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

Mario Gleirscher[1], Dmitriy Golubitskiy[1], Maximilian Irlbeck[1], and
Stefan Wagner[2]

[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 could meet this challenge 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 illustrate the effort that was needed to introduce those techniques, what kind of defects could be found and how the participating companies perceived the usefulness of the presented techniques as well as our analysis results.

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 the worldwide 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 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, we believe that automated static analysis techniques can be suitable for SMEs. The benefits of such techniques lie in their low-cost application and their potential to detect critical quality defects. Those defects are a risk for the further development and cause higher 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 actually 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 may come across and how the participating enterprises evaluate the presented techniques as well as the defects we discover 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 our contact to 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. 2.4. Following the definition of the European Commission [6], 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 have some special circumstances, such as more 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 that down into two subquestions:

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 a small enterprise, which cannot afford to reserve extra capacities employee 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 at small enterprises, 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 subquestions:

RQ 2.1 *Which defects can be found?*

First, 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 useful.

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

Second, 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 will they apply them continuously in the future? This way, we augment the information we have from the analysis of the defects.

2.3 Static Analysis Techniques

Static analysis is the checking of software 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 can give enormous benefits. 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 ConQAT[1] for code clone detection and architecture conformance analysis as well as for results processing of bug pattern detection.

Code Clone Detection Modern programming languages 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:

[1] `http://www.conqat.org`

- 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 required effort.
- If changes to duplicated source code fragments are performed inconsistently, this 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 statement is part of a clone. 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 [13].

We differentiate between conventional clone detection and gapped clone detection. During conventional clone detection, clones are considered to be syntactically identical 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 clone detection per se is an indicator of bad design, the difference between the two approaches is that only the results of gapped clone detection can reveal real 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 CCFinder[2], ConQAT, CloneDR[3], and Axivion Bauhaus Suite[4]. 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 to other frequently used software quality assurance techniques such as code reviews or testing [31]. They are considered as cost-efficient [30]. Bug patterns are a scalable approach to efficiently reveal defects or possible causes thereof. 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 gain as much knowledge as required from a static perspective. This knowledge encompasses obvious bugs, rather complex heuristics for latent defects, e.g. code clones (focussed 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. One reason 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 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.

[2] `http://www.ccfinder.net`

[3] `http://www.semanticdesigns.com/Products/Clone`

[4] `http://www.axivion.com`

Many of the rules are realised by means of individual lexer and parser algorithms, by using compiler infrastructures, or by more generic means such as pattern or rule languages and machine-learning. Rules for latent defects and coding style often stem from more abstract source code metrics as, e.g., realized in Ferzund, Ahsan, and Wotawa [8]. Among the wide variety of tools [32] available for bug pattern detection, free and more popular ones are, e.g., splint[5] for C, cppcheck[6] for C++, FindBugs[7] for Java as well as FxCop[8] for C#.

Architecture Conformance The effect of architectural erosion is a widely documented problem [7, 9, 25]. Architectural knowledge erodes or even gets lost during the lifetime of a system. Accordingly, the documented and implemented architecture are drifting apart from each other. This effect leads to a downward spiralling maintainability of the system. In some cases the effort needed to reimplement 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 techniques existing for architecture conformance analysis: *Reflexion Models (RM)*, *Source Code Query Languages (SCQL)* and *Dependency Structure Matrices (DSM).*

RM [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]. This technique is also used by the commercial tools SonarJ[9] and Structure101[10] as well as the open-source tools ConQAT and dependometer[11].

There are tools using *SCQL* like Semmle.QL [4] or *DSM* like Lattix [27], for the sake of brevity not further explained here. Both of these techniques rely strongly on the realisation of the system and cannot provide an architecture specification that is independent of the system's implementation [5].

2.4 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.

[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`

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, 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 discuss organisational issues. 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 later consolidate them. Afterwards, we consolidate all interview results to find commonalities and differences. Finally, we have one or two consolidation workshops to discuss our results and plan the further procedure.

Step 2: Raw Data Collection The *source code* of at least three versions of the study objects, e.g. major releases, is retrieved for the application of the chosen techniques for RQ 1. For bug pattern detection and architecture conformance analyses, we retrieve executables packed with debug symbols for each of these configurations. We create the required *debug builds* on our own as far as the build environment is available. 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 study objects. The following explains how this is accomplished:

Code Clone Detection We use the clone detection feature [12] of ConQAT 2.7 for all study objects. 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. We use 10 as the value of the minimal clone length, 1 as the maximal allowed number of gaps per clone and 30% as the maximal relative size of a gap in our analysis based on initial experimentation. 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 the unit coverage (cf. Sec. 2.3) evolves in subsequent versions.

Bug Pattern Detection For Java-based systems we use FindBugs 1.3.9 and PMD[12] 4.2.5. In C#.NET contexts we use Gendarme[13] 2.6.0 and FxCop 10.0. We apply three tool settings we consider relevant for the study objects to gain two alternative quality perspectives:

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

To simplify the intricate 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. depth of inheritance, nested blocks or comment quality) and 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.

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 more detailed description of this ConQAT feature can be found in [5]. In summary, we use a reflexion model where dependency and hierarchy relations between components can be expressed. In 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 with the reflexion model. Every existing dependency that is not covered by the architectural rules represents a defect. The tool highlights each violating code location which can therefore be easily accessed. 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.

[12] `http://pmd.sourceforge.net`
[13] `http://www.mono-project.com/Gendarme`

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

Table 1. Study objects

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 small enterprises: Do they plan to use the presented techniques in future projects? Therefore we perform a survey on our study subjects using questionnaires. We selected the contained questions to contribute to RQ 2.2. The executive managers of each enterprise in their role as the representatives for the whole company then fill out the questionnaire which we evaluate.

2.5 Study Subjects and Objects

Study subjects For our investigation, we collaborate with five small enterprises. Overall, the companies cover various business and technology domains, e.g. corporate finance, communal planning, public document management, or 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 substitutional chose the humanitarian open-source system OpenMRS, a development of the equally named multi-institution, non-profit collaborative [1]. Hence, our study objects are the five software systems briefly described in Tab. 1. These software systems exhibit a code size between 100 and 600 kLoC. Their development is conducted and/or audited by a small organization and started at most seven years ago. The project teams involve 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.

3 Results

We held the information event of step 1 of our procedure (cf. Sec. 2.4) in July 2009 and invited a large number of SMEs of which finally 12 participated. From these companies, five committed to take part in the project. We conducted the kick-off in March 2010, the interviews from March to July and then finally two consolidation workshops in July 2010. We did not further analyse their outcome for this paper but used it as selection criterion for the static analysis techniques and to interpret our further results.

In the following we portrait for each technique how we contribute to the presented 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.

The major issue was analysing projects containing both markup and code like ones developed with JSP or ASP.Net. Since ConQAT supports either a programming language or a markup language during a single analysis, it would need 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 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 of the analysed software systems a large part of the code was generated by the ANTLR parser generator. 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 minimal compared to the other two static analysis techniques under study. The ease of introduction of clone detection is achieved due to the minimalistic configuration of the analysis which in the simplest case includes the path to the source code and the minimal length of a clone.

In the five analysed software systems it never took longer 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 then 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 the further analysis runs.

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 to some degree in all analysed systems: cloning of exception handling code, cloning of logging code and cloning of interface implementation by different classes.

As opposed to conventional clones, results of gapped clone detection could contain real defects, which arise through unconscious, inconsistent changes in

SO	Unit Coverage [%]
1	24
2	37
3	26
4	14
5	79

Table 2. Unit coverage in the study objects

instances of the cloned code. We found a number of such discrepancies in the cloned code fragments, but we could not classify them as defects, because we lacked the needed knowledge about the software systems. Also the project partners could not classify these discrepancies as defects directly, which confirms that gapped clone detection is a more resource demanding type of analysis.

Unit coverage, defined in Sec. 2.3, varied in the analysed systems between 14% and 79% as seen in Tab. 2. Koschke [16] reports on several case studies with unit coverage values between 7 and 23% and one case study with a value of 56%, which he defines as extreme. Therefore, the study objects 1, 3 and 4 contain normal clone rates according to Koschke. The clone rate in the study object 2 is higher than the rates reported by Koschke and the clone rate in the study object 5 is extreme.

RQ 2.2 – Perceived Usefulness Following the feedback obtained from the questionnaire, we observed that two enterprises had a limited experience with clone detection, others did not know about it. Although all enterprises rated the results of clone detection medium to highly encouraging, only one enterprise estimated the relevance of clone detection to their projects as very high, others estimated it as medium relevant. Nevertheless, all enterprises evaluated the importance of using clone detection in their projects as medium important to very important and plan to introduce this technique in the future.

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 code

Second, rules exhibited significant rates of 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.

Third, due to restricted selection and filtering mechanisms in the tools as well as a bounded view of the study objects' 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 all finding reports were quite homogeneously encoded and technically well accessible, we utilized ConQAT to gain statistical information for higher-level quality metrics.

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 study object 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 a study object (selection of rules, adjustment of bug pattern parameters, framework setup) 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 be 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 calibration. 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 study objects we had insight into, we could not estimate the calibration effort for the rule selector and the findings filter.

RQ 2.1 – Found Defects We conducted bug pattern analysis in three tool settings according to Sec. 2.4. For all study objects the filtered finding reports confirmed the defects focussed or expected by these settings. Without going into the quantities and details of single findings, we summarize 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 finding related to correctness, viz. unacceptable loss of precision through wrong cast during an integer division.

Java Upon the rules with highest numbers of findings, FindBugs and PMD reported *unused local variables*, *missing validation of return values*, *wrong use of serializable*, and extensive *cyclomatic complexity*, *class/method size*, *nested block depth*, *parameter list*. There have only been two consensually critical findings, both in the same study object, 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 reliability, we frequently detected the pattern *constructor calls an overwritable method* in 4 of 5 study objects and found a number of defects related to *error prone handling of pointers.* Concerning maintainability the study objects exhibited *missing or unspecific handling of exceptions*, manifold *violation of code complexity metrics* and various forms of *unused code.*

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. The low number of consensually critical findings correlates 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. However, three of them could gain additional education in this technique. Nevertheless, all of the enterprises decided to use bug patterns as an important quality assurance instrument in their future projects.

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 organization 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 organization was aware that their system was severely lacking any architectural documentation. Nevertheless they feared that the time involved and the sheer volume that would need to be covered exceeds the benefits. The organization argued additionally that they are afraid of having to update the documentation within several months when the next release is coming out.

One company uses a dynamic architectural pattern, where nearly no static dependencies could be found between defined components. All components belonging to the system are connected at runtime. Therefore our static analysis approach could not be applied.

SO	Architecture	# Components/Rules
1	documented	12/20
2	dynamic	n/a
3	undocumented	n/a
4	undocumented	n/a
5	documented	14/9

Table 3. Architectural characteristics of the study objects

Architecture conformance analysis needs apart from the architecture documentation two ingredients: 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 was occurring in the use of ConQAT. Dependencies to components existing in compiled sources only were not recognized 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 systems with respect to their architectural properties can be found in Tab. 3.

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. 3 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 comparison to the benefit that can be achieved using architecture conformance analysis as illustrated in Sec. 2.3.

RQ 2.1 – Found Defects Overall, we found three types of defects in the analysed systems. Each of the defects represents a code location which represents a discrepancy to the documented architecture. Every analysed system had architectural defects which could be avoided when applying static analysis techniques. In the following we explain the type of defects we classified together with the responsible enterprises. All findings were rated by the enterprises 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 these layers. 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 – documented or not

– 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 therefore be considered as one component, which contradicts the goal of a well designed architecture. The reuse of those 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 observation 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 5 of the participating enterprises knew nothing about the possibility of automated architecture conformance analysis. Only one enterprise was checking the architecture of their systems, however in a manual manner. Confronted with the results of the analysis all enterprises rated the relevance of the presented technique medium to highly relevant. All enterprises agreed on the usefulness of this technique and plan its future application in their projects.

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 architecture specification available and to reconstruct such a documentation.

Second, we conclude that all of the three techniques are introducible and applicable 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 from the procedure in Sec. 2.4 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) change requests for defect removal and, to assess efficiency, (iii) 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, e.g., 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 effort is 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, for most of the efforts we had, we did not document them in detail. We made, however, rough estimations. We see this threat as small as we do not perform precise analysis of the efforts but investigate the order of magnitude. It makes a difference if it takes hours or days, but we were not interested in the exact number of minutes.

Third, we have not checked whether the defects we found have caused real problems. Hence, there can be false positives. We mitigated this risk by detailed inspections of all 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 generalize. We had five projects of small enterprises all located in Germany. We also restricted ourselves only to projects in Java and C# and a certain set of tools. Hence, the problems, defects, and perceptions can be special for this context.

Nevertheless, we believe that most of our experiences are valid for other contexts as well. The companies, we have worked 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 in the programming languages has a strong effect as for other languages there can be completely different tools and defects. For example, for bug pattern detection, Ahsan, Ferzund and Wotawa [2] report that characteristics of bug patterns may be language specific.

6 Related Work

In this research we concentrate on applying automatic static analysis techniques to enable SMEs mitigate the risk of defect-related costs. Contrary to us, 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 small software companies. It is considered in this work that the key to successful software measurement is to make metrics meaningful and to tailor them to the organisation. We observed that software measurement should be tailored to the particular organisation.

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

Hofer [11] denotes 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 know many effective methods not well and can therefore not estimate their lack concerning these techniques.

Coming back to automatic static analysis techniques our literature review shows that multiple techniques have never been applied in an SME context. There is a number of publications, however, in which the 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. Contrary to that, we do not limit clone detection to comparing functions, but we compare any arbitrary code fragment with any other code fragment. Also we did not analyse large systems in our research. Nevertheless, we came to a similar conclusion that clone detection has a potential for improving the software system quality.

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

Ayewah et al. [3] 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 [10] question "Are the defects reported by [static analysis] tools important?".

Ferzund, Ahsan and Wotawa [8] 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 focussed 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 eventually occur in the field.

Rosik et al. [25] conducted an industrial case study on architecture conformance with three participating software engineers. They conclude this technique should be integrated into the software engineering process and analysed continuously. We believe 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 manner. Our results show that automation can dramatically reduce the needed effort.

Feilkas, Ratiu and Juergens [7] analysed three .NET projects of Munich Re very similar to our procedure, however they analysed the effects of the loss of architectural knowledge. Compared to our results they report a much higher effort (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

In general, it is most effective to combine different QA techniques to find the most defects [20]. This, however, comes at the cost of performing all these different techniques and spending the effort for that. Especially SMEs have difficulties in assigning a lot of effort to diverse QA techniques and to training specialists for them. Automated static analysis techniques promise to be an efficient contribution to software quality assurance, because they do only require low effort in 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 blockers for the adoption of static analysis. Overall, the effort for introducing the analyses was small. Most techniques were set up with an effort below one person-hour. We found diverse defects, such as a high level of cloning or a circumvention of the architecture layers. At the end, our partners found all of the presented techniques relevant enough for inclusion in their quality assurance process.

In our opinion tools realizing such techniques can efficiently improve quality assurance in SMEs, if they are countinuosly used throughout the development process and are technically well integrated into the tool landscape. 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 and Elmar Jürgens for their technical support throughout the project. Additionally we thank all involved companies for the good collaboration and assistance.

References

1. *OpenMRS – An Electronic Medical Record System Platform.* `http://openmrs.org`, [Online; accessed 29-April-2011].

2. S. N. Ahsan, J. Ferzund, and F. Wotawa. Are there language specific bug patterns? Results obtained from a case study using Mozilla. In K. Boness, J. M. Fernandes, J. G. Hall, R. J. Machado, and R. Oberhauser, editors, *ICSEA*, pages 210–5, Portugal, September 2009.
3. 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, 2007.
4. O. de Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble. .QL for source code analysis. In *Source Code Analysis and Manipulation (SCAM '07)*, 2007.
5. F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens. Flexible architecture conformance assessment with ConQAT. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 247–50, New York, NY, USA, 2010.
6. 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.
7. M. Feilkas, D. Ratiu, and E. Juergens. The loss of architectural knowledge during system evolution: An industrial case study. In *IEEE 17th International Conference on Program Comprehension, 2009. ICPC'09*, pages 188–97, 2009.
8. J. Ferzund, S. N. Ahsan, and F. Wotawa. Analysing bug prediction capabilities of static code metrics in open source software. In R. R. Dumke, R. Braungarten, G. Büren, A. Abran, and J. J. Cuadrado-Gallego, editors, *IWSM/Metrikon/Mensura*, volume 5338 of *Lecture Notes in Computer Science*, pages 331–43. Springer, 2008.
9. R. Fiutem and G. Antoniol. Identifying design-code inconsistencies in object-oriented software: A case study. In *Proc. Intl. Conf. on Software Maintenance (ICSM'98)*, Bethesda, Maryland, USA, November 1998.
10. 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*, pages 83–84, New York, NY, USA, 2007. ACM.
11. C. Hofer. Software development in austria: Results of an empirical study among small and very small enterprises. *EUROMICRO Conference*, 0:361, 2002.
12. E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective – A workbench for clone detection research. In *Proc. Intl. Conf. on Software Engineering (ICSE'09)*, pages 603–6, Los Alamitos, CA, USA, 2009.
13. E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE*, 2009.
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 *IEEE/IFIP Working Conference on Software Architecture (WICSA'07)*, pages 12–12, 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 *10th Working Conference on Reverse Engineering (WCRE'03)*, page 368, 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. Intl. Conf. on Software Maintenance*, 1997.

19. F. Lanubile and T. Mallardo. Finding function clones in web applications. In *7th European Conf. on Software Maintenance and Reengineering (CSMR 2003)*, page 379, 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 *Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent*, SHARK-ADI '07, page 6, Washington, DC, USA, 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*, pages 80–89, New York, NY, USA, 2008. ACM.
26. C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, Queens 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, New York, NY, USA, 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 P. T. Devanbu, B. Murphy, N. Nagappan, and T. Zimmermann, editors, *DEFECTS*, pages 39–40. ACM, 2008.
30. S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. An evaluation of two bug pattern tools for java. In *ICST*, pages 248–57. 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(4):240–53, 2006.