

On the Benefit of Automated Static Analysis for Small and Medium-Sized Software Enterprises

Mario Gleirscher¹, Dmitriy Golubitskiy¹, Maximilian Irlbeck¹, and
Stefan Wagner²

¹ Institut für Informatik, Technische Universität München, Germany
`{gleirsch,golubits,irlbeck}@in.tum.de`

² Software Engineering Group, Institute of Software Technology,
University of Stuttgart, Germany
`stefan.wagner@informatik.uni-stuttgart.de`

Abstract. Today’s small and medium-sized enterprises (SMEs) in the software industry are faced with major challenges. While having to work efficiently using limited resources they have to perform quality assurance on their code to avoid the risk of further effort for bug fixes or compensations. Automated static analysis can reduce this risk because it promises little effort for running an analysis. We report on our experience in analysing five projects from and with SMEs by three different static analysis techniques: code clone detection, bug pattern detection and architecture conformance analysis. We found that the effort that was needed to introduce those techniques was small (mostly below one person-hour), that we can detect diverse defects in production code and that the participating companies perceived the usefulness of the presented techniques as well as our analysis results high enough to include the techniques in their quality assurance.

Key words: software quality, small and medium-sized software enterprises, static analysis, code clone detection, bug pattern detection, architecture conformance analysis.

1 Introduction

Small and medium-sized enterprises (SMEs) play a decisive role in global software industry. In many countries, like the US, Brazil or China, these companies represent up to 85% of all software organisations [24] and carry out the majority of software development [22]. Nevertheless, SMEs are confronted with special circumstances like limited resources, lack of expertise or financial insecurity.

Problem While there are many articles focusing on process improvement in SMEs [14, 22, 28], we found no study that looks at specific quality assurance (QA) techniques and their application in this context. Contrary to this observation, the properties of automated static analysis techniques seem to be suitable for SMEs. The benefits of such techniques lie in their low-cost application and

their potential to detect critical quality defects [33, 2]. Such defects are risky for the further development and increase costs. These arguments are promising for small software enterprises and their need for efficient quality assurance.

Research Objective Our goal is to answer the question whether SMEs can benefit from automated static analysis techniques. Is it possible to introduce a set of such techniques in their existing projects with low effort? What kind of defects can be found using these techniques? Finally, is the perceived usefulness for the enterprises high enough to justify the needed effort? We think that these aspects are useful for future decisions in SMEs on using static analysis techniques in their projects.

Contribution In this article, we describe our experience in analysing five projects of five SMEs using three different static analysis techniques: code clone detection, bug pattern detection and architecture conformance analysis. We evaluate the effort that is needed to introduce these techniques, the pitfalls we came across and how the participating enterprises evaluated the presented techniques as well as the defects we discovered in their projects.

2 Approach

We describe our experiences with transferring static analysis technology to small and medium-sized enterprises. This section illustrates the research context, i.e., the participating enterprises, our guiding research questions, the regarded static analysis techniques, the procedure we used to get answers to the research questions and finally the study objects we employed to gather the experiences.

2.1 Research Context

Fundamental for our research was the collaboration with five SMEs, all resident in the Munich area and selected through personal contacts and a series of information events and workshops. Details regarding the selection process can be found in Sec. 2.5. Following the definition of the European Commission [5], one of the participating enterprises is micro-, two are small and two are medium-sized considering their staff head count and annual turnover. The presented research is based on the experience with these enterprises gathered in a project from March 2010 to April 2011.

2.2 Research Questions

Our overall research objective is to analyse the transfer of new and innovative quality assurance techniques to small enterprises. We structure this objective into two major research questions.

RQ 1 *What problems occur while introducing and applying static analysis techniques at SMEs?*

SMEs exhibit special characteristics, such as generalist employees instead of specialists for quality assurance. Hence, smooth introduction and application are necessary so that the enterprises can adopt and make use of static analysis. We further break this down into two sub-questions:

RQ 1.1 *What technical problems occur?*

Static analysis is tightly coupled to tools that perform and report the analysis. Hence, the ease to introduce and apply static analysis also depends on how many and which technical problems the software engineers need to solve.

RQ 1.2 *How much effort is necessary?*

If the effort necessary to bring the analyses up and running is too large, it can be a killer criterion for an SME, which cannot afford to reserve extra capacities for that. Therefore, we analyse the effort spent in the introduction and application.

RQ 2 *How useful are static analysis techniques for SMEs?*

Beyond how easy or problematic it is to introduce and apply static analysis in SMEs, we are interested in whether we can produce useful results for them. Even a small effort should not be spent if there is no return on investment. We again break this question down into two sub-questions:

RQ 2.1 *Which defects can be found?*

We establish a measure of usefulness by analysing the types and numbers of defects found by using the static analysis tools at the SME. If critical defects can be found, the application of the techniques is considered useful. We neither focus on specification defects and whether they can be found at all, nor do we perform cause and effects analyses for defects except for some criticality assessments.

RQ 2.2 *How do the companies perceive the usefulness?*

We add the subjective perception of our project partners. How do they interpret the results of the static analysis tools? Do they believe they can work with those tools and are they going to apply them continuously in their future projects? This way, we augment the information we gained from defect analysis.

2.3 Static Analysis Techniques

Static analysis is known as the checking of software against certain properties without executing it. It includes manual techniques, such as reviews and inspections, as well as automated techniques. As manual analyses are time-consuming and prone to missing problems in the huge amount of code to analyse, automation has high potential. For example, to detect simple and reoccurring problems in source code, such as using “==” instead of “equals” to compare strings in Java, should not be the task of human reviewers. They should concentrate on the more subtle and domain-related problems. From the interviews with our partners and the experiences at our research groups, we chose three important techniques, which we introduce in detail in the following. Technically, we employ the open-source tool ConQAT¹ for code clone detection and architecture conformance analysis as well as for results processing of bug pattern detection.

¹ <http://www.conqat.org>

Code Clone Detection Modern programming languages, particularly object-oriented ones, offer various abstraction mechanisms to facilitate reuse of code fragments, but copy-paste is still a widely employed reuse strategy. This often leads to numerous duplicated code fragments—so called clones—in software systems. As stated in the surveys of Koschke [16] and Roy and Cordy [26], cloning is problematic for software quality for several reasons:

- Cloning unnecessarily increases program size and thus efforts for size-related activities like inspections and testing.
- Changes, including bug fixes, to one clone instance often need to be made to the other instances as well, again increasing efforts.
- Inconsistently performed changes to duplicated source code fragments can introduce bugs.

Code clone detection is an automated static analysis technique that focuses on finding duplicated code fragments. One of the most important metrics offered by this technique is *unit coverage*, which is the probability that an arbitrarily chosen source statement (i.e. a unit) is part of a clone. Another metric called *blow-up* denotes the ratio of the unit count of the current software w.r.t. the unit count of a hypothetical software without clones [13]. Moreover, two terms are important for clone detection: A *clone class* defines a set of similar code fragments and a *clone instance* is a representative of a clone class [12].

We differentiate between conventional clone detection and gapped clone detection. During conventional clone detection, clones are considered to be syntactically similar copies; only variable, type, or function identifiers could be changed [16]. In contrast, gapped clone detection reveals clones with further modifications; statements could be changed, added, or removed [16]. While clones are an indicator of bad design, the difference between the two approaches is that only the results of gapped clone detection can reveal defects that lead to failures, which arise through unconscious, inconsistent changes in instances of a clone class.

Clone detection is supported by a number of free and commercial tools. The most popular of them are CCFinder², ConQAT, CloneDR³, and Axivion Bauhaus Suite⁴. The former two are free, while the latter two are commercial.

Bug Pattern Detection By this term we refer to a technique for automated detection of a variety of defects. Bug patterns have been thoroughly investigated, e.g. in [33], and compared with other frequently used software quality assurance techniques such as code reviews or testing [31]. Bug patterns represent a scalable approach to efficiently reveal defects or possible causes thereof. Following Wagner et al. [30] they can be cost-efficient after detecting only three field defects. Their detectors, aka *rules*, aim at structural patterns recognisable from source code, executables and meta-data such as source code comments and debug symbols to

² <http://www.ccfinder.net>

³ <http://www.semanticdesigns.com/Products/Clone>

⁴ <http://www.axivion.com>

gain as much knowledge as possible from a static perspective. This knowledge encompasses obvious bugs, rather complex heuristics for latent defects, e.g. code clones (focused in Sec. 2.3), and less critical issues of coding style.

Because of the large bandwidth of defects, bug patterns are categorised along a variety of tool-specific, non-standard criteria. A reason for that is that generally applicable defect classifications are rare, vague or difficult to use in practice [29]. The tools used for this report classify their rules according to the consequences of findings such as security vulnerability, performance loss or functional incorrectness. By the term *finding* we denote that a rule was applied at a specific location. Often, findings are themselves categorised by their severity and their confidence levels.

Many of the rules are realised by means of individual lexers and parsers, by using compiler infrastructures, or by more reusable means such as pattern or rule languages and machine-learning. Rules for latent defects and coding style often stem from abstract source code metrics as, e.g., realised in Ferzund, Ahsan, and Wotawa [7]. Among the wide variety of tools [32] available for bug pattern detection, free and more popular ones are, e.g., splint⁵ for C, cppcheck⁶ for C++, FindBugs⁷ for Java as well as FxCop⁸ for C#.

Architecture Conformance The phenomenon of architectural erosion is a widely documented problem [6, 8, 25]. Architectural knowledge erodes or even gets lost during the lifetime of a system. Accordingly, the documented and implemented architectures are drifting apart from each other. This effect leads to a downward spiralling maintainability of the system. In some cases the effort needed to re-implement the whole system becomes lower than to maintain it. To counteract this situation different approaches are used to compare the system's implementation with its intended architecture.

Passos et al. [23] identify three static concepts existing for architecture conformance analysis: *Reflexion Models (RM)*, *Source Code Query Languages (SCQL)* and *Dependency Structure Matrices (DSM)*.

Reflexion Models Koschke and Simon [17] compare two models of a system to each other and check their conformance. The first model usually represents the intended architecture, the second one the implementation of the system [15]. The intended architecture consists of components and allowed relationships between components, expressed as rules. Each component itself can contain sub-components. The system's code is mapped to these components and then analysed for conformance to the given rules. This technique is used by the commercial tools SonarJ⁹ and Structure101¹⁰ as well as the open-source tools ConQAT and dependometer¹¹.

⁵ <http://splint.org>

⁶ <http://cppcheck.sourceforge.net>

⁷ <http://findbugs.sourceforge.net>

⁸ <http://msdn.microsoft.com/en-us/library/bb429476%28v=vs.80%29.aspx>

⁹ <http://www.hello2morrow.com/products/sonarj>

¹⁰ <http://www.headwaysoftware.com>

¹¹ <http://source.valtech.com/display/dpm/Dependometer>

SO	Platform	Sources	Size [kLoC]	Business Domain
1	C#.NET	closed, commercial	≈ 100	Corporate controlling
2	C#.NET	closed, commercial	≈ 200	Embedded device maintenance
3	Java	open, non-profit	≈ 200	Health information management
4	Java	closed, commercial	≈ 100	Communal controlling
5	Java	closed, commercial	≈ 560	Document processing

Table 1. Study objects

There are tools using *SCQL* like Semmle.QL [3] or *DSM* like Lattix [27], for the sake of brevity not further explained here. Both of these concepts rely strongly on the realisation of the system and cannot provide an architecture specification that is independent of the system’s implementation [4].

2.4 Study Subjects and Objects

Study Subjects For our investigation, we collaborate with five SMEs. These companies cover various business and technology domains, e.g. corporate and communal controlling, form letter processing as well as diagnosis and maintenance of embedded systems. Four of them are involved in commercial software development, one in software quality assurance and consulting. The latter could not provide an own software project.

Study Objects (SO) Following the suggestion of the partner without a software project, we instead chose the humanitarian open-source system OpenMRS¹², a development of the equally named multi-institution, non-profit collaborative. Hence, our study objects are the five software systems briefly described in Tab. 1. These software systems encompass between 100 and 600 kLoC. The developments of SOs 1 to 4 are conducted or audited by the study subjects and started at most seven years ago. The project teams contain less than ten persons. Except for OpenMRS, they are located in the Munich area. The development of SOs 1 and 2 has already been finished before our project started.

2.5 Procedure

This section explains milestones of our investigation (Step 1–4). It explains the starting of our research (Step 1), addresses our research questions, i.e. which data have to be collected and how to achieve that (Step 2) as well as how and under which conditions our analyses have to be carried out (Step 3–4). Steps 2 and 3 take place in terms of a single, collaborative two-week *sprint* per participating enterprise.

Step 1: Workshops and Interviews We conduct a series of workshops and interviews to first convince industrial partners to participate in our project and then to understand their context and their needs. First, in an information event,

¹² <http://www.openmrs.org>

we explain the general theme of transferring QA techniques and propose first directions. With the companies that agreed to join the project, we conduct a kick-off meeting and a workshop to create a common understanding, discuss organisational issues and plan the complete schedule. In addition, the partners present a software system that we can analyse as well as their needs concerning software quality. To intensify our knowledge of these systems and problems, for each partner we perform a semi-structured interview with two interviewers and a varying number of interviewees. Both interviewers take notes and consolidate them. We then compare all interview results to find commonalities and differences. Finally, we have one or two consolidation workshops to discuss our results and plan further steps.

Step 2: Raw Data Collection The *source code* of at least three versions of the study objects, e.g., major releases chosen by the companies, is retrieved for the application of the chosen techniques for RQ 1 to analyse effects over time. For bug pattern detection and architecture conformance analyses, we retrieve or build executables packed with debug symbols for each of these configurations. For architecture conformance we also need an appropriate *architecture documentation*. To accomplish this step, all partners have to provide project data as far as available, i.e. source code, build environment and/or debug builds, as well as documentation of source code, architecture and project management activities.

Step 3: Measurement and Analysis We apply each technique to the gathered raw data via corresponding tool runs and inspect the results, i.e. findings and statistics. To provide answers for RQ 1 we consider problems arising and efforts spent. The tool runs enable us to derive answers for RQ 2.1. To accomplish this step, the partners have to provide support for technical questions by a responsible contact or by personal attendance at the sprint meetings. One person per technique carries out this step for all SOs. The following explains how this is accomplished:

Code Clone Detection We use the clone detection feature [11] of ConQAT 2.7 for all SOs. In case of conventional clone detection the configuration consists of two parameters: the minimal clone length and the source code path. In case of gapped clone detection such gap-specific parameters as maximal allowed number of gaps per clone and maximal relative size of a gap are additionally required. Based on the experience of our group and initial experimentation, we set the minimal clone length to 10 lines of code, the maximal allowed number of gaps per clone to 1 and the maximal relative size of a gap in our analysis to 30%. After providing the needed parameters we run the analysis.

To inspect the analysis metrics and particular clones we use ConQAT. It provides a list of clones, lists of instances of a clone, a view to compare files containing clone instances and a list of discrepancies for gapped clone analysis. This data is used to recommend corrective actions. Also in a series of runs of clone detection over different versions of respective systems we monitor how several parameters (cf. Sec. 2.3) evolve in subsequent versions.

Bug Pattern Detection For Java-based systems we use FindBugs 1.3.9 and PMD¹³ 4.2.5. In C#.NET contexts we use Gendarme¹⁴ 2.6.0 and FxCop 10.0. Aside from applying all rules, we choose two additional tool settings which we consider as being relevant for the SOs to gain two focused quality perspectives:

- 1) *Selected categories* addressing correctness, performance, and security
- 2) *Selected rules* for unused or poorly partitioned code and bad referencing

The tool settings are determined during preliminary analysis test runs. Categories and rules which are considered as not important – based on discussion with the partners as well as requirements non-critical to the SOs’ application domains – are ignored during rule selection.

To simplify the issue of defect classification (cf. Sec. 2.3) for our investigation we only distinguish between rules for *bugs* (obvious defects), *smells* (simple to very complex heuristics for latent defects) and *pedantry* (less critical issues with focal point on coding style).

For additional and language independent metrics (e.g., lines of code without comments; code-comment ratio; number of classes, methods and statements; depth of inheritance and nested blocks; comment quality) as well as for result preparation and visualisation we apply ConQAT.

Next, we analyse the finding reports resulting from the tool runs. This step involves the filtering of findings as well as the inspection of source code to confirm the severity and confidence of the findings and to determine corrective actions. To get feedback and to confirm our conclusions from the findings we discuss them with our partners during a workshop.

Architecture Conformance Analysis We use ConQAT for this technique. The procedure for each system consists of four steps:

- 1) Configuration of the tool with path to source code and corresponding executables of the system
- 2) Creation of the architecture reflexion model (cf. Sec. 2.3) based on the architectural information given by the enterprises
- 3) Run of the architecture conformance analysis
- 4) Defect analysis: Identification, discussion and classification of architectural violations

A detailed description of this ConQAT feature can be found in [4]. In summary, we use a reflexion model where dependency and hierarchy relations between components can be expressed. As a next step, we map modelled components to code parts (e.g. packages, namespaces, classes). We exclude code parts from the analysis that do not belong to the system (e.g. external libraries). Then, ConQAT analyses the conformance of the system to the reflexion model. Every existing dependency that is not allowed by the architectural rules represents a defect. Defects are visualised by the tool on the level of components and on the

¹³ <http://pmd.sourceforge.net>

¹⁴ <http://www.mono-project.com/Gendarme>

level of classes and can therefore be analysed on both high and low level. To eliminate tolerated architecture violations and to validate the created reflexion model, we discuss every found defect with the enterprise. As a last step we classify all defects together with the responsible enterprise. This allows us to group similar defects and to provide a general understanding.

Step 4: Questionnaire First, we evaluate the experience of the participating enterprises regarding software quality as well as static analysis techniques. Second, we want to understand the perceived usefulness of static analysis techniques for SMEs: Do they plan to use the presented techniques in their future projects? Thus, we perform a survey on our study subjects using a questionnaire containing nine questions (Q1-9), which can be found in Appendix A. This way we contribute to RQ 2.2. The executive managers of each enterprise in their role as a representative for their company then fill out this questionnaire and we evaluate the answers. To avoid the risk of biased or too narrowly formulated answers we use both, open and closed questions.

3 Results

We held the information event of Step 1 of our procedure (cf. Sec. 2.5) in July 2009 and invited more than thirty SMEs of which finally 12 participated. From these companies, five committed to take part in the project. The other companies were not able to provide the necessary commitment because of schedule or budget constraints. As the first discussions were generally about improving quality assurance, it was not caused by the choice of techniques. We conducted the kick-off meeting in March 2010, the interviews between March and July and, finally, two consolidation workshops in July 2010. We did not particularly analyse their outcome for this paper. But based on these interviews and the experience of our research group, we selected the three static analysis techniques and interpreted our further results.

In the following we portrait for each technique how we contribute to the posed research questions.

3.1 Code Clone Detection

RQ 1.1 – Technical Problems Code clone detection turned out to be the most straightforward and least complicated of the three techniques. It has, however, some technical limitations that could hinder its application in certain software projects.

A major issue was the analysis of projects containing both, markup and procedural code like JSP or ASP.NET. Since ConQAT supports either a programming language or a markup language during a single analysis, it is required to aggregate the results for both languages. To avoid this complication and to concentrate on the code implementing the application logic we took into consideration only the code written in the programming language and ignored the

Phase	Work step	(C)lone (D)etection	Bug Pattern Detection	Architecture Conformance
Introduction (configuration) and calibration	Analysis tools	$\leq 0.5h$	$\leq 1h$	$\leq 0.5h$
	Aggregation via ConQAT	n/a	$\leq 0.5d$	$\leq 0.5h$
	Recalibration, x -times	n/a	$\leq x * 0.5h$	n/a
Application (analysis)	Run analysis	$\leq 5min$	$1min \leq . \leq 1h$	$\leq 10sec$
	Inspection of results	$\leq 1h$, more for gapped CD	$5min \leq . \leq 0.5h$	$5min \leq . \leq 0.5h$

Table 2. Efforts spent (RQ 1.2) per study object for applying each of the techniques

markup code. Nevertheless, it is still possible to combine the results of clone detection of the code written in both languages to get more precise results.

Another technical obstacle was filtering out generated code from the analysed code basis. In one SO large parts of the code were generated by a parser generator, viz. ANTLR. We excluded this code from our analysis using ConQAT’s feature to ignore code files specified by regular expressions.

RQ 1.2 – Spent Effort The effort required to introduce clone detection is small compared to the other two techniques under study. The ease of introduction of clone detection is achieved due to the minimalist configuration of the analysis which in the simplest case includes the path to the source code and the minimal length of a clone.

For all SOs it took less than an hour to configure clone detection, to get the first results and to investigate the longest and the most frequent clones. Running the analysis process itself took less than five minutes.

In case of gapped clone detection it could take a considerable amount of time to analyse if a discrepancy is intended or if it is a defect. To speed up the process ConQAT supports that the intended discrepancies can be fingerprinted and excluded from further analysis runs. An overview of the efforts can be found in Tab. 2.

RQ 2.1 – Found Defects The results of conventional clone detection can be interpreted as an indicator of bad design or of bad software maintainability, but they do not point at actual defects. Nevertheless, these results give first hints, which code parts must be improved. The following three design flaws were detected in all analysed systems to a certain extent: cloning of exception handling code, cloning of logging code and cloning of interface implementation by different classes.

Tab. 3 shows the clone detection results for three versions of each SO, sorted by time. In the analysed systems unit coverage as defined in Sec. 2.3 varied between 14 and 79%. Koschke [16] reports on several case studies with unit coverage values between 7 and 23% and one case study with a value of 59%, which he defines as extreme. Therefore, the SOs 1, 3 and 5 contain normal clone

SO	Version	Analysed Units [kUnits]	Cloned Units [kUnits]	Blow-up [%]	Unit Coverage [%]	Longest Clone [Units]	Most Clone Instances
1	I	15,9	3,5	119.5	22.2	112	39
	II	25,3	5,8	118.9	23.0	117	39
	III	32,3	7,8	119.2	24.0	117	39
2	I	35,4	14,3	143.1	40.5	63	64
	II	41,6	18,9	150.2	45.4	132	47
	III	39,9	14,6	137.4	36.7	89	44
3	I	51,7	9,4	114.5	18.2	79	21
	II	56,8	8,6	111.2	15.1	52	20
	III	61,6	8,4	110.0	13.7	52	19
4	I	8,9	6,0	238.8	68.0	217	22
	II	22,4	17,3	309.6	77.6	438	61
	III	38,3	30,4	336.0	79.4	957	183
5	I	196,3	48,7	122.3	24.8	141	72
	II	211,3	53,4	122.7	25.3	158	72
	III	208,6	53,2	122.8	25.5	156	72

Table 3. Results of code clone detection

SO	Version	Analysed Units [kUnits]	Cloned Units [kUnits]	Blow-up [%]	Unit Coverage [%]	Longest Clone [Units]	Most Clone Instances
1	I	13,3	3,0	119.9	22.3	34	39
	II	21,0	4,5	117.9	21.5	37	52
	III	27,1	6,0	117.4	22.1	52	52
2	I	24,3	4,6	116.3	19.0	156	37
	II	34,7	8,7	123.2	25.0	156	37
	III	37,1	9,4	123.7	25.3	156	37
3	I	46,7	12,0	124.4	18.2	73	123
	II	46,1	10,0	120.0	15.1	55	67
	III	49,1	10,0	118.6	20.5	55	64
4	I	7,8	4,5	192.1	58.6	42	34
	II	18,8	11,0	206.2	59.8	51	70
	III	32,2	19,2	211.1	59.5	80	183
5	I	142,3	29,4	117.4	20.7	66	68
	II	154,0	32,8	118.0	21.3	85	78
	III	151,9	32,7	118.2	21.5	85	70

Table 4. Results of gapped code clone detection

rates according to Koschke. The clone rate in SO 2 is higher than the rates reported by Koschke and for SO 4 it is extreme. Regarding maintenance the calculated blow-up for each system is an interesting value. For example version III of SO 4 is more than three times bigger as its hypothetically equivalent system containing no clones. SO 4 shows that cloning can be an increasing factor over time, while SO 3 reveals that it is possible to reduce the amount of clones existing in the system code.

Cloning is considered harmful because it increases the chance of unconscious, inconsistent changes, which can lead to faults in a system [12]. These changes can be detected when applying gapped clone detection. We found a number of such changes in the cloned code fragments, but we could not classify them as defects, because we lacked the knowledge needed about the software systems. Also the project partners could not directly classify these discrepancies as defects, which confirms that gapped clone detection is a more resource demanding type of analysis. Nevertheless, in some clone instances we identified additional instructions or deviating conditional statements compared to other instances of the same clone class. Gapped clone detection does not go beyond method boundaries, since experiments showed that inconsistent clones that cross method boundaries in many cases did not capture semantically meaningful concepts [12]. This explains why metrics such as cloned units or clone coverage may differ from values observed with conventional clone detection. Tab. 4 shows the results of gapped clone detection.

RQ 2.2 – Perceived Usefulness Following the feedback obtained from the questionnaire, two enterprises had limited experience with clone detection, the others did not know about it at all (Q2). Three enterprises estimated the relevance of clone detection to their projects as very high, the others estimated it as medium relevant (Q3). Concerning Q3, one stated that “clones are necessary within short periods of development.” Finally, all enterprises evaluated the importance of using clone detection in their projects as medium to high and plan to introduce this technique in the future (Q5). For details see Tables 7 and 8 in Appendix A.

3.2 Bug Pattern Detection

RQ 1.1 – Technical Problems Following Sec. 2.3, we confirm that bug patterns are a powerful technique to gather a vast variety of information about potentially defective code. However, most of its effectiveness and efficiency is achieved through carefully done, project-specific fine-tuning of the many setscrews available.

First, the impact of findings on quality factors of interest and their consequences for the project (e.g. corrective actions, avoidance or tolerance) were difficult to determine by the tool-provided rule categories, the severity and confidence information. Based on our experience we identified the following study object characteristics this impact depends on:

- Required usage-level qualities, e.g., security, safety, performance, usability
- Required internal qualities, e.g., code maintainability, reusability
- Technologies, i.e., language, framework, platform, architectural style
- Criticality of the context the findings belong to, e.g., platform or driver code

Second, some rules exhibited many false positives, either because their technical way of detection is fuzzy or because a definitely precise finding is considered not relevant in a project-specific context. The latter case requires an in-depth

understanding of each of the rules, the impacts of findings and, subsequently, a proper redlining of rules as pedantry or, actually, irrelevant. We neither measured the rates of false positives nor investigated costs and benefits thereof as our focus lay on the identification of the most important findings only.

Third, due to restricted selection and filtering mechanisms in the tools as well as a bounded view of the SOs' life-cycles, we were hindered to apply and calibrate appropriate rule selectors and findings filters. We saw that the usefulness of results is crucially influenced by the conversion of project-specific information on rule impacts into queries for rule selection and findings filtering. The tools greatly differ in their abilities to achieve this task via their graphical or command-line interfaces.

We addressed the first two issues by group discussion also with our partners and improved rule selection and findings filtering to principally avoid the findings reports to get overloaded or prone to false positives of the second kind. Also, the third issue could only be largely compensated by manual efforts. As most finding reports were quite homogeneously encoded and technically well accessible, we utilised ConQAT to gain statistical information for higher-level quality metrics as listed in Step 3 of Sec. 2.5.

RQ 1.2 – Spent Effort We achieved the initial setup of a single bug pattern tool in less than an hour. This step required knowledge about the internal structure of the SO such as, e.g., its directory structure and third party code. We used the ConQAT framework to flexibly run the tools in a specific setting (Java only) and for further processing of the finding reports. Having good knowledge of this framework, we completed the analysis setup for an SO (selection of rules, adjustment of bug pattern parameters, framework setup, etc.) in about half a day.

The runs took between a minute and an hour depending on code size, rules selection and other parameters. Hence, bug pattern detection should at least be selectively included into automated build tasks. Part of the rules are computationally complex and some tools frequently required more than a gigabyte of memory. The manual effort after the runs can be split into review and recalibration. The review of a report took us a few minutes up to half an hour. Due to the short period of the life-cycle of the SOs we had insight into, we could not estimate the recalibration effort for the rule selector and the findings filter. An overview of the efforts can be found in Tab. 2.

RQ 2.1 – Found Defects We conducted bug pattern analysis in three selective tool settings according to Step 3 in Sec. 2.5, but only for one version of each SO. For all SOs the filtered finding reports confirmed the defects focused or expected by these settings. Without going into the quantities and details of single findings, we summarise language-specific results:

C# Upon the rules with highest numbers of findings, FxCop and Gendarme reported *empty exception handlers*, *visible constants*, and *poorly structured code*. There was only one consensually critical kind of findings related to

Tool (Lang.)	Rule (recommendations in parentheses)	Study Objects					Most affected Qualities
		1	2	3	4	5	
FxCop (C#)	Empty / general exception handlers	47	106	n	n	n	Maintainability
	Nested use of generic types	44	.	n	n	n	Maintainability
Gend- arme (C#)	Deep namespaces	35	.	n	n	n	Maintainability
	Visible constants	18	338	n	n	n	Security
	Extensively large classes	.	3	n	n	n	Maintainability
	Extensively long methods	.	17	n	n	n	Maintainability
	Suspicious type conversion	.	3	n	n	n	Correctness
Gend., PMD	Constructor calls overwritable method	8	.	x	x	x	Security, stability
Find- Bugs (Java)	Unused local variables	n	n	142	.	.	Maintainability
	Inefficient string manipulation	n	n	46	.	.	Performance
	Corrupted serialisable	n	n	55	.	.	Correctness
	Return values not validated	n	n	30	.	.	Correctness, sec.
	Access of a null pointer	n	n	.	.	1	Sec., stability
	Integer shift beyond 32 bits	n	n	.	.	4	Correctness
PMD (Java)	Empty method in abstract class	n	n	x	.	x	Maintainability
	Max. cyclomatic complexity (≤ 10)	n	n	78	156	216	Maintainability
	Extensive length / size / parameter count, too many methods / fields	n	n	.	x	x	Maintainability
ConQAT	Max. nested block depth (≤ 5)	13	11	19	17	14	Maintainability

Table 5. Overview of bug pattern results. Legend: Cells contain the number of findings or a maximum value, “n” ...not applicable, “.” ...not noticeable, “x” ...noticeable, but PMD did not offer an appropriate way to exactly count the many findings

correctness in SO 2, viz. unacceptable loss of precision through *wrong cast during an integer division* used for accounting calculations.

Java Upon the rules with highest numbers of findings, FindBugs and PMD reported *unused local variables*, *missing validation of return values*, *wrong use of serialisable*, and extensive *cyclomatic complexity*, *class/method size*, *nested block depth*, *parameter list*. There have only been two consensually critical findings, both in SO 5, related to correctness, viz. foreseeable *access of a null pointer* and an *integer shift beyond 32 bits* in a basic date/time component.

Independent of the programming language and concerning security and stability we frequently detected the pattern *constructor calls an overwritable method* in 4 of 5 SOs and found a number of defects related to *error prone handling of pointers*. Concerning maintainability the SOs exhibited *missing or unspecific handling of exceptions*, manifold *violation of code complexity metrics* and various forms of *unused code*. Details are shown in Tab. 5.

RQ 2.2 – Perceived Usefulness According to the questionnaire, all of the partners considered our bug pattern findings to be medium to highly relevant for their projects (Q3). The sample findings we presented during our final workshop were perceived as being non-critical for the success of the SOs but would have

been treated if they had been found by such tools during the development of these software systems. The low number of consensually critical findings correlated well with the fact that the technique was known to all partners and that most of them have good knowledge thereof and regularly used such tools in their projects, i.e. at least monthly, at milestone or release dates (Q1-2). However, three of them could gain additional education in this technique (Q4). Nevertheless, all of the enterprises decided to use bug patterns as an important QA instrument in their future projects (Q5). For details see Tab. 7 and 8 in Appendix A.

3.3 Architecture Conformance Analysis

RQ 1.1 – Technical Problems We observe two kinds of general problems that prevent or complicate each architectural analysis: The absence of an architecture documentation and the usage of dynamic patterns.

For two of the systems there was no documented architecture available. In one case the information was missing because the project was taken over from a different organisation that was not documenting the architecture at all. They reasoned that any later documentation of the system architecture would be too expensive for their enterprise. In another case the organisation was aware that their system was severely lacking any architectural documentation. Nevertheless they feared that the time involved and the sheer volume of code to be covered exceeds the benefits. The organisation additionally argued that they are afraid of having to update the documentation within several months as soon as the next release is coming out.

In SO 2 a dynamic architectural pattern is applied, where nearly no static dependencies could be found between defined components. All components belonging to the system are connected at run-time. Thus, our static analysis approach could not be applied.

Architecture conformance analysis needs two ingredients apart from the architecture documentation: The source code and the executables of a system. This could be a problem because the source has to be compilable to analyse it. Another technical problem occurred when using ConQAT. Dependencies to components solely existing as executables were not recognised by the tool. For that reason all rules belonging to compiled components could not be analysed.

Beside these problems we could apply our static analysis approach to two systems without any technical problems. An overview of all SOs with respect to their architectural properties can be found in Tab. 6.

RQ 1.2 – Spent Effort For each system the initial configuration of ConQAT and the creation of the reflexion model in ConQAT could be done in less than one hour. Tab. 6 shows the number of modelled components and the rules that were needed to describe their allowed connections. The analysis process itself finished in less than ten seconds. The time needed for the interpretation of the analysis results is of course dependent on the amount of defects found. For each defect we were able to find the causal code parts within one minute. We expect that the effort needed for bigger systems will only increase linearly but staying small in

SO	Architecture	Version	Violating Component Relationships	Violating Class Relationships
1	12 Components 20 Rules	I	1	5
		II	3	9
		III	2	8
2	dynamic	n/a	n/a	n/a
3	undocumented	n/a	n/a	n/a
4	14 Components 9 Rules	I	0	0
		II	1	1
		III	2	4
5	undocumented	n/a	n/a	n/a

Table 6. Architectural characteristics of the study objects

comparison to the benefit that can be achieved using architecture conformance analysis as illustrated in Sec. 2.3. An overview of the efforts can be found in Tab. 2.

RQ 2.1 – Found Defects As shown in Tab. 6 we observed several discrepancies in the analysed SOs over nearly all version. Only one version did not contain architectural violations. Overall, we found three types of defects in the analysed systems. Each defect represents a code location showing a discrepancy to the documented architecture. The two analysable SOs had architectural defects which could be avoided if this technique had been applied. In the following we explain the types of defects we classified together with the responsible enterprises. The companies rated all findings as critical.

- *Circumvention of abstraction layers:* Abstraction layers (e.g. presentation layer) provide a common way to structure a system into logical parts. The defined layers are hierarchically dependent on each other, reducing the complexity in each layer and allowing to benefit from structural properties like exchangeability or flexible deployment of each layer. These benefits vanish when the layer concept is harmed by dependencies between layers that are not connected to each other. In our case e.g. the usage of the data layer from the presentation layer was a typical defect we found in the analysed systems.
- *Circular dependencies:* We found undocumented circular dependencies between two components. We consider these dependencies – whether or not documented – as defects themselves, because they affect the general principle of component design. Two components that are dependent on each other can only be used together and can thus be considered as one component, which contradicts the goal of a well designed architecture. The reuse of these components is strongly restricted. They are harder to understand and to maintain.
- *Undocumented use of common functionality:* Every system has a set of common functionality (e.g. date manipulation) which is often grouped into components and used across the whole system. Consequently, it is important to know where this functionality is actually used inside a system. Our observa-

tion showed that there were such dependencies that were not covered by the architecture.

RQ 2.2 – Perceived Usefulness Following the feedback gained from the questionnaire, we observed that 4 of the 5 participating enterprises did not know about the possibility of automated architecture conformance analysis (Q1). Only one of them already checked the architecture of their systems, however in a manual way and less frequently. Confronted with the results of the analysis all enterprises rated the relevance of the presented technique medium to highly relevant (Q3). One of them stated that as a new project member it is easier to become acquainted with a software system if its architecture conforms to its documented specification. All enterprises agreed on the usefulness of this technique and plan its future application in their projects (Q5). Details of the questionnaire can be found in Tab. 7 and 8 in Appendix A.

4 Discussion

General Observations First, we observed that code clone detection and architecture conformance analysis have been quite new to our partners as opposed to bug pattern detection which was well known. This may result from the fact that style checking and simple bug pattern detection are standard features of modern development environments. However, we consider it as important to know that code clone detection can indicate critical and complex relationships residing in the code at minimum effort. We made our partners aware of the usefulness of architecture conformance analysis, both in the case of an available architecture specification and to reconstruct such a documentation.

Second, we conclude that all of the three techniques can be introduced and applied with resources affordable for small enterprises. We assume, that except for calibration phases at project initiation or after substantial product changes the effort of readjusting the settings for the techniques stays very low. This effort is compensated by the time earned through narrowing results to successively more relevant findings. Moreover, our partners perceived all of the discussed techniques as useful for their future projects.

Third, we perceive our analyses of the study objects as successful. We found large clone classes, a significant number of pattern-based bugs aside from smells and pedantry as well as unacceptable architecture violations.

Usage Guidelines During the repetitive conduct of Steps 2 and 3 of the procedure in Sec. 2.5 we gained a lot of experience in applying the chosen techniques. For their introduction and application to a new software project we consider the following generic procedure as very helpful:

- 1) Establish a project-specific *configuration*. This includes the choice, particularly for bug patterns, of appropriate rules aiming on relevant quality factors or just the strengthening of design or coding guidelines.

- 2) Define events for *measurement*, *findings filtering* and *documentation*. Filtering requires in-depth knowledge of the system and its critical components. For bug pattern detection this influences severity and confidence levels, and for architecture conformance analysis this influences the definition of allowed, tolerated, and forbidden dependencies.
- 3) Decide whether to *treat or tolerate* findings. This involves (i) the inspection of results and defective code, (ii) the issue of change requests for defect removal and, (iii) to assess efficiency, the documentation of efforts spent.
- 4) Determine whether and how defects can be *avoided* regarding lessons learned from defect treatment.
- 5) Strengthen *quality gates* by improved criteria, which follow patterns such as, e.g., “Clone coverage in critical code package *A* below *X*% prior to any bundled feature introduction.”, “No critical security errors with confidence $> Y\%$ according to tool *Z* for any release.”, or “No architecture violations originating from change sets of new features.”
- 6) For *project control* in the context of *continuous integration*, derive statistics and trends from findings reports by a quality control dashboard such as ConQAT.

5 Threats to Validity

In the following, we discuss threats to the validity of our results. We structure them in internal and external validity threats.

5.1 Internal Validity

First, a potential threat to the internal validity is that most of the project participants had little experience with the specific tools we were applying. This could give us additional technical problems, which would not have occurred with experts. Furthermore, the efforts are probably higher. We mitigated this risk by discussions with experts and we assume that the introduction in other companies would also not necessarily be performed by experts.

Second, we did not record exact details about the efforts we spent. We rather made order of magnitude estimations only. In our context we consider this threat as small as we do not require precise analyses of these efforts including time measurement.

Third, we did not completely check whether all defects we found have caused real problems such as, e.g., critical system failures during operation or significant budget overruns. Hence, there may be false positives. We reduced this risk by detailed inspections of the defects we listed.

Fourth, the questionnaire results could be wrong, because a participant either knowingly or unknowingly gave incorrect answers. We mitigated this threat by asking participants to be careful in filling it out and at the same time assured anonymity to them.

5.2 External Validity

As this is an experience report on a technology transfer project, the results are inherently difficult to generalise. We had five projects of SMEs all located in Germany. We also restricted our analysis to systems realised in Java and C# and only applied specific analysis tools for it. Hence, the problems, defects, and perceptions may be particular to this context.

Nevertheless, we think that most of our experiences are valid for other contexts as well. The companies, we have collaborated with, range in their size from only several to a hundred employees. The domains they build software for differ quite strongly. Finally, the tools are all prominent examples and had been used in industrial projects before. Only the restriction to two programming languages has a strong effect as for other languages there may exist rather different tools and defects. For instance, with bug pattern detection, Ahsan, Ferzund and Wotawa [1] report that characteristics of bug patterns may be language specific.

6 Related Work

In this research we concentrate on applying automated static analysis techniques to enable SMEs mitigate the risk of defect-related costs. Different from our approach, the research community devotes its attention primarily to software process improvement in SMEs. There are a number of papers covering this topic.

Kautz [14] developed and used metrics to evaluate how new practices and tools for configuration and change management were affecting the software process at three SMEs. This work considers that the key to successful software measurement is to make metrics meaningful and to tailor them to a particular organisation. We confirm that observation in the context of software measurement.

Von Wangenheim et al. [28] investigated the assessment of software processes in SMEs to improve these processes. They developed MARES, a set of guidelines for conducting an ISO/IEC 15504-conforming software process assessment, focused on small companies. We perceive the usage guidelines we reported as a potential bridge between automated static analysis and more general guidelines for software process improvement.

Hofer [10] states that only 10% of the analysed SMEs in Austrian software industry believe to suffer from a lack of methods. He concludes that appropriate tool support as well as the knowledge of methods is available. On the contrary, we argue that SMEs may not be aware of many effective methods and can therefore not estimate their lack concerning these techniques.

Returning to automated static analysis techniques, to the best of our knowledge, multiple techniques have never been applied in a study in an SME context. However, there are several publications in which such techniques were investigated separately and in other contexts:

Lague et al. [18] report on application of function clone detection to a large telecommunication software system. As opposed to that, we do not limit clone

detection to the comparison of functions but compare arbitrary code fragments with each other. In this work we also did not analyse large systems. Nevertheless, we came to the similar conclusion that clone detection has potential to improve software quality.

Lanubile and Mallardo [19] performed research on finding clones in web applications developed using markup and programming languages. As mentioned earlier, our approach is technically limited in analysing such software systems. Introducing a semi-automatic approach presented by Lanubile and Mallardo could remove this limitation.

Ayewah et al. [2] evaluate the accuracy and value of FindBugs findings and discuss but not solve the problem of properly filtering false positives. They use the term *trivial bugs* for what we call smells and pedantry. We confirm their conclusions on the usefulness of findings and believe that an application of bug pattern detection has to undergo calibration guided by the staff of a software project. Moreover, by answering RQ2, we contribute to Foster’s, Hicks’ and Pugh’s [9] question “Are the defects reported by [static analysis] tools important?”.

Ferzund, Ahsan and Wotawa [7] report on the effectiveness of rules for smell detection. The rules they developed are based on machine learning and source file statistics provided by static code metrics. They used training information from two software projects including bug databases. We did not address the estimation of rule effectiveness but focused on their selection and application.

Wagner et al. [30] similarly applied FindBugs and PMD to two industrial projects. They could not find defects reported from the field that are covered by bug pattern detection. However, our results show that this technique indeed captures critical defects that may eventually occur in the field.

Rosik et al. [25] conducted an industrial case study on architecture conformance with three participating software engineers. They conclude that this technique should be integrated into the software engineering process and applied continuously. We think that the procedure we presented is able to satisfy their needs, because it explicitly focuses on continuous integration.

Mattsson et al. [21] illustrate their experience in an industrial project and the huge effort that is needed to keep the architectural model in conformance with the implementation. However, they tried to reach this goal in a manual way. Our results show that automation can dramatically reduce efforts.

Feilkas, Ratiu and Juergens [6] analysed three .NET platform projects of Munich Re very similar to our procedure, but they analysed the effects of the loss of architectural knowledge. Compared to our results they report a much higher effort of about five days to apply the technique, mainly because of time consuming discussions. We think that the lower effort we are reporting is mainly caused by the fact that we were collaborating with small enterprises and experienced a lower communication overhead.

7 Conclusions and Future Work

In general, it is most effective to combine different QA techniques to find most of the defects [20]. This, however, comes at the efforts and costs of performing many different techniques. Particularly, SMEs have difficulties in assigning large efforts to diverse QA provisions and to training specialists for them. Automated static analysis techniques promise to be an efficient contribution to software QA, because they only require little effort for their application.

We reported our experience in applying three static analysis techniques to small enterprises: code clone detection, bug pattern detection and architecture conformance analysis. Consequently, we assessed potential barriers for introducing these techniques as well as the observations we could make in a one-year project with five German SMEs.

We found several technical problems, such as multi-language projects with single language clone analysis or false positives, but we believe that these are no major road blocks for the adoption of static analysis. Overall, the effort for introducing the analyses was small. Most techniques were set up with an effort of less than one person-hour. We found various defects, such as high levels of cloning, null pointer access, erroneous calculations or circumvention of architecture layers. In the end, our partners found all of the presented techniques relevant for inclusion into their quality assurance processes.

In our opinion static analysis tools can efficiently improve quality assurance in SMEs, if they are continuously used throughout the development process and are technically well integrated into the tool landscape. But as our research was not focused on long term observations we can not address this issue. Consequently it is an interesting area of future work to investigate the long term effects of static analyses in SMEs' software projects and their continuous integration into their development processes. Questions arising from the application of these techniques such as their long term efficiency, their inclusion into an overall QA strategy, their acceptance by developers, their application to non-code development artefacts or their effects on the daily work could then be investigated.

We will continue to work in this area to better understand the needs of SMEs and investigate our current findings.

Acknowledgements

We would like to thank Christian Pfaller, Bernhard Schätz and Elmar Jürgens for their technical and organisational support throughout the project. We thank all involved companies for their reproach-less collaboration and assistance.

References

1. S. N. Ahsan, J. Ferzund, and F. Wotawa. Are there language specific bug patterns? Results obtained from a case study using Mozilla. In *Proc. Fourth International Conference on Software Engineering Advances (ICSEA'09)*, pages 210–215. IEEE Computer Society, 2009.

2. N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proc. 7th Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*, pages 1–8. ACM Press, 2007.
3. O. de Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble. .QL for source code analysis. In *Proc. Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 3–16. IEEE Computer Society, 2007.
4. F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens. Flexible architecture conformance assessment with ConQAT. In *Proc. 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, pages 247–250. ACM Press, 2010.
5. European Commission. Commission recommendation of 6 May 2003 concerning the definition of micro, small and medium-sized enterprises. *Official Journal of the European Union*, L 124:36–41, May 2003.
6. M. Feilkas, D. Ratiu, and E. Juergens. The loss of architectural knowledge during system evolution: An industrial case study. In *Proc. IEEE 17th International Conference on Program Comprehension (ICPC'09)*, pages 188–197. IEEE Computer Society, 2009.
7. J. Ferzund, S. N. Ahsan, and F. Wotawa. Analysing bug prediction capabilities of static code metrics in open source software. In *Proc. International Conferences on Software Process and Product Measurement (IWSM/Metrikon/Mensura '08)*, volume 5338 of *LNC3*, pages 331–343. Springer, 2008.
8. R. Fiutem and G. Antoniol. Identifying design-code inconsistencies in object-oriented software: A case study. In *Proc. International Conference on Software Maintenance (ICSM'98)*. IEEE Computer Society, 1998.
9. J. Foster, M. Hicks, and W. Pugh. Improving software quality with static analysis. In *Proc. 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'07)*, pages 83–84. ACM Press, 2007.
10. C. Hofer. Software development in Austria: Results of an empirical study among small and very small enterprises. In *Proc. 28th Euromicro Conference*, pages 361–366. IEEE Computer Society, 2002.
11. E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective – A workbench for clone detection research. In *Proc. 31th International Conference on Software Engineering (ICSE'09)*, pages 603–606. IEEE Computer Society, 2009.
12. E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proc. 31th International Conference on Software Engineering (ICSE'09)*, pages 485–495. IEEE Computer Society, 2009.
13. E. Juergens and N. Göde. Achieving accurate clone detection results. In *Proceedings 4th International Workshop on Software Clones*, pages 1–8. ACM Press, 2010.
14. K. Kautz. Making sense of measurement for small organizations. *IEEE Software*, 16:14–20, 1999.
15. J. Knodel and D. Popescu. A comparison of static architecture compliance checking approaches. In *Proc. IEEE/IFIP Working Conference on Software Architecture (WICSA'07)*, pages 12–12. IEEE Computer Society, 2007.
16. R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*, Schloss Dagstuhl, Germany, 2007.
17. R. Koschke and D. Simon. Hierarchical reflexion models. In *Proc. 10th Working Conference on Reverse Engineering (WCRE'03)*, page 368. IEEE Computer Society, 2003.

18. B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proc. International Conference on Software Maintenance (ICSM'97)*, pages 314–321. IEEE Computer Society, 1997.
19. F. Lanubile and T. Mallardo. Finding function clones in web applications. In *Proc. 7th European Conference on Software Maintenance and Reengineering (CSMR 2003)*, pages 379–388. IEEE Computer Society, 2003.
20. B. Littlewood, P. T. Popov, L. Strigini, and N. Shryane. Modeling the effects of combining diverse software fault detection techniques. *IEEE Transactions on Software Engineering*, 26:1157–67, December 2000.
21. A. Mattsson, B. Lundell, B. Lings, and B. Fitzgerald. Experiences from representing software architecture in a large industrial project using model driven development. In *Proc. Second Workshop on SHaring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent (SHARK-ADI '07)*. IEEE Computer Society, 2007.
22. A. Mishra and D. Mishra. Software quality assurance models in small and medium organisations: A comparison. *International Journal of Information Technology and Management*, 5(1):4–20, 2006.
23. L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. das Chagas Mendonca. Static architecture-conformance checking: An illustrative overview. *IEEE Software*, 27:82–9, September 2010.
24. I. Richardson and C. Von Wangenheim. Guest editors' introduction: Why are small software organizations different? *IEEE Software*, 24(1):18–22, Jan. 2007.
25. J. Rosik, A. Le Gear, J. Buckley, and M. Babar. An industrial case study of architecture conformance. In *Proc. 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '08)*, pages 80–89. ACM Press, 2008.
26. C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, Queen's University at Kingston, 2007.
27. N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proc. 20th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*, pages 167–176. ACM Press, 2005.
28. C. G. von Wangenheim, A. Anacleto, and C. F. Salviano. Helping small companies assess software processes. *IEEE Software*, 23:91–8, 2006.
29. S. Wagner. Defect classification and defect types revisited. In *Proc. 2008 Workshop on Defects in Large Software Systems (DEFACTS 2008)*, pages 39–40. ACM Press, 2008.
30. S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. An evaluation of two bug pattern tools for java. In *Proc. First International Conference on Software Testing, Verification, and Validation (ICST 2008)*, pages 248–257. IEEE Computer Society, 2008.
31. S. Wagner, J. Juerjens, C. Koller, and P. Trischberger. Comparing bug finding tools with reviews and tests. In *Proc. 17th International Conference on Testing of Communicating Systems (TestCom '05)*, volume 3502 of *LNCS*, pages 40–55, 2005.
32. Wikipedia. List of tools for static code analysis — wikipedia, the free encyclopedia, 2011. [Online; accessed 6-May-2011].
33. J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32:240–253, 2006.

A Results of the Questionnaire

Question	Closed Answers (without comments)						
		<i>daily</i>	<i>weekly</i>	<i>monthly</i>	<i>less freq.</i>	<i>never</i>	
Q1) Which of these static analysis techniques have you already been using in your projects?	Architecture conformance	0	0	0	1	4	
	Bug pattern detection	2	2	1	0	0	
	Clone detection	0	0	0	2	3	
Q2) What is your estimate of the experience of your company in these techniques?	Architecture conformance	++ +	o	-	- -	none	
	Bug pattern detection	1	2	1	1	0	0
	Clone detection	1	3	1	0	0	0
		0	0	1	0	1	3
Q3) How do you perceive the relevance of our analysis results for your study object?	Architecture conformance	high	o	low	none		
	Bug pattern detection	3	2	0	0		
	Clone detection	2	3	0	0		
		3	2	0	0		
Q4) How much education could you gain from the topics of our research project?	Architecture conformance	much	o	little	none		
	Bug pattern detection	2	2	1	0	0	0
	Clone detection	2	0	1	1	1	0
		2	2	1	0	0	0
Q5) Which of the following analysis techniques do you plan to apply at which level of priority?	Architecture conformance	++ +	o	-	- -	none	*)
	Bug pattern detection	1	3	0	1	0	0
	Clone detection	4	1	0	0	0	0
		0	2	3	0	0	0
*) application of the technique is planned							

Table 7. Summary of closed answers of the questionnaire for RQ 2.2 (five results, contents and answers have been translated from German to English). Legend: ++ .. very high, + .. high, o .. medium, - .. low, - - .. very low

Open Answers and Comments

Q1) Architecture conformance analysis has not been used because ...

- “projects have been developed cleanly or without [need of] architecture.”
- “manual inspection was carried through.”
- “the prerequisites ... would have needed to be established for our projects. Manual inspection (code reviews) already takes place irregularly.”
- “it was not known to us.”

Clone detection has not been used because ...

- “[clones were] not known to us as a problem.”
- “we did not recognise its necessity.”

Q3) The results have been relevant because ...

- “manual [code] analysis is significantly more cost-intensive, ...clone detection is only feasible with tool support.”
- “we learned about concepts, experiences and tools ...it is easier to become acquainted with [a project if its architecture conforms to its documented specification].”
- “Clones are necessary within short periods of development.”

Q5) “The results of this research project shall be included into our internal development process.”

Q6) *Your estimate of the current status of your organisation w.r.t. software quality:*
Strengths: “Seamless process for requirements QA ...regarded design guidelines for all languages used ...flexible adaptation of guidelines to customer needs ...performed QA provisions (from unit testing to selective pair programming) seem to work ...so far we only experienced high customer satisfaction ...mature in testing techniques and management.”

Weaknesses: “No consequent QA provisions ...no systematic QA ...automation and tool usage either project specific or even left out ...still learning to apply the tools.”

Q7) *Where do you expect the highest potential of your organisation to improve its software quality?*

- “Consequent QA provisions,”
- “integrated tools and more automation ... QA dashboard for project managers,”
- “better knowledge transfer between teams and projects,”
- “improved quality control ... backflow of QA results into development process.”

Q8) *Your estimate of the usefulness of static analysis for your software projects:*

Positive: “Important”, “high”, “trend analyses are important”, “very important, because of early and efficient defect detection ... help identify structural deficits ... ease [code] maintenance ... quality improvement starting with first build ... for internal projects better control and indication of deficits.”

Negative: “Often not feasible in projects externally conducted at the customers’.”

Table 8. Summary of open answers and comments of the questionnaire for RQ 2.2 (five results, contents and answers have been translated from German to English)