by using a GQM approach [14]) by first listing the violations that one would be most interested in, and secondly defining the data and measures that have to be collected to detect these violations. However, the collection activities should have the goal of minimizing (or better, avoiding) any kind of interference with the studied process.

3.2 Detect Conformance Violation

After the violations have been defined automated tools and algorithms can be built to detect and extract them from the collected data. These tools might, for example, mine code repository data or self reported data (assuming it is available in electronic form). The algorithms can then be executed on a recurring basis. The frequency of execution might vary from one process to another; in our case study we used daily intervals to generate a list of violations, since one day matched one XP development iteration.

3.3 Gathering Supplemental Information

Once the list of violations has been made available, the process conformance analyst should augment the violations with more detail. This supplemental information can include related quantitative measures as well as data of a qualitative nature (e.g. through interviews with developers). The goal of the augmentation activity is to support the following rule and process improvement step. It is necessary for multiple reasons and purposes:

(1) The violations identified can be false positives, e.g. if the conformance rules are not yet fully tailored to the environment, or if their definition is incomplete. For example, our first version of the Test-Driven Development detection assumed that all Java compilation units should be accompanied by a unit test class. Investigating the violations more closely, we realized that specific types of units, such as Interface classes, do not require test cases[1]. These interfaces were therefore falsely identified as violations. This fact was included in the second version of the conformance rule.

(2) Secondly, it is important to get insight into the cause-effect relationships among violations. Only finding root causes can help us understand why a process is not being followed and what actions have to be taken to improve the situation. Some of these causes might be found through further inspection of the collected data. Others might require interviewing the developers to reveal the reasons for not following the process under investigation.

3.4 Improve Rule and Process

The fourth and final step in the scheme is to make decisions about how to *improve the agreement* between the executed and expected practice. There are essentially three ways to reduce the number of violations for the next iteration:

(1) In the case of false positives, the conformance analyst will have to adjust the rules for detection in the process template (e.g. through modification of the algorithms in step 2 or changing the thresholds for metrics).

(2) In settings that allow modification of the process itself the manager might decide to change or tailor the expected process so it better fits the executed one.

(3) In settings that do not allow process modification the manager might put additional effort into the enforcement of the process. Process enforcement can either be done by reminding the subjects to execute the necessary activities or by the use of mechanisms that enforce the process (e.g. through tools).

3.5 Knowledge Packaging and Transfer

Once the conformance process has gone through multiple iterations, the rules should become more and more stable. These rules, in form of the template in Table 1, now represent *transferable knowledge*. A new project inside the organization can benefit from using optimized rules from prior projects. In experimentation a study replication can make use of the rules formulated in a prior experiment. Therefore, it is necessary to store and package the final rules and, even better, all versions that led to the final rule, in an *experience base* [15]. The implementation of the experience base can vary from simply adding it to the appendix of an experimental paper, to using more sophisticated electronic systems such as reports, data bases or web wikis.

Table 2: Process Conformance Template for Test-Driven-Development (green/light grey font text are additions and modifications made to tailor the template)

Pr. Name	Test-Driven Development
Pr. Focus	Improved correctness.
Process Description	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 scheme for implementation and test classes: Implementation class: `SomeName.java` Test class: `SomeNameTest.java`
Process Violations	**Syntactic**: (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. **Semantic:** (1) The line coverage of the test cases is below 70% (2) The branch coverage of the test cases is below 70%

3.6 Research Questions

To test the feasibility and applicability of our method, we set up the following research questions:

R1- Feasibility: Is the approach able to find nonconformance issues in process execution using minimally intrusive methods?

R2 - Correctness: Do these violations give useful insights and do they match the perceived conformance of the developers?

R3 - Steering: Can these violations be used to actively steer and improve process conformance?

R4: Rule improvement: Can our initial rule set be iteratively refined and improved?

R5: Quality Mapping: Does a lack of process conformance correlate with product quality?

We conducted a case study to test our approach. The following section will give an overview of the study design, the collected data, and the results in relation to our research question.

4. XP STUDY

In order to show that the suggested approach is feasible we conducted a case study. The study should help to answer the research questions given in Section 3, help in understanding where our process needs improvement, and where its limits are. To address R1 (Feasibility) we chose to investigate three popular XP practices:

1. Test-Driven Development
2. Continuous Refactoring
3. Collective Code Ownership

[1] Java interface classes solely define function names, parameters, and return type, but implement no testable functionality.

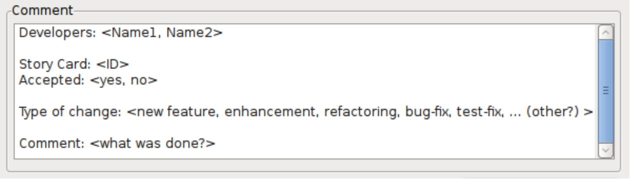
Table 3: Process Conformance Template for Continuous Refactoring

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	**Syntactic:** (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) **Semantic:** (3) Increasing amount of God Class code smells

R2 (Correctness) was investigated by a comparison of the perceived conformance versus the measured one and R3 (Steering) was evaluated using one instance where developers were actively advised to improve conformance to a practice during project runtime. R4 and R5 were investigated internally by the team of conformance analysts during execution. Note that for none of the questions we are able to produce statistically significant results due to the nature of the study.

4.1 Study Design

The case 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 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 without XP experience prior to the class, were split into two groups with seven developers each. Both groups developed a different product; in the following we will refer to them as team *Zeit* and team *KlaRa* in accordance with the names of the two products. The goal of *Zeit* was to build a time logging system. The system is currently in use in several software and research projects. *KlaRa* is a tool that helps coordinate room assignments during exam time at a university. The implementation language was Java in both cases. 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 [16].



```
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 2: SVN commit template for additional data collection

Table 4: Process Conformance Template for Collective Code Ownership

Process Name	Collective Code Ownership (Pair Programming + 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	**Syntactic**: (1) The same developer pair working together on two consecutive *story cards* (2) The same developer pair working together on two consecutive *iterations* **Semantic:** (1) The project's Truck Factor (explained later is low

Before the start of the programming project the researcher teams from LUH and UMD agreed to investigate the conformance of the three XP practices: Test-Driven Development, Collective Code Ownership, and Continuous Refactoring. Each of the three practices was translated into a process conformance template using the criteria specified in Table 1 (see Tables 2 to 4). Further, they agreed on the type of data to collect. Automatically and implicitly collected data was derived from the Subversion code repository that the subjects used to coordinate their work.

Additionally, a small amount of manually collected data was captured. The researchers provided the developers with a special Subversion commit template that had to be filled in every time they committed new code to the repository. As shown in Figure 2 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, who took the role of the *conformance analysts* in Figure 1, created a report with the results of steps 2 and 3 of the method presented in Section 3 (Figure 1).The report was sent to the researchers on site (*process managers*) before the start of the next iteration. There is a time difference 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 1: 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

Table 5: Test-Driven Development Results.

	Zeit			**KlaRa**		
Iteration	New Classes	Test First	**Conf. Level (%)**	New Classes	Test First	**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**

descriptions of how the violation detection rules were tailored over time (Figure 1: step 4). Optimizing the rules of the templates was done by a manual in depth analysis of false negatives and false positives (Figure 1: step 3). 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 process managers 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 had to provide neither their name nor the project they were working on.

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.

4.2.1 Test-Driven Development

Table 5 shows the results for the two groups (*Zeit* and *KlaRa*). The conformance level (in the Table abbreviated with "Conf. Level") for Test-Driven-Development was calculated as follows: for each of the four iterations the newly developed Java classes (in Table "New Classes") were considered and the analysts 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 the practice is followed all times the conformance level would be 100%, if the practice is followed half of the time the level would be 50%, and so on.

Table 6: End of study questionnaire answers for Test-Driven Development

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 7: Continuous Refactoring Results.

	Zeit			**KlaRa**		
Iterat.	**Commits**	**Refac.**	**Ratio**	**Commits**	**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%**

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 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. Overall, both groups adhered in about 50% of the cases to the practice.

The end of study questionnaire data show 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 6 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%).

4.2.2 Continuous Refactoring

The second practice under investigation was continuous refactoring. In comparison to the other investigated practices the process violations were rather weakly formulated (see Table 3). 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, we 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. In addition to the self reported data the number of code smells was measured, in particular the God Class code smell. Inspection of God Classes can give insight if classes implement multiple responsibilities and grow too complex. We used the God Class identification strategies as defined by Marinescu and Lanza[22].

The data in Table 7 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*). In both projects the code smell analysis did not detect any God Classes during development. Therefore, violations of the practice as defined in Table 3 could not be detected. Even though 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 non conformance when used as thresholds in a future study. Further, the self-reported data can help give the numbers more meaning. From the post-study questionnaire (Table 8) 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 be still below an optimal, desired ratio.

4.2.3 *Collective Code Ownership*

The third XP practice under investigation was Collective Code Ownership. The goal of the practice 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 nonconformance in Collective Code Ownership two measures were investigated:

1. Syntactic: Adherence to the activities defined by Pair Switching.
2. Semantic: Assessment of the project's truck factor

As for pair switching, we note that the process managers required that programming pairs were reshuffled at the beginning of each development day (i.e. each iteration). That means that they partly enforced the pair switching practice.

1. Pair switching showed a significant amount of violations. Figure 4 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 process managers).

Table 8: Questionnaire answers for Continuous Refactoring

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%

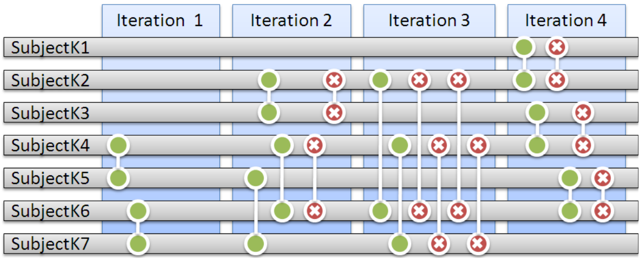


Figure 4: Pair-Switching for team KlaRa.

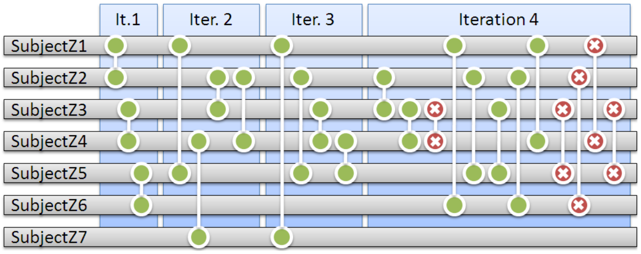


Figure 4: Pair-Switching for team Zeit

For project *Zeit* (Figure 4) the pair switching was followed the first three iterations without violations. Developers did as instructed and 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.

Again, the reported conformance from the questionnaire shows a similar result (Table 9). Only one developer agreed to have followed the practice all the time.

2. The **Truck Factor Analysis** gives insight into how well the code is collectively owned at the end of the projects. For this we define (to our knowledge for the first time) an analysis technique that builds upon the data collected through the code repository to assess the *Truck Factor*. In the box *Truck Factor Metric* we give details on how this analysis can be performed. As pointed out in Section 3.1 one might not have always a clear understanding what the expected measures should look like in such cases (i.e. which truck factor 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 5 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

Table 9: Questionnaire results for Pair Switching

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%

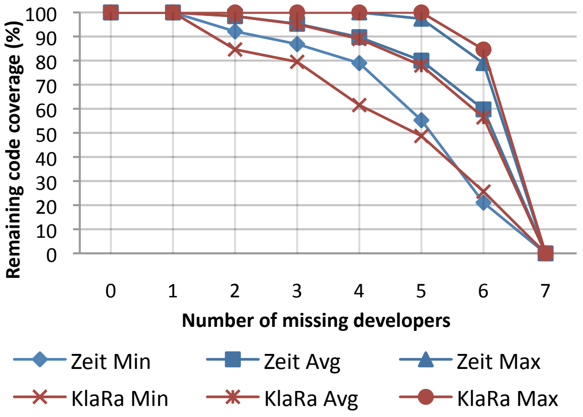


Figure 5: Truck Factor Chart for Zeit and KlaRa

developers. The average case performance is almost equal with a slight advantage for Zeit. Figure 5 also shows the impact of pair programming: the loss of one programmer can always be covered by the programmers she/he worked in a pair with. 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 that we are describing in more detail in [9] and the development of two research tools developed at the two participating universities: CodeVizard and HeRa (a requirements editor mostly developed by one programmer [19]). HeRa was included to demonstrate what a lower bound Truck Factor could look like.

Figure 6 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.

In the end of study questionnaire subjects were asked what percentage of the final system they feel to 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 10 and Table 11.

4.3 Discussion of XP Conformance 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 did perform poorly in following Pair Switching.

Table 10: End of study questionnaire answers for Collective Code Ownership: question 2

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

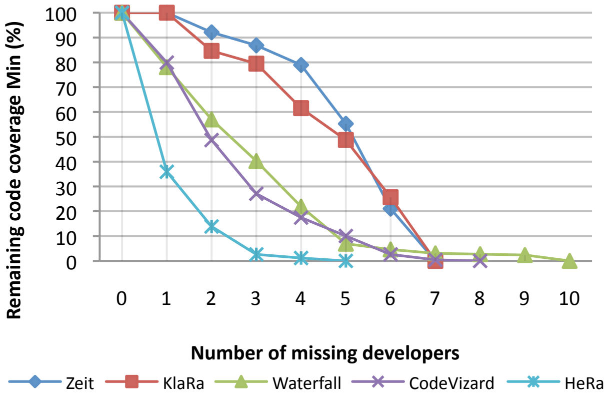


Figure 6: Worst case scenarios for 5 different projects including the two XP projects (Zeit, KlaRa).

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*".

5. Addressing the Research Questions

For the stated research questions of our approach we make following conclusions:

R1 Feasibility: We were able to translate three XP practices into our scheme, to collect data non-intrusively with minimal manual effort, and formulate and detect violations against these. For most of the semantic violations thresholds and measures were found and tailored during the execution of the processes. Semantic violations seem to be harder to define upfront. The study produced a set of conformance rules (Tables 2-4) that might be adapted for other studies.

R2 Correctness: The perceived conformance of the subjects fits the measured one to some extent. For two practices we could find a significant amount of violations and subjects admitted to have not followed the practice all times.

R3 Steering: Subjects of team Zeit were advised to improve their conformance to Test-Driven Development one time before iteration 3. The impact is visible in the conformance level (Table 5). The number of violations was lowered, but they still occurred after that.

Table 11: Questionnaire answers for Collective Code Ownership: question 1

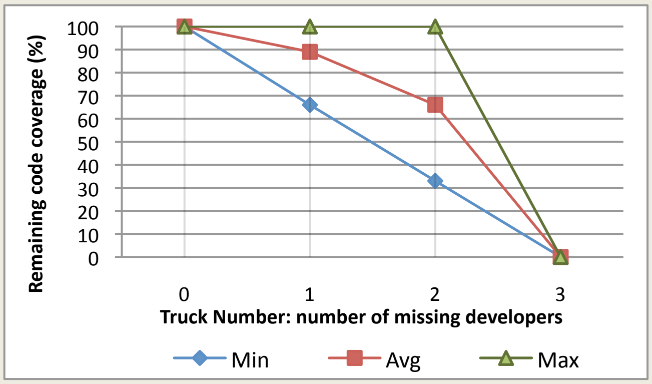
How much percent of the system have you been working on?	*Instances*	*Percentage*
<25 %	1	7%
25-50%	5	36%
>50-75%	5	36%
>75%, <100%	2	14%
100%	1	7%

TRUCK FACTOR METRIC: 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*" [17]. 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 [10], situations where a small set of programmers owns a large part of the code base *exclusively*. To our knowledge, this measure has been proposed informally only so far and we are the first to derive this number by using information about code ownership from a code repository. The key idea of our analysis is that a source component (e.g. a Java file) in the repository is collectively owned by the developers who worked on that component. For the purpose of simplicity one can assume that all developers who edited the file have knowledge about it. More sophisticated methods to assign ownership have been proposed [23][24].

The table on the right side 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 components from the code repository data we can generate different scenarios where we assume that a certain subset of developers has been "hit by a truck". For each component we can decide if the remaining developers have knowledge about it (light cells with "+" sign) or not (dark cells with "-" sign). A coverage number $\text{cov}_x(n)$ 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 on the right 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 target 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.

		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

Example analysis with 3 files and 3 developers



Truck factor characteristics: x-axis shows the number of missing developers

Therefore, we define the truck factor as: $\text{tf}_{x,\,c} = \max\ \{n \mid \text{cov}_x(n) \geq c\}$

For example, the worst case 60% coverage truck factor of our example would be: $\text{tf}_{\min,\,60\%} = \max\ \{n \mid \text{cov}_{\min}(n) \geq 60\%\} = 1$

R4 Rule Improvement: Our study shows that we could improve and adjust the rules to the environment and practices. For all the rules we did not have a good understanding of the semantic levels before the study but were able to derive measures and thresholds during the execution. Further we were able to catch some special cases (i.e. Java interface classes) to improve the automated detection of violations.

R5 Quality Mapping: So far we were 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 seem indeed to be linked to a better truck factor when compared to conventional non-XP projects. Future investigation is needed to understand if process nonconformance results in lower quality products.

6. DISCUSSION OF VALIDITY

The study presented has internal and external threats to validity. Since the work's main contribution is considered the generic approach to detect conformance violations we will focus on discussing threats that are introduced by the approach itself (and not the threats related to drawing conclusions about the applicability of the inspected XP practices in other environments). The major internal threat is the correctness of the measured data, especially the manually collected data. Subjects might have not accurately reported having performed refactoring activities, or about having switched their pair programming partners. Evidence that this was not the case can be found in the post study questionnaire data: if developers would have intentionally lied in order to improve conformance then it is likely that they would have done the same when filling in the questionnaire. Further the data extracted for Test-Driven Development from the repository has some limitation in time resolution. Since the detection can only check the test-first order at check in time, developers might have still developed test cases after the implementation but checked in both files together at the same time into the Subversion repository. Therefore, we might miss violations in those cases (in other words: the conformance to the practice could be even lower).

Considering external validity we used methods of automatic and manual data collection that are applicable as well in industrial environments that use version control systems. Further the presented approach was shown to be capable of adapting three different XP practices (and with [9] two different non XP processes). Still the question remains unanswered to how many it can be applied of the wide range of processes, practices, techniques, and methods already proposed in the software engineering literature.

7. CONCLUSION

In this paper we presented a step by step approach to investigate conformance issues in process execution. We tested the ideas in an XP classroom study and illustrated that it is possible to find conformance violations using minimally intrusive methods. Further, we provide evidence that this can be done remotely by an independent research group. Our findings suggest that nonconformance is indeed an issue present when teaching new processes to developers, and that it should be assessed to better understand applicability and effectiveness of such processes. As part of the presented approach, we provide a common template to facilitate knowledge transfer across software projects and studies.

Last, we have learned useful lessons during the execution of our model. The biggest challenge was to find definitions for the XP practices that contained enough detail. In all cases, we had to define and iterate the semantic properties ourselves. Further, we had to define the truck factor as a measurable metric.

We have started to work with an industrial partner to investigate nonconformance in a long term study. Future work will focus on actively tailoring our templates and the partner's processes to improve conformance and investigate the relationships between conformance and product quality (R5).

8. ACKNOWLEDGEMENTS

This research was supported by NSF grant CCF 0916699, "Measuring and Monitoring Technical Debt ".

9. REFERENCES

[1] Roethlisberger, Fritz J., and W. J. Dickson. Management and the Worker. Harvard University Press, 1939.

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

[3] Information Technology - Software life cycle processes. International Organization for Standardization, 1995.

[4] Lanubile, F. and Visaggio, G., "Evaluating Defect Detection Techniques for Software Requirements Inspections", ISERN Report no. 00-08, 2000.

[5] Cook, J. E. and Wolf, A. L. 1998. Discovering models of software processes from event-based data. ACM Trans. Softw. Eng. Methodol. 7, 3 (Jul. 1998), 215-249.

[6] Cook, J. E. and Wolf, A. L. Software process validation: quantitatively measuring the correspondence of a process to a model. *ACM Trans. S. E. Meth.* 8, 2 (Apr. 1999), 147-176.

[7] Huo, M., Zhang, H., and Jeffery, R. 2006. An exploratory study of process enactment as input to software process improvement. In *Proceedings of the 2006 international Workshop on Software Quality* .WoSQ '06. ACM, New York, NY, 39-44.

[8] S. Sørumgård. "Verification of Process Conformance in Empirical Studies of Software Development". Ph.D. thesis, Norwegian University of Science and Technology, 1997.

[9] Zazworka, N. Basili,V.R., and Shull. F. "Tool Supported Detection and Judgment of Nonconformance in Process Execution" in Proceedings of ESEM 2009, 312-223, October 16-15, 2009

[10] Kent Beck, Extreme Programming Explained: Embrace Change, Addison Wesley 1999

[11] Ken Schwaber and Mike Beedle, Agile Software Development with Scrum, Pearson Education 2001

[12] Mike Cohn, Agile Estimation and Planning, Prentice Hall 2007

[13] Krebs, William (2002): Turning the Knobs: A Coaching Pattern for XP through Agile Metrics.Springer, Lecture Notes on Computer Science 2418, 60-69

[14] Basili, V. R. 1992 Software Modeling and Measurement: the Goal/Question/Metric Paradigm. Technical Report. University of Maryland at College Park.

[15] Basili, V., Caldiera, G., and Rombach, D. Experience Factory. Encyclopedia of Software Engineering Volume 1:469-476, Marciniak, J. ed. John Wiley & Sons, 1994

[16] Stapel, K., D. Lübke, and E. Knauss: Best Practices in eXtreme Programming Course Design. in Proceedings of 30th International Conference on Software Engineering. 2008. Leipzig, Germany,769-776.

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

[18] Tadao Murata, "Petri Nets: Properties, Analysis and Applications", in: Proceedings of the IEEE, vol. 77, no. 4, April 1989

[19] Knauss, Eric.; Lübke, Daniel; Meyer, Sebastian, "Feedback-Driven Requirements Engineering: The Heuristic Requirements Assistant", ICSE'09, Formal Research Demonstrations Track, 2009, 587 - 590

[20] Cugola, G., Di Nitto, E., Ghezzi, C., and Mantione, M. 1995. How to deal with deviations during process model enactment. In *Proceedings of the 17th international Conference on Software Engineering*. ICSE '95. ACM, New York, NY, 265-273.

[21] Sean Thompson, Torab Torabi, Purva Joshi, "A Framework to Detect Deviations During Process Enactment," Computer and Information Science, ACIS International Conference on, pp. 1066-1073, 6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007), 2007.

[22] Marinescu, Radu; Lanza, Michelle (2006). Object-Oriented Metrics in Practice. Springer.

[23] Anvik, J., Hiew, L., and Murphy, G. C. 2006. Who should fix this bug?. In *Proceedings of the 28th international Conference on Software Engineering* (Shanghai, China, May 20 - 28, 2006). ICSE '06. ACM, New York, NY, 361-370.

[24] Hattori, L. and Lanza, M. 2009. Mining the history of synchronous changes to refine code ownership. In *Proceedings of the 2009 6th IEEE international Working Conference on Mining Software Repositories* (May 16 - 17, 2009). MSR. IEEE Computer Society, Washington, DC, 141-150