MASTER'S THESIS | LUND UNIVERSITY 2017

Implementing and Evaluating the Use of Automatic Tests for Architectural Rules

Johan Bäckström, Fredrik Karåker Sundström

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2017-20

Implementing and Evaluating the Use of Automatic Tests for Architectural Rules

Johan Bäckström
dic12jba@student.lu.se

Fredrik Karåker Sundström
dic12fsu@student.lu.se

September 11, 2017

Master's thesis work carried out at Robert Bosch AB.

Supervisors: Robert Lagerstedt, Robert.Lagerstedt@se.bosch.com
Elizabeth Bjarnason, Elizabeth.Bjarnason@cs.lth.se

Examiner: Martin Höst, Martin.Host@cs.lth.se

Abstract

The development of high quality software often demands good software architecture. The architecture is often partly defined by guidelines that needs to be communicated and enforced in the project. We have investigated how this can be achieved by implementing automatic checking of the architectural guidelines in the tool-chain of a software project. In this thesis we applied a design science approach to implement a solution for automatically checking architectural rules and evaluated this solution by applying it to a live project at Robert Bosch AB in Lund. We concluded that it is an efficient way of both communicating the guidelines to the developers as well as ensuring that they are followed. It was shown that this worked well as a compliment to the manual code reviews that were previously used in the project for ensuring that the architectural guidelines were followed. Only a third of the breaches found by the automatic checkers were found by the manual reviews which shows the potential and need for automation. The checkers can also save both time and effort for the architect and the developers since a lot of manual work can be avoided.

The results of this thesis can be used to motivate the further automation of architectural conformance checking in the industry and as an introduction to more extensive research in the field.

Keywords: Software Architectural Conformance, Architectural guidelines, Architectural Erosion, Static Code Analysis, Automatic Tests

Acknowledgements

We would like to thank Robert Bosch AB and our supervisor Robert Lagerstedt for the opportunity to do this master thesis, Elizabeth Bjarnason for great guidance and feedback during our work, and Fredrik Svedberg for his patience and help during the implementation.

Contents

1 Introduction **7**
- 1.1 Architectural Guidelines . . . 7
- 1.2 Purpose . . . 8
- 1.3 Outline . . . 9
- 1.4 Contributions . . . 9

2 Background and Related Work **11**
- 2.1 Architectural Erosion . . . 11
- 2.2 Architectural conformance . . . 12
  - 2.2.1 Architectural Conformance Checking . . . 12
  - 2.2.2 Approaches for Static Architecture Conformance Checking . . . 13
- 2.3 Architecture Description Languages . . . 14
  - 2.3.1 Dependency Constraint Language . . . 14
  - 2.3.2 inCode.Rules . . . 15
  - 2.3.3 Dictō . . . 15
- 2.4 Tools for Static Analysis . . . 16
  - 2.4.1 Dependency Analysis . . . 16
  - 2.4.2 Additional Tools . . . 17

3 The Project Environment at Bosch **19**
- 3.1 Continuous Integration . . . 19
- 3.2 The Architectural Guidelines . . . 20
- 3.3 The Developers' View of the Architectural Guidelines . . . 21

4 Research Method **25**
- 4.1 Exploring Previous Knowledge . . . 26
- 4.2 Understanding the Problem Space . . . 27
  - 4.2.1 Observations . . . 27
  - 4.2.2 Interviews . . . 27

4.3 Design and Implementation 28
4.4 Evaluation 29
4.4.1 Breach Metrics 29
4.4.2 Deployment of the Checkers in the Project 30

5 Solution Approach **31**
5.1 Categorisation of Architectural Rules (RQ1) 31
5.2 Solution 32
5.2.1 The Checker Implementation 32
5.2.2 The Rules Tested by the Checkers 35
5.2.3 Additional C++ Checker 38

6 Evaluation Results **41**
6.1 Breaches of Architectural Rules (RQ2) 41
6.2 Deployment in the Project (RQ3) 44

7 Discussion **47**
7.1 Categorisation of Architectural Rules 47
7.2 Affect on the Development Process 48
7.3 Reaction from the Developers 48
7.4 Validity 49
7.4.1 Evaluation Data 49
7.4.2 Interviews 50

8 Conclusions **51**
8.1 Conclusion 51
8.2 Future Work 52

Bibliography **55**

Appendix A Interview Guide **61**
A.1 Introduction 61
A.2 Questions 61

Chapter 1

Introduction

Software architecture can be defined as the way the components are structured and communicates with each other in a software system. It is the blueprint of how everything should fit together and how the system is intended to work on a higher level of abstraction. This makes it possible to view a system in terms of a collection of structures which represents different parts of the functionality. The architecture defines how these structures should behave in the system and what their purpose are. A structure can be of different magnitudes, some may be comprised of several thousand lines of code while others may just be a single or a few functions. If designed and enforced correctly a good architecture makes it easier to keep the software system organised as it is being developed. In order to ensure that the intended architecture is followed, the architecture has to be communicated to the developers, something that in many cases can be done with a set of architectural guidelines.

This master thesis is a case study where a design science approach has been used to investigate how static code analysis can be used in a live software project in order to check the architectural conformance of the code. It was done in a live software project at Bosch AB in Lund with the goal to improve the enforcement of their architectural guidelines and make it easier for the developers to follow them.

1.1 Architectural Guidelines

In large software development projects architectural rules and guidelines are often enforced to ensure that the source code conforms to the intended architecture. These rules help the developer to write code of good quality in the way that is anticipated by the architect and to meet non-functional requirements. Some examples of areas covered by architectural rules are module dependencies, file locations and naming conventions. There are however different ways of classifying many of these rules since they often can be regarded as coding rules as well. A coding rule is often described as a rule concerning details of the code and

specifics of how it should be written, but sometimes such details can impact architectural aspects as well. Naming conventions are an example of this grey area, which when used well can increase the readability of code, making it easier to understand by someone who has not written it. This could make it easier to make changes i.e. the maintainability is improved, an area highly connected to software architecture. Because of this, naming conventions and some other rules that may not be considered architectural rules by others are in this thesis regarded in the same context as architectural guidelines. It could be argued that these rules are not architectural since they do not concern the communication between components or modules but since they do have some impact on architectural areas this thesis considers them in this context.

How the architectural guidelines are organised and communicated in a project varies as well as the level of detail and coverage they provide. It is often the developers responsibility to keep these rules in mind during development and thus it is crucial that they have a good knowledge about them. The training of the developers can be time-consuming and is therefore sometimes over-looked. This is of course a problem since it can lead to unnecessary breaches of the architectural rules and thus negatively affect the conformance of the architecture. The fact that changes and additions can be continuously made to the guidelines only worsens the situation. If there is no way of detecting these breaches a long time can pass before they are discovered. This does not only mean that the code will have unfixed breaches during this time, but also that the developer potentially has begun coding a different component when the discovery is made. Once a breach is detected it may be difficult for the developer to fix the issue since he needs to context switch back to the earlier work. These problems can to some extent be solved by giving the developers some kind of direct feedback if their code breached any of the architectural guidelines. Providing such feedback can be achieved by adding automatic checkers to the development tool-chain which analyses the code according to the architectural rules that has been defined and notifies the developer of breaches. Placing the feedback closer to the developers will lead to a short feedback loop which will not only help to ensure the conformance of the architecture but at the same time educate the developer about the architectural guidelines [28].

1.2 Purpose

This project has studied the usage of architectural guidelines and how they are enforced in a software project at Bosch AB in Lund. The goal was to implement and evaluate automatic checkers based on these guidelines and integrate them in the existing tool chain of the project. The purpose of this was to provide direct feedback to the developers if a breach was detected in their code, giving them the possibility to correct it and at the same time improving their knowledge and understanding of the guidelines. The approach was then evaluated both by collecting data of architectural breaches in the project and running the checkers live in the tool-chain where developers could come in contact with their feedback.

The thesis aimed to answer the following research questions:

RQ1 How can architectural rules be categorised according to suitability for automatic checking?

RQ2 How can automatic checking of architectural rules affect the development process?

RQ3 How do developers perceive the existence of automatic checkers of architectural rules in the tool chain?

RQ1 was formulated since it was doubtful if different types of architectural rules could be tested in a satisfactory way or if some were more difficult than others. Answering this question would thus make it easier to find which rules that could be tested and to what extent.

RQ2 was formulated to see if automatic tests would be positive or not for the development process. It was also interesting to see if the automatic checkers could be used as an alternative way of communication of the architecture, and if this in turn could improve the development process.

Finally RQ3 was formulated to see how the developers reacted to the automatic checkers and if they liked or disliked them. Answering this question could together with any empirical evaluation motivate if the solution of this thesis was a suitable approach.

1.3 Outline

This report starts with a background chapter discussing some of the main terms and concepts of the field and the related work that has been carried out by others. Chapter 3 describes the project environment at Bosch and also includes results from the interviews with the developers. Chapter 4 reports about the research method of the thesis. Chapter 5 mainly concerns the implementation of the checkers and the two finishing chapters evaluates the approach and presents the conclusions the thesis.

1.4 Contributions

This master thesis has to a large extent been made in collaboration. Both Johan and Fredrik contributed to the literature study and many of the parts in the report was written in collaboration with a few exceptions. Sections 2.1, 2.2 and 2.4 were mainly written by Johan while the Sections 2.3 and 2.5 were mostly written by Fredrik. The interviews were designed and executed in collaboration as well as the analysis of the results. The implementation was partly divided among the two of us even though pair-programming was applied to a large extent. Johan wrote most of the FIDL checkers while Fredrik focused on the JavaScript checkers and XML parsing. The integration in the tool chain and the evaluation were made in collaboration.

Chapter 2
Background and Related Work

One characteristic of software system development is the constant change that the system goes through over time. To ensure maintainability it is therefore often important to specify a system architecture that should be followed when adding new components or doing modifications. However, as a system grows and many changes are made it can get increasingly difficult to follow the planned architecture. This makes it even harder to make future changes without violating the architecture, creating a vicious cycle. These violations may cause the system to deviate further and further from the intended architecture while lowering the quality of the code. This phenomenon has been widely discussed using many different terms such as architecture degeneration [29] [36], software aging [31], code decay [22] and architectural erosion [32]. While some distinctions have been made between the definitions of these terms this thesis uses architecture degeneration and architectural erosion interchangeably to describe violations of the intended architecture in a software system.

2.1 Architectural Erosion

Hochstein and Lindvall list a number of different causes for architectural degeneration gathered from Parnas, Eick et al. and Van Gurp and Bosch [26] [31] [22] [39]. Some of the causes most relevant for this master thesis are *rushed development*, *different competence among developers*, *not natural for developers to write modular code*, *poor handling of design documentation* and *difficulty to communicate the architecture* which are further described below.

One important cause mentioned is *rushed development* due to the time pressure developers often are under. The developers and management put so much focus on being ready for the next deadline that they take short cuts without keeping possible future consequences in mind. Even if reviewing of the code is utilised, making sure the functionality is in place can be more important than following the architectural rules if it could mean a

better delivery on the next deadline.

Another cause is the *different competence among the developers* working on a project. Architectural erosion can occur if a developer tries to make changes to complex code, written by a more skilled colleague, which he or she does not fully understand.

Furthermore a cause can also be that it is *not natural for the developers to write modular code*. The concept of modularity might be difficult to grasp which can lead to poor separation of the different modules. In an architecture based on modularity this could then lead to architectural erosion.

Architectural erosion is also more likely to occur if the *design documentation is not properly organized, incomplete or imprecise*. It could also be difficult to ensure that all developers in a project get information that is added or modified in the documentation. Verbal communication could perhaps help reduce the risk of architectural erosion due to poor documentation, but in larger software projects this would be impractical.

There are a large number of additional causes such as the architecture not being designed for change from the start, programming tools not being good enough for managing a project, imprecise requirements etc.

In the project where this thesis was carried out the *difficulty of communicating the architecture* and any modifications to it appeared to be the predominant cause. This difficulty was increased as the project proceeded due to more developers joining the teams and the responsibility of managing the architecture being placed on very few individuals.

There have been numerous approaches to deal with these problems and they can be divided into three main categories, attempting to prevent, minimise and repair architectural erosion [20]. The repair of architectural erosion often involves reverse engineering the implemented architecture from the source code and then comparing it to the intended architecture, finding and fixing the eroded parts. Attempts to prevent architectural erosion is focused on eradicating the possibility for any deviations between the planned and implemented architecture. One approach for achieving this is architecture to implementation linkage, which is done by establishing a continuous correlation between the architecture and the implementation. However, the category that is of interest for this thesis is mainly the attempt to minimize software erosion by architectural conformance monitoring. One way of doing this is by implementing checkers that detect violations of the architecture according to predefined rules and requirements. This will be discussed in the next section.

2.2 Architectural conformance

Architectural conformance, and the synonym architectural compliance, can be defined as the measure to which degree that the implemented software conforms to the properties and requirements of the intended architecture [27]. In other words it is the opposite of architectural erosion.

2.2.1 Architectural Conformance Checking

During architectural conformance checking (ACC) the goal is to monitor the source code and its behaviour in order to detect any deviations from the planned architecture. This type

of checking can be divided into two major categories, static checking and dynamic checking. Static checking is done by examining the source code, structure and configuration of the system and does not require execution of the code. With these checks it is possible to extract static architectural information such as class dependencies and file structures.

In order to verify the conformance of any dynamic properties dynamic checking is done using data extracted from the system during execution. By analysing the runtime-state of the system, properties such as dynamic loading and interactions in the system can be measured. The extracted data is then used for comparison with the planned architecture to check the conformance of the system. This approach has the ability to check many of the static properties as well but one challenge is to minimise the performance impact that the analysis have on the product since it needs to run in parallel with the system during execution [21].

This thesis will use the static approach by checking the conformance of the source code against pre-defined architectural rules.

2.2.2 Approaches for Static Architecture Conformance Checking

Static ACC requires analysis of the source code and there are different approaches for doing this, some of which are described in this section.

Dependency analysis

Knodel and Popescu [27] identified three main approaches for static architecture compliance checking: reflexion models, relation conformance rules and component access rules. Reflexion models uses two models of the software system, the planned architectural model and the implemented architecture derived from the source code. A mapping then has to be made between the two before they can be compared and possible conformance deviations detected. Relation conformance rules specifies the allowed or forbidden relation between two components. The mapping can here be made automatically and similar deviations as with reflexion models can be detected. With this approach it is also possible to specify allowed and forbidden relations to third party libraries. The last approach, component access rules, instead focuses on one component at a time and the communication that is allowed to it. This is done by specifying ports for the component which other components is able to call. This can for example be methods in a Java class that has to be public but is not intended to be called by any component.

Knodel and Popescu further states that the use of either one of these approaches in the static conformance checking will result in categorising the relations between components into three different types: convergence, divergence and absence. A convergence is a relation in the implemented system that is allowed by the planned architecture. Divergence is the opposite, a relation that does not conform to the planned architecture. Lastly, an absence is a relation in the planned architecture that is missing in the implementation.

Manual Code Reviews

Even though there are tools available for making ACC automatic most companies use manual reviews or do not test the architectural rules and constraints at all [17]. Manual reviews have the advantage of making it possible to write well formed comments and even identify mistakes hard to find by computer programs. An issue with manual code reviews is however that the responsibility of knowing the architectural rules and guidelines is placed on the person reviewing the code. If the reviewer does not have a good knowledge of these rules many errors and breaches could be missed. The effectiveness of the reviews are thus heavily dependent on the expertise of the reviewer [35] [24]. Other issues are time spent on carrying out the manual code reviews and also the quality of the reviews. It is not enough if only a small subset of the reviewers produce reviews of good quality since many breaches then could slip through and if reviewing code is too time consuming it could lead to less time for actual development.

Usage Areas

How the architectural rules and constraints are tested seems to be heavily dependent on what type of rule that is defined. Some areas where automatic tests are used the least are dependencies, recoverability, software update, hardware infrastructure, file location and naming conventions. Moreover the areas where automatic tests are used the most are authorization, throughput, response-time, data retention policy and authentication. What can be seen is that rules and constraints in dynamic areas are more often tested using automation while static properties are mostly checked using manual reviews [17].

2.3 Architecture Description Languages

One way to simplify and structure architectural rules and constraints in a way to make them easier to test is using a Domain Specific Language (DSL). In software architecture such a language can also be called an architecture description language (ADL). Using an ADL it is possible to formally and unambiguously describe the architecture of a software system thus creating a specification that can be tested easier than other informal approaches [18]. There is a vast amount of ADLs and this section only describes a couple of them that were found especially interesting, mainly since they all are research products. The three ADLs described also show how different such a language can be: one identifies issues and also suggests fixes for them, one is highly integrated in an IDE and is thus restricted in which environments it can be used and one has a more general approach.

2.3.1 Dependency Constraint Language

The dependency constraint language, DCL [38] focuses only on constraints regarding dependencies between different modules in object-oriented software systems. The language consists of three main elements: modules, divergences and absences. A module is a representation of an arbitrary number of classes. It can for example consist of all classes in a certain package of a project.

Divergences are different constraints used to identify divergences in the system. An example of a constraint designed to capture divergences is: `only A can-access B`, where A and B are modules. This declares that classes in module A are allowed to call methods and read or write to fields on non-private members in classes contained in module B.

Absences are constraints used to identify absences in the system. For example; in comparison to the `can-access` constraint in the divergences types there is a `cannot-access` constraint that can be used for absences.

In order to verify if the constraints declared using DCL are followed a tool called `dclcheck` has been implemented. It is an Eclipse plug-in which extracts all dependencies in a system, controls if the constraints declared with DCL are breached and outputs the result in an Eclipse view. Due to the implementation of the tool it is only usable in the Java environment but the DCL language is compatible with several object-oriented programming languages.

In compliment to the DCL language and the breach identification tool `dclcheck`, another tool called `dclfix` was created [37]. This tool uses the constraints defined in DCL, the source code of a project and the breaches detected by `dclcheck` as input and proceeds to output suitable actions that could solve the issues and violations.

2.3.2 inCode.Rules

The ADL inCode.Rules is designed solely for Java projects and specifically projects using the Eclipse IDE [30]. InCode.Rules is not only a language used to specify architectural constraints but also an Eclipse-integrated tool. This tool consists of an editor were you write the architectural rules and constraints, as well as a rule checker to control if any rules are broken. However, all of the functionality relies heavily on Java and the Eclipse environment which makes it unusable for projects using any other programming language or IDE.

Even though this programming language and IDE integration could be seen as a weakness, this approach is intended to bring the architecture closer to the developers. By integrating the validation of architecture constraints directly into the IDE the feedback loop becomes shorter compared to if it was placed at the build stage. A shorter feedback loop could make it easier for the developers to keep the architecture in mind during the development process and thus improve the code quality.

2.3.3 Dictō

An attempt to develop a DSL on a high-level of abstraction is currently being made under the name of Dictō and it aims to be used by software architects together with other project stakeholders to specify software architectures [16]. By making the syntax of the language less technical it is attempted to make it possible for non-technical stakeholders to have more influence over architecture rules and requirements.

Dictō is meant to be used together with Probō, a tool coordination framework which is used to verify the rules written in Dictō by using different third-party tools. The idea of this approach is to reduce the time needed learning different testing tools and instead let Probō manage these using adapters. The hope is then that these adapters, when implemented, can

be reused for multiple projects and thus eliminating much of the time needed to configure the different tools. It is however unclear if reuse of adapters will be possible in a large scale or if projects often will need to implement adapters that are specific for the current development. If new adapters always need to be implemented for new projects one still will need good knowledge and understanding of the tools meant to be used.

However, in an evaluation study where Caracciolo et al. [15] used Dictō and Probō on three different industrial projects they found that many of the constraints and rules were similar, which would mean reuse of adapters is possible, at least when the projects use the same underlying technology. They also found that their approach worked well for programming languages other than Java since one of the projects was developed using PHP. This however requires a parser for the targeted language, rework on analysers and new adapters written for Probō.

Another problem Dictō and Probō aims to solve is that tools often have different syntax when it comes to output of analysis. Probō is expected to solve this by merging all output into a single report making it easier for the user to analyse any results.

2.4 Tools for Static Analysis

Apart from the tools directly associated with an ADL there are a lot of other tools that can be used to check architectural rules and constraints as well as other static properties of source code. Many of these other tools focus on testing only a small subset of all possible rules while some try to be more general and can thus be applicable on a larger diversity of rules. In this section some examples of such tools will be discussed.

2.4.1 Dependency Analysis

One of the important areas of ACC is the checking of dependency constraints and violations against such constraints. There are many tools that focus on this area of dependency checking using static analysis.

Prujit et al. [33] carried out a study where they compared 10 different static analysis tools regarding how well the tools analyse dependencies in a system and report violations of dependency rules. The tools where tested in a benchmark test that the authors developed iteratively together with Java- students and lecturers. The result was 64 test cases aimed to see if a tool can find different types of dependencies. In addition to the benchmark test the tools were tested on the source code of the open source project Freemind in order to see how they performed on a real system. The Freemind project was selected since it contained a good variety and amount of dependencies when compared to four reference systems. The study showed that on average 77% of the tested dependencies were identified by the tools which is an indication that there is still room for much improvement in this area.

Two of the tools that performed well in Prujit et al's study were JITTAC [7] and Sonargraph [12]. JITTAC is a plugin to the Eclipse IDE that uses reflexion modelling on Java systems to check the conformance of the implemented architecture. It lets the user drag-and-drop source code entities into the model and then calculates the dependencies between them. Sonargraph is a commercial collection of tools which can be used in several ways,

both by the architect and the developers. It includes build integration and IDE plugins that help developers to ensure that their code conforms to the architecture before it is committed. The architect tool helps the architect to design the system and monitor its performance and solve problems such as duplicated code and cyclic dependencies.

2.4.2 Additional Tools

The study carried out by Prujit et al. [33] only mentions a few of the available tools for static analysis. In this section some additional tools are discussed that are especially related to this thesis.

CppDepend

CppDepend is a very comprehensive tool for analysing C and C++ code [3]. It includes several different features such as a large number of code quality metrics and diagnostics as well as analysis of the software architecture. It is built on Clang [2], a front-end to the LLVM compiler, which it uses to parse and statically analyse the source code. Clang offers hundreds of diagnostics in it self which covers a lot of potential problems and these are of course used in CppDepend. More than 80 quality metrics are also included ranging from counting the lines of code in components to different complexity metrics. Another extensive feature is CQLinq which lets you query the code base using LINQ queries [8]. This enables you to extract information from the code in the same way you do from an SQL database. A CQLinq editor is also provided to support the writing of the queries.

It also includes a number of ways to analyse and visualise the architectural dependencies in the source code. This gives you the possibility to ensure the architectural conformance of the code and get an overview of all components and how they communicate.

Semmle

Semmle [11] is a software engineering analysis tool which extracts data from the source code and puts it in a database. The database can then be queried using the included query language. This can be used for all kinds of static analysis such as finding all instances of a security vulnerability, reporting code metrics and ensuring correct usage of APIs. It includes query libraries for most of the common programming languages including Java, C++, C, C# and Python.

A case study was carried out by De Schutter [19] demonstrating the potential of the Semmle tool when automating architectural reviews. In the study a team of developers used Semmle over the course of a year while continuously writing custom queries to both check compliance with coding standards and ensure architectural conformance. The conclusion of the study was that the team had successfully implemented automatic checkers that kept the source code in sync with the intended architecture. The main reason stated for this was the possibility to write specialised queries that could provide results that were of high relevance to the specific project.

SonarQube

SonarQube [14] is a platform for continuous code quality. It offers code analysers for over 20 programming languages and a common interface in which you can monitor the quality of your project. It can be used for finding bugs and vulnerabilities as well as calculating quality metrics. It can be integrated with build systems like Maven and MakeFile and it is therefore easy to schedule the analysis from Continuous Integration engines like Jenkins. An example of the graphical interface can be seen in Figure 2.1

An IDE plugin called SonarLint [13] is also available which can give direct feedback to the developer when specific code and style rules are broken and can also be connected to a SonarQube server in order to utilize the full capabilities of the SonarQube analysers. It is however only available for Java, JavaScript and PHP.

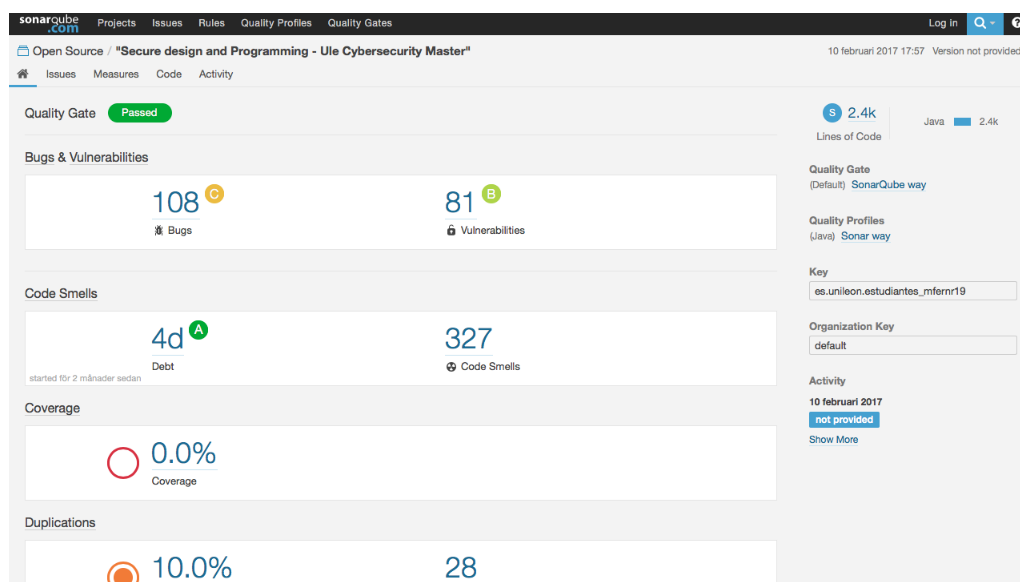

Figure 2.1: A project view in Sonarqube.

Chapter 3
The Project Environment at Bosch

This master thesis was done in an embedded software project at Robert Bosch AB in Lund. Bosch is a large company with interests in many different areas such as household appliances, security systems and power tools. They are also the largest independent subcontractor in the world when it comes to developing hardware and software technology for the automotive industry [1].

The case project for this thesis was a subproject in a larger project which included both hardware and software components. At the start of the project the case project had run for approximately 1 year. The Scaled Agile Framework (SAFe) [10], an agile software development framework intended for Lean-Agile development on enterprise scale, was utilised in the case project. This meant a scrum-like development process with 2 week sprints and 8 week increments. At each increment the case project was synced with the rest of the product.

The case project team consisted of around 100 developers spread across three different locations. The developers were divided into different teams which all had one responsible lead developer. A software architect had the ultimate overall responsibility for the software architectural work of the case project. Architectural discussions primarily involved this software architect and the lead developers. The system architecture was defined in Enterprise Architect [4], a tool for managing software architecture models, and continuously maintained by the software architect.

The case project included code at different levels of the software stack with the main parts being written in C++ and JavaScript. The implemented checkers focused on the C++ and JavaScript interfaces.

3.1 Continuous Integration

To be able to work in an agile way the case project used continuous integration. The version control system Git was used together with Gerrit [5], a code collaboration tool which

integrates closely with Git and allows reviewing of each others code. The work flow can be seen in Figure 3.1 and begins with a commit from a developer. The commit is received by Gerrit where it is called a *change*. Gerrit lets the developer add reviewers to the change which then are notified to review the code. Reviewers either approve or reject the submitted code and can also write comments to make it easier to fix any faults. If the change is rejected by any of the reviewers the developer has to make the required modifications and upload a *patch* for the change. This process is repeated until all reviewers are satisfied with the code and the commit can be merged, i.e. the code in the last patch is what is actually merged into the code base.

In addition to these code reviews the case project also utilised a Jenkins [6] server which built and tested newly added code. The Jenkins server had also been integrated with Gerrit and could as such approve or reject added code depending on if it built correctly and passed all tests.

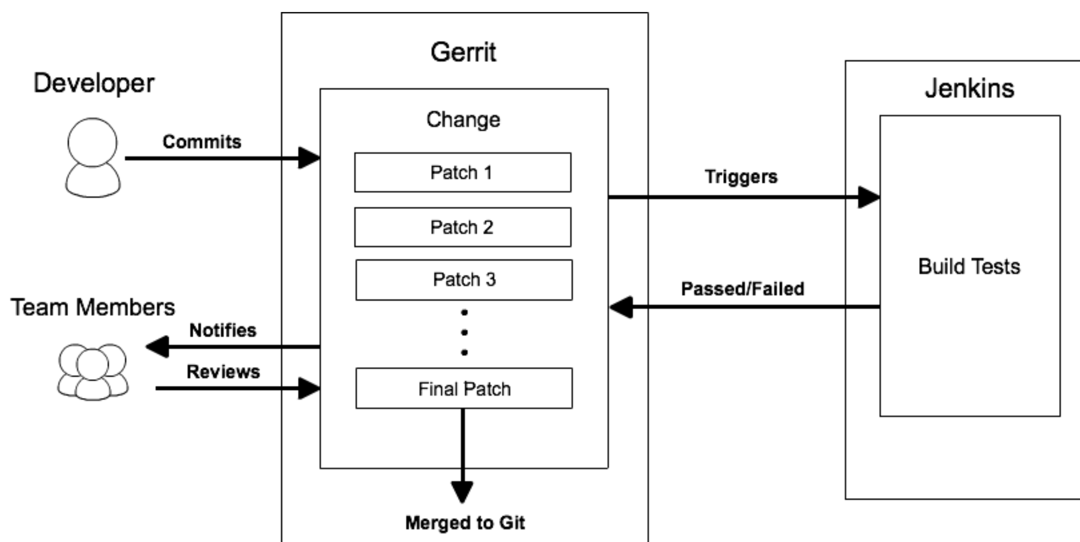

Figure 3.1: The work flow in the project.

3.2 The Architectural Guidelines

In order to make it easier for the developers to follow the intended architecture it had been summarised in a number of guidelines and rules that were communicated to the developers through an online wiki. The developers are supposed to read these guidelines and keep them in mind during their daily work. Due to lack of time and other priorities the developers had not received any proper education on the system architecture and the architectural conformance was therefore dependent on the developers reading and following the guidelines. The team utilised manual reviews to ensure the quality of the code before it was merged with the latest build, and even though some architectural violations could be detected at this stage, it still required the reviewer to know the guidelines.

Since the project was merely one year old at the time of this thesis, the software architecture was still a work in progress and the number of fully defined guidelines were limited. There were approximately 25-30 rules associated with the system architecture, concerning

either JavaScript or C++ components, which could be divided into the following categories depending on the quality requirement that the rule aimed to ensure:

- **Maintainability and Readability**
  - Naming conventions for classes, variables, methods etc., e.g. "Variables shall use camel case format names without spaces".
  - Descriptions for classes, interfaces, methods etc., e.g. "All methods in the interface must contain a description that makes it easy to understand how to use the method".
- **Compatibility**
  - File locations in the project, e.g. "A service component must be located in the service folder".
  - Version management of code components, e.g. "The interfaces must contain a version tag".
- **Portability**
  - Dependencies between different software components, e.g. "The service components should only communicate with other service components using ASF FIDL interfaces".

3.3 The Developers' View of the Architectural Guidelines

To understand how the architectural guidelines were used by the developers in the project, interviews were made with some of them. These are further described in Section 4.2.2. In Table 3.1 the characteristics of the interviewees can be seen.

	Site	Experience (year)	Time in the project (months)	Role
I1	Lund	17	12 (From start)	Developer
I2	Lund	23	12 (From start)	Developer
I3	Lund	5	12 (From start)	Developer
I4	Lund	10	10	Lead Developer
I5	B	10	10	Lead Developer
I6	B	6	7	Developer
I7	B	1	6	Developer
I8	C	10	9	Lead Developer
I9	C	10	12 (From start)	Lead Developer

Table 3.1: Characteristics of the interviewees.

The information gathered in the interviews was summarized and the main findings are listed below:

The developers do not look at the guidelines very often and some do not recall ever reading them at all. The reason for this was, for many of the interviewees, that they had been in the project from the start and had therefore been a part of the architectural design. Many of them felt that they had good knowledge about the architecture and when they encountered something they were uncertain about they asked the architect or lead developers instead of consulting the guidelines. One of the interviewees pointed out that asking in person meant they could get additional background information motivating the architectural rules, something that is not available in the guideline document itself.

The lead developers and the architect discusses the architecture and makes decisions on how to go forward. Almost all communication regarding the architecture is verbal and uncertainties are solved as they appear and changes are then added to the guidelines. Since the developers does not seem to read the guidelines much it could be problematic to communicate such changes to all developers in the different teams, especially since the teams are placed in three different countries. A lot of work and responsibility is also put on the architect, something that will eventually cause problems. The architecture was mainly discussed with colleagues when a new component was to be developed and not so much otherwise.

When new developers join the project they do not get any extensive education about the system architecture more than a small presentation and access to the guidelines. The main approach seems to be that the developers learn as the development proceeds and asks the architect or lead developer for guidance if needed. This is also something that seems to differ a lot between the different offices. Several of the interviewees mentioned using previously written code of good quality as templates when implementing something similar. This way the main ideas of the design and architecture can be followed without needing much knowledge of the rules and guidelines.

All interviewees were very positive towards some kind of automated way of checking the guidelines. They believed that this could make the manual code reviews more efficient since they would have fewer things to correct in the code reviewing process. Some also said that the reviews often does not focus on the architectural point of view but rather on coding rules and the efficiency and logic of the code. Something that also was mentioned was that some of the rules maybe should be checked locally in the IDE instead of later in the tool-chain. This would bring the feedback even closer to the developer, giving them the possibility to correct any mistakes before committing the code.

It is clear that the project relies heavily on verbal communication when it comes to the architecture. This puts a lot of responsibility on the architect and lead developers to make sure communication of all and any changes reaches the developers. As the teams are expanding and new developers are coming into the project this communication becomes increasingly difficult, especially since many of the developers does not seem to read the guideline document continuously. Automatic tests of the architectural guidelines could therefore have merit both as an alternative way of communication and a way to decrease the time developers spend on code reviews. In Table 3.2 a summary from the interviews can be seen.

Has read the guidelines	5/9
Reads the guidelines continuously	2/9
Is satisfied with the current handling of the architectural guidelines	3/9
Is positive to automatic tests of the architectural rules	9/9

Table 3.2: Summary of the interview results.

Chapter 4
Research Method

A design science research approach [25] was applied for the work presented in this thesis. This was a suitable methodology since it is designed for solving problems within the specific problem environment by designing and evaluating a solution using scientific methods. In Figure 4.1 the main activities of the research method can be seen. The larger boxes represents the main parts involved in the work, namely exploring the environment and previous knowledge, in order to identify a solution approach. The contained boxes represent activities carried out for each of these main parts.

First a literature study was done in order to understand the problem domain and what previous work that had been done in the area. The outcome of this literature review is described in Chapter 2. The environment and problem space were then explored through discussions with the software architect in the project and by interviewing the developers. These findings are described in Chapter 3. The combined insight into the knowledge base and the environment was then used to design and implement a fitting solution to the problem. The solution is described in Chapter 5. Finally, the implementation was deployed and evaluated in a project within the problem environment. The results of the evaluation are presented in Chapter 6.

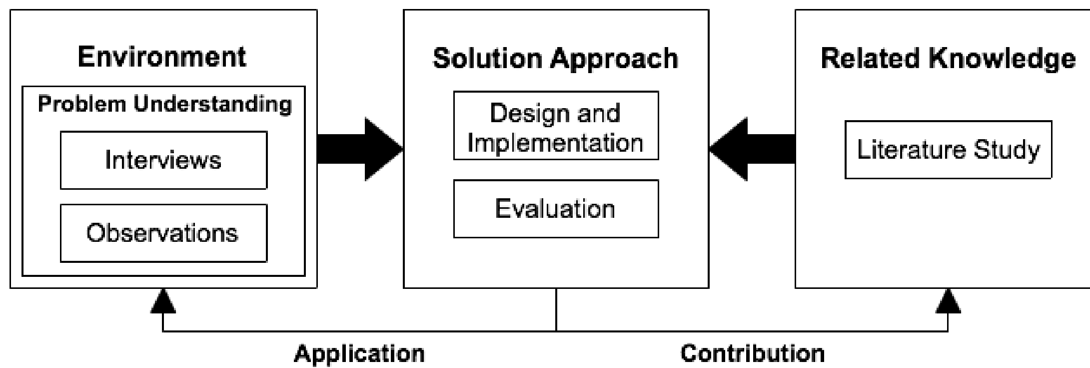

Figure 4.1: The work process of this thesis based on design science [25].

4.1 Exploring Previous Knowledge

The literature study focused on finding previous work that either described the problem area or aimed to solve similar problems. To find relevant scientific material both LUB-search[1] and Google Scholar [2] was used. This allowed the results of the searches to contain articles and scientific papers from a large amount of different databases.

The main keywords and areas that were of interest were the following:

- Software Degeneration
- Software Erosion
- Software Architectural Conformance
- Software Architectural Conformance Checking
- Automatic Tests of Software Architecture
- Tools for Static Analysis

The results found when searching for these keywords were used as a starting point. The first thing carried out in order to identify if an article was relevant or not was scanning the title and abstract. If this indicated that the article may contain relevant information the conclusion was scanned as well before reading the contents more thoroughly. This provided an efficient way to filter out the interesting publications in the large amount of search results. If an article was found to contain relevant information other publications from the same author were investigated. This often led to other persons of interest who may have been collaborating with the author on other papers.

Another way to find interesting articles was to look at the bibliography of papers found to be interesting. This made it possible to find many publications that did not come up during the searches but that still was related to the same area.

[1] http://www.lub.lu.se/soksystem-och-verktyg/lubsearch
[2] https://scholar.google.se

The number of interesting articles found were quite numerous. Many of the authors discussed the problem of software erosion and how it could be prevented. Something that was more difficult to find were publications that compared or evaluated the vast number of different tools available for software architectural conformance checking. Some such articles were found but considering the amount of tools available more evaluations could be needed.

4.2 Understanding the Problem Space

In order to answer RQ2 (How can automatic checking of architectural rules affect the development process?) it was first necessary to understand the development process used in the project. To get this information and to understand the environment in which the architectural checkers were to be implemented, it was required to analyse how the developers worked on a daily basis. It was also necessary to understand the different rules and guidelines defined in the project and how the developers used them. The information was gathered through observations and interviews.

4.2.1 Observations

To be able to make observations of how the people in the project worked with the software architecture and development in general, it was important to do the thesis work at the Bosch offices. This allowed for discussing any questions with the architect or developers making it easier to understand both the problem at hand and a possible solution. Discussions with the architect took place throughout the thesis both in order to understand the guidelines and how they were meant to be used.

Being at the Bosch office also made it possible to access the different tools used by the developers during implementation and observe how these were used. By observing Gerrit and Jenkins an understanding of their function in the project could be obtained. This was important since the implemented checkers would be integrated in the tool chain.

4.2.2 Interviews

In order to understand what the developers knew about the software architecture in the project, and what their attitude towards the guidelines were, interviews were carried out.

The interviews were designed to answer the following questions:

- How are the architectural guidelines used by the developers today?
- What is the general attitude towards the architectural guidelines among the developers?
- What is the developers opinion on automatic checkers?

Sampling of the interviewees was done to get a wide and general view on how the developers worked in the project. In order to get information from all three sites that were working on the project, interviews were held in person with the developers in the Lund

office while developers in other locations were interviewed via Skype. The majority of the interviewees had been working with the project from the start but previous work experience varied greatly, for some it was the first job after graduating and others had been in the field for more than 15 years.

The interviews were designed in a semi-structured way and an interview guide, which can be seen in Appendix A, was used. It was written with questions formulated as openly as possible but while still trying to focus the interview on the subjects of interest. This was done to avoid influencing the interviewee with biased questions. The interview guide was developed iteratively and reviewed by the supervisor at Bosch, Robert Lagerstedt, and the supervisor at LTH, Elizabeth Bjarnason. The top level questions were the ones primarily asked and the sub questions were used as a support to ensure that all the desired information was obtained. The interviewee was encouraged to speak freely about the main subjects and the more specific questions were only asked if it was required. This approach was suitable since it made it possible to explore the subject of interest without knowing the specific questions that needed answers.

For every interview the same work process was used, one acted as an interviewer and the other took notes and asked complementing questions if he felt anything was missed. All of the interviews were recorded which made it possible to listen to them again if needed. The time spent on each interview varied, between 10 to 20 minutes, as some of the participants were more talkative while others were more direct and less inclined to elaborate on their answers. The recorded material was used together with the notes taken during the interview session to summarise the answers of each interviewee. The notes were not full transcriptions of every word but short sentences of what was found to be most important in order to quickly afterwards be able to write more thorough summaries.

To draw conclusions the summaries were analysed. This was done by finding similar or different answers to the same questions or areas discussed during the interviews, a method similar to coding [23] but not as systematic.

The results of these interviews are presented in Section 3.3.

4.3 Design and Implementation

Since the goal of this thesis was to integrate the checkers in the tool-chain and evaluate them it was important that the implementation would be feasible. Therefore some of the rules that were believed to be easier to automatically test were identified together with the architect. The corresponding checkers were then implemented and integrated in the tool-chain. The design was based on the problem understanding and the literature study. Prototypes were designed and developed iteratively and tested both with actual source code from the project and with test cases written to be able to test a larger variety of possible input. When a checker could go through the latest code base providing the expected output, and without errors, it was seen fit to use for further evaluation. When the first checkers had been evaluated the work continued with the checkers considered more difficult to implement using the same work flow. The design and implementation are further described in Section 5.2.

The rules and their meaning were continuously discussed with the architect to ensure that the checkers were working as intended and to avoid false positives and negatives.

Some rules were also added or modified in result of these discussions.

4.4 Evaluation

The evaluation of the checkers was aimed at answering RQ2 (How can automatic checking of architectural rules affect the development process?) and RQ3 (How do developers perceive the existence of automatic checkers of architectural rules in the tool chain?). To further specify what data that was desired from the evaluation RQ2 was divided into the following sub-questions:

RQ2.1 What breaches were found by the checkers that were missed in the manual code reviews?

RQ2.2 Which of the breaches caught in the manual code reviews could instead have been caught automatically by the checkers?

By answering these two questions the potential of the checkers could be assessed and what affect it could have on the development process.

The following sections describes the different parts of the evaluation. To answer RQ2.1 and RQ2.2, measurements of breaches in the project were made, while RQ3 was answered by deploying the checkers in the live project.

4.4.1 Breach Metrics

To measure how many breaches that the checkers found and how this compared to the manual code reviews currently used, the checkers were run on the following code:

1. All past commits before manual code reviews had been applied
2. Patches done on all individual commits after manual code reviews
3. The complete code base at the start of this thesis

$$F_{reviews} = \frac{B_{commit} - B_{patch}}{B_{commit}}$$

Where B_{commit} is the sum of all breaches found in the initial commits and B_{patch} is the number of breaches that still remained after manual code reviews and patching. This could be used as a measurement of the checkers potential and give an indication of how much they could contribute in addition to the manual code reviews.

To be able to run the checkers on the unpatched commits in the project a way to simulate these was required. These commits were only available in Gerrit since the commits in the actual Git repository were only the final patches of each commit. This led to a solution consisting of a bash script that had the following work flow:

1. Get the final commit id:s from the git log up until the start of the master thesis. A date was chosen to make sure developers working in the project had not already been affected by knowing about the thesis. These commit ids only include the final patches of each Gerrit change.

2. Get all the changes from Gerrit.
3. Find the corresponding change reference for each commit id.
4. Modify the last part of the reference to point to the initial commit instead of the last patch.
5. Repeat the following for every commit in chronological order:
    (a) Fetch the initial commit, using the gerrit reference created in (4).
    (b) Run the checkers on every file that was either created or modified in the commit.
    (c) Checkout to the final patch of the commit.
    (d) Run the checkers again in the same way as before.

The breaches found by the checkers were written to separate log files depending on if they were found in an initial commit or in a patch. Breaches that had been found in an earlier commit were not added again. These log files could then be analysed both separately and together to find out what breaches that were fixed in the patches, i.e. caught in the manual code reviews.

The checkers were then run on the code base as it was when the thesis was started to find any breaches present in the system at that moment. This could be used to further motivate the need for automatic tests in order to ensure architectural conformance.

4.4.2 Deployment of the Checkers in the Project

To find out how the checkers functioned in the tool-chain and how their feedback was received by the developers, the checkers were deployed live in the project. During a period of two weeks the activity of the checkers was monitored to get information about which developers that had been in contact with the checkers. If any breaches were committed during this time the developer responsible was contacted to discuss how the feedback of the checker was received and how it was acted upon. This information could then be used to identify problems regarding the functionality of the checkers and the interaction with them. The following main questions were asked to the developers:

- Did you understand the feedback provided by the checkers?
- How did you act after receiving the feedback?

After the checkers had been live for two weeks the commits made during this period were analysed to see what breaches had been found.

Chapter 5
Solution Approach

This chapter describes how the checkers were designed, implemented and integrated in the tool-chain. A classification of the architectural guidelines was first made and is described in Section 5.1 while Section 5.2 describes the implementation.

5.1 Categorisation of Architectural Rules (RQ1)

The architectural guidelines were categorised in order to better understand how different guidelines could be tested. The categories were based on the information required to test the rule and considered two aspects, the number of information sources and the complexity needed to extract this information. This resulted in 4 categories with the following definitions:

Isolated Simple

The source file contains all the required information needed to test the rule and the extraction of the information does not require any extensive understanding of the code.

Example: *"The interfaces must contain a description"*.

Multiple Simple

Further information sources are needed in addition to the source file but there are still no need for deeper understanding of the code.

Example: *"The interfaces must be defined in Enterprise Architect."*

Isolated Complex

The individual file contains sufficient information to test the rule but understanding of how the code works and what it means is also required. For example understanding concepts like variable declarations, function calls and loops.

Example: *"Variables shall use camel case format names without spaces."*

Multiple Complex
Both multiple information sources and deeper understanding of the code is needed to test the rule.
Example: *"Every needed definition should be included directly in a source file or the corresponding header file to avoid indirect dependencies."*

These categories demand different level of complexity in the test implementation and can therefore act as a rough estimate when trying to decide if a rule is easy to automate or not. Category *Isolated Simple* and *Multiple Simple* can be seen as the easiest since regular text parsing often can be sufficient to test the rule. The difficulty to implement rules of category *Multiple Simple*, *Isolated Complex* and *Multiple Complex* is not as easy to rank since the difficulty differs greatly. For category *Isolated Complex* and *Multiple Complex* it depends on how much information is needed of how the code actually works. If deeper knowledge is required perhaps an external tool has to be used, either a more sophisticated parser or even a compiler. This could greatly increase the time needed to implement a test since either knowledge of the used tool is needed or implementation of the specific parser functionality has to be done. For rules that fall into category *Multiple Simple* the implementation complexity depends heavily on the external information, both the quantity of it and how difficult it is to extract the necessary data from it. If the information in the external sources can be extracted using simple parsing the complexity will not increase much compared to rules of category *Isolated Simple*, but if the extraction requires additional separate tools or if there is much external information needed from different sources the complexity increases.

5.2 Solution

The design of the solution needed to match a number of contextual requirements posed by the project where this work was performed. To begin with the use of commercial tools were not an option for this thesis. Most of the tools described in Chapter 2 were therefore discarded at an early stage. Furthermore, when inspecting the tools it was also found that most are designed for managing Java source code and also that they require a lot of time to analyse large projects. In the project at Bosch the code that was to be analysed was written in either C++, JavaScript or Franca IDL (FIDL), which limited the number of already implemented tools that could be used. The time it took for the tools to analyse the code was also important in the project since building and testing already consumed a lot of valuable time. Lastly it was desired that the checkers should be closely integrated in the already existing tool-chain. Because of these requirements the implementation was limited to free and lightweight tools.

5.2.1 The Checker Implementation

Since the checkers were to be integrated in the existing tool-chain, described in Section 3.1, it was decided early on that they should run on the Jenkins server which already was

running build tests, see Figure 5.1. The feedback could then be given to the developers through Gerrit. The team already used a plugin in Jenkins which allowed build tests to automatically run on specific Gerrit events. Just as these build tests, the architecture checkers should fire when a new commit was made to Gerrit, thus making it possible for the developers to patch any breaches in the commit before merging the code. The feedback was sent from Jenkins to Gerrit by writing it to a text file and specifying that the content of that file should be sent to Gerrit if the Jenkins job failed.

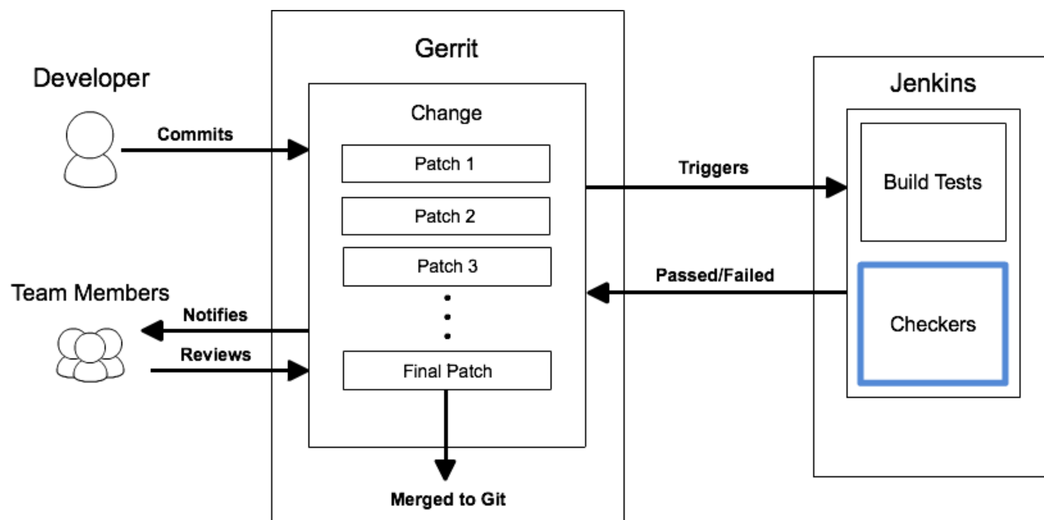

Figure 5.1: The checkers location in the tool-chain.

In Figure 5.2 an example of the feedback given to the developer in Gerrit can be seen. The information includes which file and which line in the file the breach was found in, as well as information of what type of breach that was identified. Using this information the developer could then fix the issues and create a patch for the commit.

The different parts of the checker implementation can be seen in Figure 5.3 and the work flow is described below.

1. The *Start Script* is a bash script that runs whenever a commit or patch is made in Gerrit. The script first looks in the *Config File* to determine if any checkers should be run for the affected Git repo. If that is not the case the script exits.

2. The Start Script generates a list of files modified in the commit which then is sent to the *Python Script* as an argument.

3. The Python Script looks in the Config File to see which individual checkers should be run on each file and if any of the files should be ignored.

4. The Python Script instantiates the necessary *Checkers* using the *Checker Factory*.

5. The Python Script then runs each of the instantiated Checkers on the files and sends an array which is to be filled with potential breach messages.

6. When each checker is done it returns a status to the Python Script, 1 if breaches were detected and 0 otherwise.

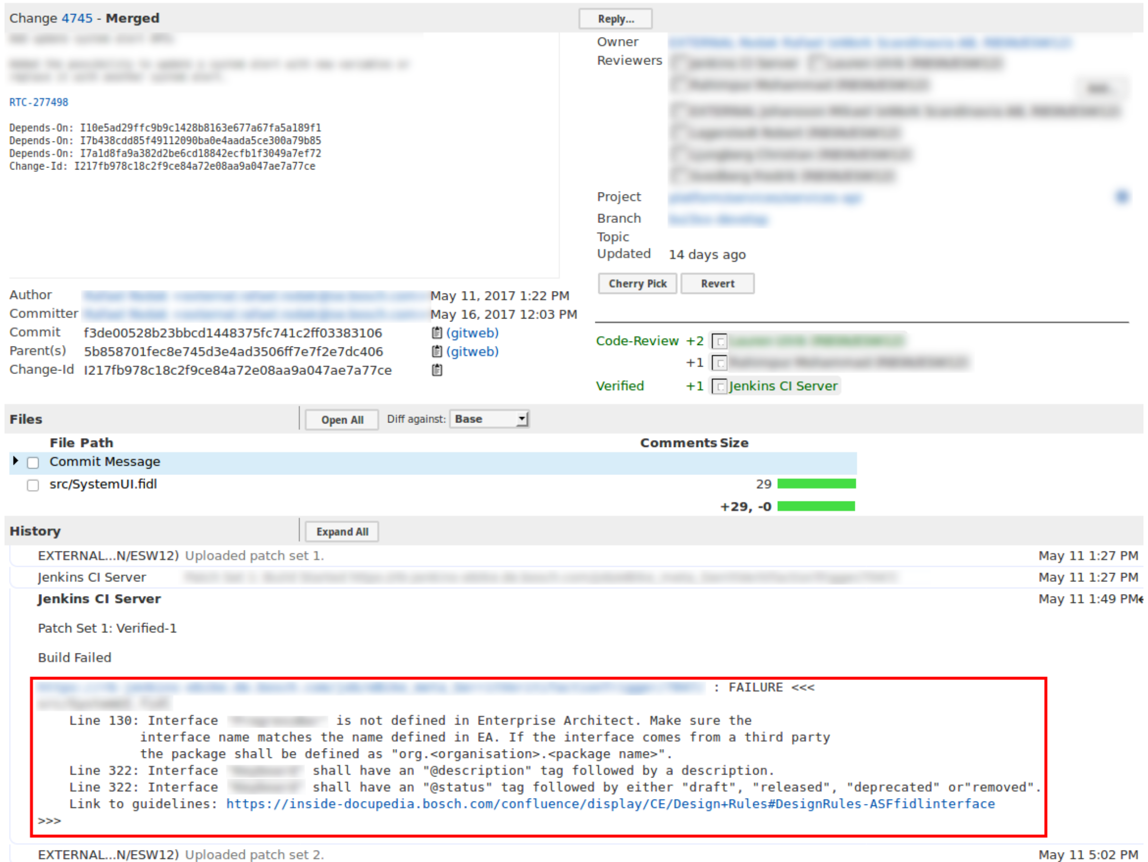

Figure 5.2: An example of the feedback from the checkers presented in Gerrit. The text confined in the red rectangle is the feedback generated by the checkers.

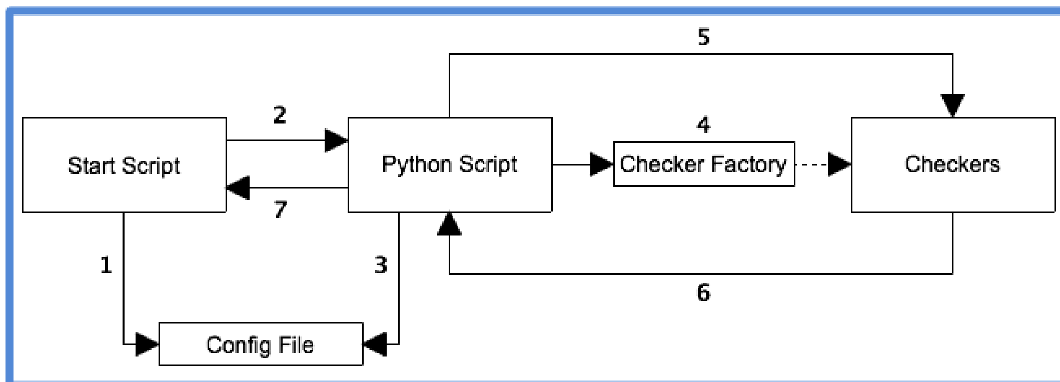

Figure 5.3: The design of the checker solution.

7. The Python Script uses the statuses received from each of the checkers to determine the overall status of the architectural checkers and returns this to the Start Script. All the breach messages that were collected from the checkers are written to a text file.

The motivation for this design was that it should be easy to add new checkers in the future. This was achieved by using the Checker Factory and Config File in the implementation and specifying a standardised way for the checkers to return their feedback. The

division of the design between Python and Bash was made since the necessary calls to Git and Gerrit was most easily done in Bash while parsing and actual analysis of the code files was found to be easier using Python scripts. During the thesis three separate checkers were implemented: FidlInterfaceChecker (FIC), JsInterfaceChecker (JIC) and FidlTagChecker (FTC). These checkers tested various rules further described in the next section.

5.2.2 The Rules Tested by the Checkers

The major part of the implementation focused on rules concerning the FIDL and JavaScript interfaces in the project. This was mainly due to the fact that these rules were found to be the ones primarily related to the system architecture when discussing them with the architect. The concerned files were also quite small and testing the rules did not require any complex parsing. Using the categories defined in section 4.3 these rules were either category *Isolated Simple* or *Multiple Simple*. Because of this it was decided to parse these files using libraries in Python for text parsing combined with regular expressions, instead of using more refined tools.

In the checkers of category *Multiple Simple* it was necessary to compare the architecture defined in Enterprise Architect, the tool used for architecture modelling in the project, with the information gathered from the source code. In order to achieve this the architecture was extracted from Enterprise Architect in an XML file. Ideally this file would be regenerated automatically at least once a day to ensure the checkers had up to date information of the system. This was however not implemented during the thesis and had to be done manually by the architect. The XML file could then be parsed using the ElementTree library in Python, which allows fast and easy parsing of XML files, to gather the required information of the system.

The following rules regarding FIDL and JavaScript files were implemented in checkers that were put live in the project:

1. ASF FIDL interfaces
   (a) The interfaces governed by Bosch must be in the namespace com.bosch
   (b) The interfaces governed by other companies must be kept in the original namespace, like *org.companyname.interfacename*
   (c) The interfaces must contain a version tag
   (d) The interfaces must be defined in Enterprise Architect
   (e) All methods in the interface must contain a description that makes it easy to understand how to use the method
   (f) The interfaces must have a status tag
   (g) A deprecated API must use the *deprecated* tag as well as setting the status to deprecated, and also a description giving the developer alternative
2. Javascript interfaces
   (a) The interfaces must contain a version tag
   (b) The interfaces must have a module tag

(c) The interfaces must have a status tag

(d) The interfaces must be defined in Enterprise Architect

In Table 5.1 information on what category the different rules were considered to be in and which checker they were implemented in can be found.

	Implemented in checker	Category	Fully checked
1a	FIC	*Isolated Simple*	Yes
1b	FIC	*Isolated Simple*	Yes
1c	FTC	*Isolated Simple*	Yes
1d	FIC	*Multiple Simple*	Yes
1e	FTC	*Isolated Simple* (*Isolated Complex*)	No
1f	FTC	*Isolated Simple*	Yes
1g	FTC	*Isolated Simple*	Yes
2a	JIC	*Isolated Simple*	Yes
2b	JIC	*Isolated Simple*	Yes
2c	JIC	*Isolated Simple*	Yes
2d	JIC	*Multiple Simple*	Yes

Table 5.1: Information on the different guidelines.

The rules 1.a-1.c and 2.a-2.c could all be tested using ordinary text parsing and regular expressions since no further information was needed than the file itself. The rules 1.d and 2.d required information about the package structure of the system and in which package each interface should be. This was obtained from the Enterprise Architect XML by the use of ElementTree.

Some of the checkers did not test the exact rules formulated in the guideline document directly. For example rules 1.a, 1.b and 1.d where all checked using the same checker, FIC. This checker looked at if an interface existed in the Enterprise Architect model and also if it was placed in the correct package. This meant that an interface governed by Bosch could have the correct namespace: *com.bosch*, and still be seen as a breach if it was not in the correct package or if the interface was not found in Enterprise Architect. However the checker would output different messages depending on what type of breach was identified in order to make it possible for the developers to understand which guideline that had not been followed.

Some of the rules were more difficult to test, such as rule 1.e. In order to automatically test if a text is descriptive enough and easy to understand one would need to use natural language processing to some extent, something that would place these rules in the *Isolated Complex* category, and make the rule very difficult to test automatically. To circumvent this problem the checkers did not actually look at the content of the description. The checker thus simply checked if there was a description present for each method and that it was written in the correct format, placing the rule in the *Isolated Simple* category. This meant the checker did not truly test these defined architectural rules, only a part of them.

Something that was learned from the implementation of the checkers was that it is sometimes easy to underestimate the complexity of the parsing that has to be done to extract the correct information when checking a file. The regular expressions used became

more complicated than expected and several measures had to be taken to be able to handle all special cases. This may motivate the use of more refined parsers even for simpler rules to avoid the extra work.

5.2.3 Additional C++ Checker

In addition to the checkers described in the previous sections some experimentation was also done with rules for C++. This was the main focus at the start but during the thesis it was found that the guidelines did not include many rules about the C++ code and the ones that existed were mostly not directly related to the system architecture. However, it was still decided that at least one checker should be implemented in order to evaluate how rules that required more complex analysis could be tested, i.e. rules of category *Isolated Complex* and *Multiple Complex*.

The checker that was implemented tested a rule that stated that all modules used should always be included directly, i.e. include what you use. This means that for every definition used in a file there has to be an include directive in either the file itself or its corresponding header file. The rule was of category *Multiple Complex* and in order to check it automatically it was needed to parse the targeted source code with a deeper understanding of the semantics of the code. For C++ source code the best alternative that was found for the use case was Clang, which is a front end for the C and C++ compiler LLVM [9]. Clang can generate an abstract syntax tree (AST), which is a representation of the source code in form of a traversable tree structure consisting of nodes. These nodes represent different parts of the source code and contain information such as what type of code the node represents and its location in the source file. In Figure 5.4 a piece of source code can be seen together with the AST generated from it.

To work with Clang there is a C-API called libclang used to traverse, transform and analyse the different nodes. This API can be used directly by writing code in C but there are also Python bindings available to make the interaction more versatile.

To check the rule the source file is parsed with libclang, finding all uses of structs, classes, enumerations etc., and in what file they are defined. Then all the include directives in the file and in its header file are extracted and compared with the definition paths. If any definition is not included in these directives a warning is printed.

This checker was developed separately as a single python script and was not integrated in the design described in Section 5.2.1. It was however evaluated separately and the results are discussed in Section 6.1.1.

```
class A{
private:
  int m_id;
public:
  int getId(){
    return m_id;
  }
};

int main(){
    int sum;
    for(int i = 0; i<10; i++){
      sum += i;
    }
    return sum;
}
```

```
|-CXXRecordDecl 0xb63cfd0 <ast_demo.cpp:1:1, line:8:1> line:1:7 class A definition
| |-CXXRecordDecl 0xb63d0f0 <col:1, col:7> col:7 implicit class A
| |-AccessSpecDecl 0xb63d188 <line:2:1, col:8> col:1 private
| |-FieldDecl 0xb63d1c8 <line:3:3, col:7> col:7 referenced m_id 'int'
| |-AccessSpecDecl 0xb63d210 <line:4:1, col:7> col:1 public
| `-CXXMethodDecl 0xb63d298 <line:5:3, line:7:3> line:5:7 getId 'int (void)'
|   `-CompoundStmt 0xb63d3f8 <col:14, line:7:3>
|     `-ReturnStmt 0xb63d3e0 <line:6:5, col:12>
|       `-ImplicitCastExpr 0xb63d3c8 <col:12> 'int' <LValueToRValue>
|         `-MemberExpr 0xb63d390 <col:12> 'int' lvalue ->m_id 0xb63d1c8
|           `-CXXThisExpr 0xb63d378 <col:12> 'class A *' this
`-FunctionDecl 0xb63d440 <line:10:1, line:16:1> line:10:5 main 'int (void)'
  `-CompoundStmt 0xb63d8a8 <col:11, line:16:1>
    |-DeclStmt 0xb63d5c0 <line:11:5, col:12>
    | `-VarDecl 0xb63d560 <col:5, col:9> col:9 used sum 'int'
    |-ForStmt 0xb63d818 <line:12:5, line:14:5>
    | |-DeclStmt 0xb63d670 <line:12:9, col:18>
    | | `-VarDecl 0xb63d5f0 <col:9, col:17> col:13 used i 'int' cinit
    | |   `-IntegerLiteral 0xb63d650 <col:17> 'int' 0
    | |-<<<NULL>>>
    | |-BinaryOperator 0xb63d6e8 <col:20, col:22> '_Bool' '<'
    | | |-ImplicitCastExpr 0xb63d6d0 <col:20> 'int' <LValueToRValue>
    | | | `-DeclRefExpr 0xb63d688 <col:20> 'int' lvalue Var 0xb63d5f0 'i' 'int'
    | | `-IntegerLiteral 0xb63d6b0 <col:22> 'int' 10
    | |-UnaryOperator 0xb63d738 <col:26, col:27> 'int' postfix '++'
    | | `-DeclRefExpr 0xb63d710 <col:26> 'int' lvalue Var 0xb63d5f0 'i' 'int'
    | `-CompoundStmt 0xb63d7f8 <col:30, line:14:5>
    |   `-CompoundAssignOperator 0xb63d7c0 <line:13:7, col:14> 'int' lvalue '+=' ComputeLHSTy='int' ComputeResultTy='int'
    |     |-DeclRefExpr 0xb63d758 <col:7> 'int' lvalue Var 0xb63d560 'sum' 'int'
    |     `-ImplicitCastExpr 0xb63d7a8 <col:14> 'int' <LValueToRValue>
    |       `-DeclRefExpr 0xb63d780 <col:14> 'int' lvalue Var 0xb63d5f0 'i' 'int'
    `-ReturnStmt 0xb63d890 <line:15:5, col:12>
      `-ImplicitCastExpr 0xb63d878 <col:12> 'int' <LValueToRValue>
        `-DeclRefExpr 0xb63d850 <col:12> 'int' lvalue Var 0xb63d560 'sum' 'int'
```

Figure 5.4: Example source code with the corresponding AST generated by Clang.

Chapter 6
Evaluation Results

The checkers described in Chapter 5 where evaluated by applying them to the code base of the case project and deploying them live in the tool-chain.

6.1 Breaches of Architectural Rules (RQ2)

The effect of applying the implemented checkers was evaluated with the metrics described in Section 4.4.1. In order to set the breach metrics in perspective a measurement of the number of potential breaches was required. Since the JavaScript and FIDL checkers only concerned individual interfaces, each rule could at most produce one breach per interface with the exception of the method description tags which potentially could report one breach for every method in the interface. This made it easy to obtain a measure of the number of potential breaches throughout the project by counting the interfaces and their methods present in the code at different times. In Figure 6.1 the number of potential breaches together with the number of found breaches throughout the project can be seen. This shows that approximately 10-20% of the potential breaches were actual breaches at any given time of the project.

In Figure 6.2 the result of running the checkers on all past commits can be seen. It shows the different types of breaches and how many of them that were fixed in the manual code reviews before the commits were merged. These results could then be used to calculate the value of $F_{reviews}$:

$$F_{reviews} = \frac{B_{commit} - B_{patch}}{B_{commit}} = \frac{178 - 122}{178} = 0.315$$

This means that approximately 32% of all the breaches found by the checkers were discovered and fixed in the manual code reviews. The result shows that there were a lot of

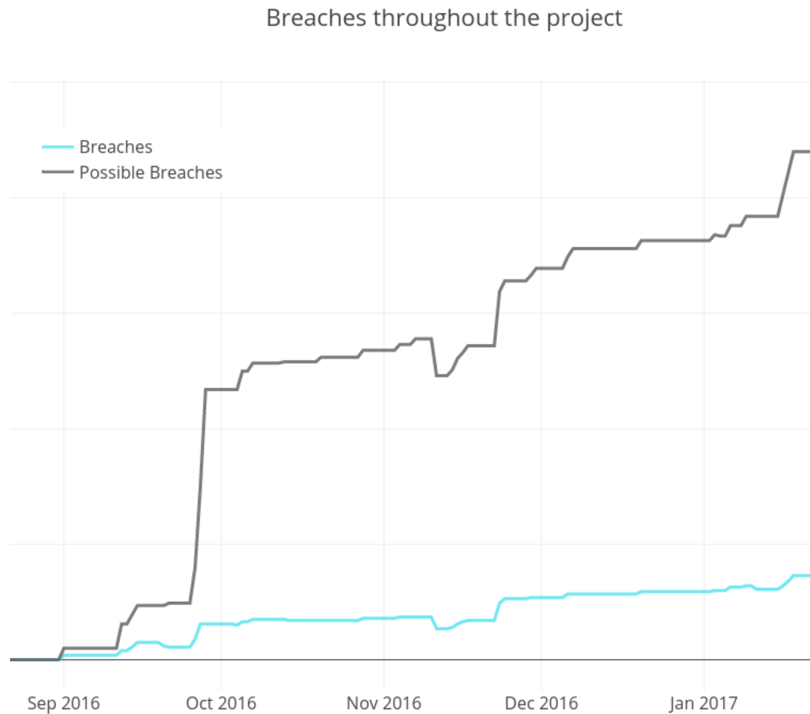

Figure 6.1: The number of potential breaches and the actual breaches found throughout the project.

breaches that slipped through the manual code review process and were therefore merged into the code base.

In Figure 6.3 the breaches found in the project at the start of the thesis can be seen. The total sum of these breaches could be compared with B_{patch} to see how many of the merged breaches that had been fixed in later commits. The result of this was 45%. As can be seen in the figure there were no breaches of the types "js-module tag" or "fidl-method tag", the latter not being so surprising since a lot of those breaches were fixed in patches already.

Since two thirds of the breaches were not found this can motivate that testing these rules using automatic checkers could decrease the number of breaches in the code base while also reducing the effort needed during the manual code reviews.

The results also showed that even though breaches may not be found during the manual code reviews a large percentage of them are found later during the project and fixed in separate commits. However this does not seem to reduce the percentage of breaches present in the merged code base at any given time as seen in figure 6.1 which indicates new breaches are introduced at almost the same rate as previous breaches are fixed. Every breach that is merged results in more work for the architect since the code needs to be monitored continuously. If all the breaches could be detected by the checkers as soon as they are introduced this would save the architect both time and effort in addition to ensuring that the code base is free from breaches.

A C++ checker for a rule of category *Multiple Complex* was also implemented. This checker required the code to be built in order to resolve all the include paths. Since this was very time consuming there was no time to run this checker on every past commit in

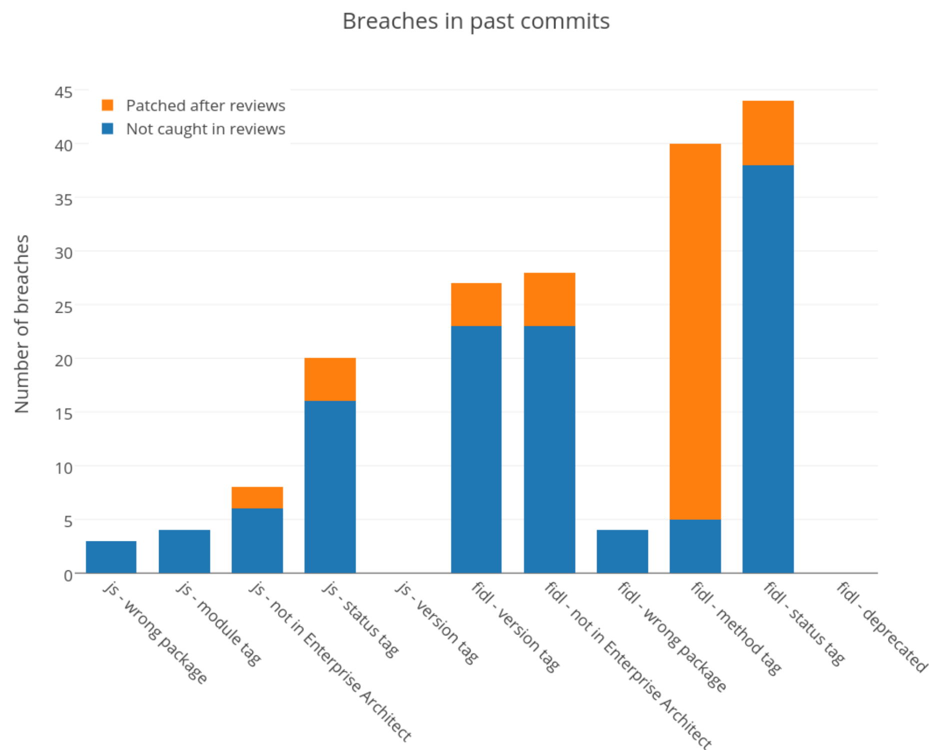

Figure 6.2: The breaches found when running the checkers on all commits up to 2017-02-10.

the project like the others and it was therefore only tested on the current code base. This checker was run on 367 code files and 88 breaches were detected. Because of the small amount of breaches they could all be manually verified as correct, there could however be false negatives due to the complex build system.

This checker was not put live in the project for a number of reasons. To start with it would have needed the Clang compiler to be installed on all of the Jenkins servers, something that was decided to be too much overhead work for this checker only. Another reason for not putting it live was, as mentioned, that the rule was not considered as close to the architecture as the rules concerning the FIDL and JavasScript interfaces. Another issue that was discovered was the complexity of the build system used in the project and how include paths were managed. There was a lot of preparatory work being done before all the include directives could be resolved by the compiler and the code could be built. Since the checker ran its own compiler it would need to be inserted into the current build chain, something that was not an option since the checkers should run separately. This made it difficult to generalise and automate the checker to the extent that it could be used in the tool-chain. It was however possible to partly solve the problem manually in order to test the checker on the current code base.

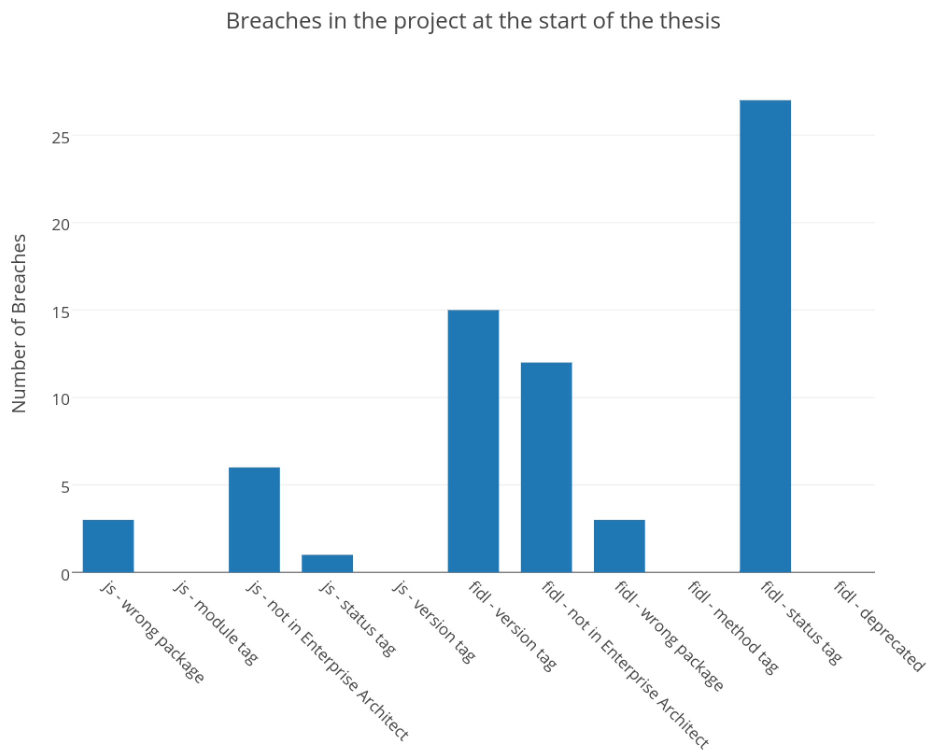

Figure 6.3: The breaches present in the code at the time of 2017-02-10.

6.2 Deployment in the Project (RQ3)

In order to test the checkers in the actual tool-chain they were put live in the project for a period of 2 weeks. During this time they checked every commit and patch that were made to the affected repositories which in total was 106 with an average of 10 per day.

In these commits and patches 25 breaches were found in total, 10 of which were newly introduced breaches. The other 15 were breaches previously identified when the checkers had been run on legacy code, but that had not been fixed prior to the checkers being put live. This seemed to cause a bit of confusion since a few developers remarked that the checkers were blocking their commits and referring to breaches in parts of the code they had not written. This will however not be a problem as soon as all the legacy breaches are fixed.

8 out of the 10 new breaches were cases where a developer committed an interface with a name that was not defined in Enterprise Architect. When most of these breaches occurred the architect had not added any corresponding interface in EA at all, in other words the result of such a commit could not become anything but a breach. This could be seen more as an issue in the process of how new interfaces are defined than developers causing breaches. To avoid these breaches any interfaces prepared to be developed should be discussed and defined beforehand by the architect.

The goal of this part of the evaluation was first and foremost to get feedback on the checkers from the developers. During the two week period 6 different developers came in contact with the checkers and 4 of them answered the questions in Section 4.4.2. All of

them found the feedback from the checkers to be easy to understand and they knew what they had to do right away. They all got at least one error concerning Enterprise Architect which required them to contact the architect of the project, something that they all did as soon as they saw the checker feedback. This was something that was a bit annoying for the developers since they had to wait until the EA model was updated before they could merge their commit. One potential problem that was discovered was that when the checkers reported several errors with the first being the EA breach, one developer did not read the rest of the breaches that could be fixed immediately, without the need to contact the architect. This shows that the way the feedback is presented could be given some additional thought.

Chapter 7
Discussion

In this chapter we discuss the different results obtained presented in Chapter 6 from the perspective of the research questions in Section 1.2.

7.1 Categorisation of Architectural Rules

To answer the first research question,"How can architectural rules be categorised according to suitability for automatic checking?", the categorisation of the architectural guidelines suggested in Section 5.1 was done. The categories were based on two factors, the number of information sources required for checking the rule and the needed complexity for retrieving the information. These categories can be used as a framework when analysing architectural guidelines from the perspective of automatic checking, mainly when deciding the complexity of automatically testing a specific rule.

The advantage of our categorisation is that it is easy to make a fast analysis of the architectural rules that you want to automatically test and get a first notion of the required complexity. The defined categorisation is however a bit shallow and not enough to definitively decide the implementation complexity for a rule. This means that a rule which conforms to category *Isolated Simple* can in some cases be much more time consuming to implement than a rule of category *Multiple Complex*. This greatly depends on which approach that is chosen for the implementation and which tools that are used. To get a more precise categorisation more factors therefore need to be considered when defining the categories. This type of categorisation can be done in many ways and our categories should not be seen as a definite, unambiguous model but rather as a guidance. The purpose of the categories is to make it easier to discuss and reason about the rules.

We have not come across this type of categorisation of architectural rules in any previous work studied. When discussing different types of architectural rules it is often focused on what the rules concerns, rather than what is required to test it.

7.2 Affect on the Development Process

To answer RQ2, "How can automatic checking of architectural rules affect the development process?", the evaluation described in Section 6.1 was done. The breach metrics shows how the automatic checkers could affect the development process in a positive way. It can detect many breaches that are currently missed in the manual code reviews and in that way aid the architect and the developers in the architectural conformance checking of the code. At the same time it can save some of the time that is currently being spent on manual reviews and fixing breaches that are later discovered in the code.

In the case project the checkers did not lead to any great difference in the work flow of the developers and could therefore be adopted quickly. One reason for this was that the checkers were so tightly integrated into the existing tool-chain. We think that this is a crucial aspect of the solution since it avoids adding any further step to the developers work flow, something that could counteract the desired efficiency impact on the development process. However, to fully implement the use of automatic checking into the project the development of the checkers themselves needs to be organised and planned for in the project. This would be a greater change in the development process of the project than the actual use of the checkers and is something that should be further studied.

When the checker solution is used to its full potential, i.e. when most of the architectural rules are implemented as checkers continuously in the project, this could fundamentally change the way the rules are communicated from the architect to the developers. It would then be possible to make a checker for a rule instead of putting it in a guideline document and in that way communicate it to the developers and at the same time ensuring that it is followed in the code. This is the main purpose of our solution approach.

7.3 Reaction from the Developers

The third research question, "How do developers perceive the existence of automatic checkers of architectural rules in the tool chain?", was investigated by the live evaluation described in Section 6.2. The interviews that were done with the developers beforehand showed that they were generally positive to the use of automatic checking and this was also the case during the evaluation. It showed that the developers appreciated the automatic checkers and understood the feedback that was provided. It therefore seems that the checkers can quickly become a natural part of the tool-chain and development process in the project. It also became clear that the guideline document, used for communicating the architectural rules in the case project, was not properly used by the developers which further motivates the need for the automatic checkers.

The only concern expressed about the automatic checkers was about potential false positives. Since the checkers stops commits from being merged into the code base this would be disrupting and slow down the development process. Much time was therefore spent on eliminating any false positives before the checkers were put live in the project.

It would have been desired to more thoroughly examine how different feedback was received by the developers in a more comparative manor. If more rules of different types had been implemented and with different types of feedback, RQ3 could be more extensively answered. The live evaluation was also done during a rather short period of time and not

that many developers came in contact with the checkers feedback. This was however not the main focus of the thesis and the time was not sufficient.

7.4 Validity

This section discusses the validity of the different results obtained during the thesis. Both the results gathered during the evaluation of the automatic checkers and during the interviews are discussed in the following sections.

7.4.1 Evaluation Data

Internal Validity

The *Internal validity* concerns the causal analysis that is made by the researcher when investigating how different factors affects each other. If there are factors that the researcher is not aware of this could lead to false causal conclusions [34]. In order to increase the internal validity of the study several measures were taken.

The checkers were tested thoroughly before using them in any extraction of data and efforts were made to eliminate false positives or negatives. The breaches found in the code were to a large extent manually verified against the breaching code and if false positives were detected the checkers were modified and the evaluation was redone. This was an iterative process that lasted until no false positives could be found. False negatives were also minimized by testing the checkers on many different parts of code and a lot of special cases. It was however more important to eliminate false positives in this case since the checkers were to be used live in the project and would stop code from being merged if any breaches were detected.

One issue with the JavaScript interface checkers was detecting an interface declaration from the start since it is not a feature provided by the language. The regular expression used for finding the interfaces was therefore very specific and could only be used on the JavaScript interface files included in the specific project. Otherwise there was a risk that the checker would detect interface declarations that were not meant to be interfaces. To detect an interface it used the package names defined in EA so if an interface declaration had a package name that did not exist at all the checker would ignore it, causing false negatives. This was far from an optimal solution but the assessment was made that is was better to find some of the JavaScript breaches than none at all.

The value on $F_{Reviews}$ may not be representative since most of the patched breaches, almost all of the FIDL-method tag breaches, were fixed in a single commit. Without this commit the value of $F_{Reviews}$ would be significantly lower. This however only strengthens the conclusion that automatic tests could help minimise the number of breaches merged into the project code base.

External Validity

The *external validity* [34] concerns to which extent the evaluation findings can be generalized beyond the case project. This is affected by the fact that the implemented rules

concerned such specific areas of the project. How the JavaScript and FIDL interfaces are used in the project may be something that is specific for this single project. The concept of architectural guidelines of the similar kind is however something that is used in many projects and if the communication and usage of them are the same, these results could probably be applied to those projects as well.

The results show that manual reviews are not sufficient to find all breaches and this is most certainly not limited to this specific case project. The case project was quite young and not that many rules had been defined and you could imagine that this problem would be even greater in larger and more complex software projects.

Construct Validity

Construct validity regards if the tests or measurements actually tests what is claimed [34]. One thing that affected the construct validity was how individual breaches in the code were identified. When running the checkers on past commits it was desired to only log new, unique breaches, but this proved to be difficult. One thing that made this difficult was file name changes or file moves. Since there is no other way to identify a file in Git than the file name, it was used as part of the identification of all breaches found in that file. If a file name changed, even without any of the code changing, all the breaches in that file would therefore be counted again. To avoid this these breaches were identified and removed from the final dataset.

Another issue encountered during the evaluation was the version of the Enterprise Architect XML representation used by the checkers. Since the only available model of the architecture was the current one this had to be used in the evaluation, even if it would have been better to use a snapshot of the architecture from the same time as the commit currently checked. This was discussed with the architect and the conclusion was made that he mostly had made additions and not so many modifications to the already existing architecture during the project. This meant that a breach of the current architecture committed at an earlier stage of the project in most cases also would have been a breach to the defined architecture at the time. A joint decision was therefore made to still use the current version of the architecture when running the checkers on old commits.

7.4.2 Interviews

Measures were taken during the interviews to reduce the risk of biased and incorrect results. By using an interview guide the results of the interviews were more comparable. There were always two interviewers present during the interviews and the data analysis where made in pair which reduced the risk of research bias [34]. Furthermore to ensure that the interviewees could speak their minds it was made sure that they were aware of the fact that the software architect would not get access to any specific answers from the interviews. Also in order to verify that the conclusions made were consistent with what the interviewee was meaning to convey, the summaries were sent out to the corresponding participants by email, giving them the chance to correct any misinterpretations.

Chapter 8
Conclusions

In this master thesis automatic tests of architectural rules have been implemented and evaluated in a software project at Robert Bosch AB. The implementation resulted in checkers that were placed in the tool-chain of the project to be run whenever the developers committed new code. These checkers could then hinder architecturally incorrect code from being checked-in and give the developers feedback in the same tool that was used for writing and receiving manual code reviews. The implemented checkers tested a subset of the different architectural rules and guidelines meant to be enforced during the project. The rules that were tested concerned interfaces for either C++ or JavaScript code.

The solution was evaluated applying the checkers in the project in two different ways. First a simulation was made of all previous commits and then the checkers were put live in the project. This made it possible to both extract historic breach data and test the checkers in the live tool-chain and see how the developers reacted to the automatic checkers.

8.1 Conclusion

This thesis has shown that automatic architectural conformance checking can be a more efficient alternative to manual code reviews for enforcing a system architecture. The evaluation showed that the manual code reviews only catch around a third of the breaches that where detected by the checkers. We have demonstrated how such automatic checking can be implemented in a specific software project and integrated in an existing tool-chain. The implemented approach was introduced in the production environment and was appreciated by the software architect and by the developers. Even though only a small number of simple checkers were implemented it provides a proof of concept of how this approach can be used in the project.

This thesis has also defined a set of categories: Isolated Simple, Multiple Simple, Isolated Complex and Multiple Complex. These categories can be used when looking at architectural rules from the perspective of automatic conformance checking. We have

demonstrated that these categories can act as a framework for analysing and selecting architectural rules of implementation as automatic checkers.

We found that many developers did not consult the architectural guidelines during development, something that would most likely be increasingly problematic as the project and the software architecture grows. The automatic checkers could therefore be of help to ensure the architectural rules are followed without demanding full knowledge of all guideline documentation from the developers. The hope is also that being exposed to the feedback of the checkers will educate the developers giving them a greater knowledge of the defined guidelines. The automatic checkers can in this way also support communication of changes to the architectural guidelines.

The implementation of any automatic architectural conformance checking is often only a cost in the short term, but can provide value as the project begins to evolve and the risk of architectural erosion increases. The breaches found at an early stage of the project could be fixed manually without any extensive effort but as the project grows this will not be the case, especially if the software architect has to do most of the work. The checkers will at that stage be of great use for ensuring architectural conformance. Adding new checkers will also be less costly when the framework and the initial checkers are in place which also adds to the increasing value. If the checkers are used for both communicating and enforcing the architectural guidelines, this saves the architect a lot of time and effort. It would then not be necessary to do the same large amount of manual reviews and if all rules can be implemented in checkers guideline documents will not be as essential to write.

8.2 Future Work

Since this thesis was done in a medium sized and young software project it would be interesting to see a similar study at a larger scale in a more mature project with a complex architecture and greater number of guidelines. Would light-weight checkers as the ones implemented here be of any use in that case or does it demand the use of a complete tool? It would also be welcomed to study the use of automatic checkers over time to see if it is something that is feasible to maintain as the guidelines changes and further checkers has to be implemented. One possible approach could be to find two similar projects and study them in parallel from the start, implementing automatic checking in one of them but let the other rely on manual code reviews.

It would also be interesting to put more effort into evaluating libclang as a mean of architectural conformance checking. It is clear that this is a powerful tool for static analysis of C++ code but exactly what can be done in terms of testing architectural guidelines is mostly undocumented. CppDepend clearly uses it for this purpose but an extensive evaluation of libclang itself would be welcomed.

One large area of architecture conformance are ADLs and their use for defining architectural rules. It would be welcomed to see more research about how such ADLs could be used to automatically generate checkers from defined rules. This could be used to directly link the rule definition with a checker implementation and thus save a lot of effort and implementation cost.

As mentioned in this thesis there are a vast amount of tools for automatic architectural conformance checking available today. There has however been few studies made that

evaluates and compares these tools for specific use cases. This is something that is needed in order for the industry to be able to make well-informed decisions regarding the use of such tools in production, which in turn could led to more companies automating their architectural conformance checking.

Bibliography

[1] Bosch. http://www.bosch.se/sv/se/om_oss/affarsomr_den_och_divisioner/aff_rsomraden-och-divisioner.html. Description of Boschs business areas in Sweden. Accessed: May 16 2017.

[2] clang. https://clang.llvm.org/. The clang project, a C and C++ compiler. Accessed: Mars 08 2017.

[3] CppDepend. http://www.cppdepend.com/. A tool for static analysis of C and C++ code. Accessed: Mars 08 2017.

[4] Enterprise Architect. http://www.sparxsystems.com/products/ea/. Description of the tool Enterprise Architect. Accessed: May 8 2017.

[5] Gerrit. https://www.gerritcodereview.com/. The code review tool Gerrit. Accessed: Mars 24 2017.

[6] Jenkins. https://jenkins.io/. The automation tool Jenkins. Accessed: May 15 2017.

[7] Jittac. http://actool.sourceforge.net. Tool for architecture conformance. Accessed: February 1 2017.

[8] Linq. https://msdn.microsoft.com/en-us/library/mt693024.aspx. Description of Language-Integrated Query. Accessed: April 25 2017.

[9] LLVM. http://www.llvm.org/. The LLVM project. Accessed: April 25 2017.

[10] SAFe. https://www.scaledagileframework.com/. A description of the SAFe framework. Accessed: June 11 2017.

[11] Semmle. https://semmle.com/. A tool for software analytics. Accessed: Mars 08 2017.

[12] Sonargraph. https://www.hello2morrow.com/products/sonargraph. Description of the static analysis tool Sonargraph. Accessed: February 1 2017.

[13] Sonarlint. http://www.sonarlint.org/eclipse/index.html. Eclipse plugin for static analysis, compatible with Sonarqube. Accessed: Mars 08 2017.

[14] Sonarqube. https://www.sonarqube.org/. Homepage of SonarQube, a tool for static analysis. Accessed: Mars 08 2017.

[15] Andrea Caracciolo, Mircea Lungu, Oskar Truffer, Kirill Levitin, and Oscar Nierstrasz. Evaluating an architecture conformance monitoring solution. 7th IEEE International Workshop on Empirical Software Engineering in Practice, pages 41–44. IEEE, 2016.

[16] Andrea Caracciolo, Mircea F. Lungu, and Oscar Nierstrasz. A unified approach to architecture conformance checking. IEEE International Conference on Software Architecture, pages 41–50. IEEE, 2015.

[17] Andrea Caracciolo, Mircea Filip Lungu, and Oscar Nierstrasz. How do software architects specify and validate quality requirements? Software Architecture: 8th European Conference, ECSA 2014, Vienna, Austria, August 25-29, 2014. Proceedings, pages 374–389. Springer, Cham, 2014.

[18] Paul Clements. A survey of architecture description languages. International Workshop on Software Specification and Design '96, pages 16–25. IEEE Computer Society, Mar 22, 1996.

[19] Kris. De Schutter. Automated architectural reviews with Semmle. 28th IEEE International Conference on Software Maintenance, pages 557–565. IEEE, 2012.

[20] Lakshitha de Silva and Dharini Balasubramaniam. Controlling software architecture erosion: A survey. *The journal of systems and software*, 85(1):132–151, 2012.

[21] Lakshitha R. de Silva. Towards controlling software architecture erosion through runtime conformance monitoring, May 2014. PhD thesis, University of St Andrews.

[22] Stephen. G. Eick, Todd. L. Graves, Alan. F. Karr, J. S. Marron, and Audris. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[23] Graham R. Gibbs and Celia Taylor. How and what to code. http://onlineqda.hud.ac.uk/Intro_QDA/how_what_to_code.php. The method of coding. Accessed: May 16 2017.

[24] Les. Hatton. Testing the value of checklists in code inspections. *IEEE Software*, 25(4):82–88, 2008.

[25] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research, 2004.

[26] Lorin Hochstein and Mikael Lindvall. Combating architectural degeneration: a survey. *Information and Software Technology*, 47(10):643–656, 2005.

[27] Jens Knodel and Daniel Popescu. A comparison of static architecture compliance checking approaches. Working IEEE/IFIP Conference on Software Architecture. IEEE, 6-9 Jan. 2007 Jan, 2007.

[28] Robert Lagerstedt. Using automated tests for communicating and verifying nonfunctional requirements. T1st International Workshop on Requirements Engineering and Testing (RET), pages 26–28. IEEE, 2014.

[29] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[30] Radu Marinescu and George Ganea. incode.rules: An agile approach for defining and checking architectural constraints. Proceedings of the 2010 IEEE 6th International Conference on Intelligent Computer Communication and Processing, pages 305–312, 2010.

[31] David L. Parnas. Software aging. Proceedings of the 16th International Conference on Software Engineering, pages 279–287. IEEE, 1994.

[32] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52, 1992.

[33] Leo Pruijt, Christian Koppe, Sjaak Brinkkemper, and Jan M. Werf. The accuracy of dependency analysis in static architecture compliance checking. *Software: Practice & Experience*, 47(2):273–309, 2017.

[34] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131, 2008.

[35] C. Sauer, D. R. Jeffery, L. Land, and P. Yetton. The effectiveness of software development technical reviews: a behaviorally motivated program of research. *IEEE Transactions on Software Engineering*, 26(1):1–14, 2000.

[36] C. Stringfellow, C. D. Amory, D. Potnuri, A. Andrews, and M. Georg. Comparison of software architecture reverse engineering methods. *Information and Software Technology*, 48(7):484–497, 2006.

[37] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha. Recommending refactorings to reverse software architecture erosion. 16th European Conference on Software Maintenance and Reengineering, pages 335–340. IEEE, 2012.

[38] Ricardo Terra and Marco T. Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 39(12):1073–1094, Aug 25, 2009.

[39] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *The Journal of Systems & Software*, 61(2):105–119, 2002.

Appendices

Appendix A
Interview Guide

A.1 Introduction

We conduct these interviews to get an understanding of how the software architecture is managed and enforced in this project. Your answers will be confidential and any published results will be anonymous.

A.2 Questions

1. Formalities
   - What is your title here at Bosch?
   - What are your main tasks?
   - For how long have you been working at Bosch and in this project?
   - Tell us about your previous work experiences.
2. How is the software architecture managed and enforced in this project?
   - Are you familiar with the architectural rules and guidelines?
     - How often do you look at them?
     - Are they something you think about in your daily work?
     - Do you find them easy to follow and understand?
     - Are there any specific rules you consider more difficult to follow than others?
     - Have you received any form of education regarding the architectural guidelines?

  - When and in what way?
  - Do you consider it to have been enough?
- Is the architectural guidelines something that you and your colleagues discuss?
  - How often?
  - Have you ever been remarked on not following an architectural rule or guideline? Do you remember the specific rule or what it was about?
  - Have you ever remarked on a colleague not following an architectural rule or guideline when reviewing his or her code? Do you remember the specific rule or what it was about?

3. Are you satisfied with how the architectural rules are managed today?

   - What changes would you like to make?
   - If you could decide, how would the guidelines be communicated?
   - Do you think the system follows the intended architecture in a good way?
   - What is your view on having automatic tests on these guidelines? Do you think you or the project could benefit by introducing such tests?

EXAMENSARBETE Implementing and Evaluating the Use of Automatic Tests for Architectural Rules
STUDENT Johan Bäckström, Fredrik Karåker Sundström
HANDLEDARE Elizabeth Bjarnason (LTH), Robert Lagerstedt (Bosch)
EXAMINATOR Martin Höst (LTH)

Automatiska tester för arkitekturregler i mjukvaruprojekt

POPULÄRVETENSKAPLIG SAMMANFATTNING **Johan Bäckström, Fredrik Karåker Sundström**

En stor del av testningen i mjukvaruprojekt är numera ofta automatiserad till så hög grad som möjligt. Trots detta förlitar sig kontrollen av mjukvaruarkitekturen i de flesta projekt på manuella granskningar. Genom att automatisera även detta kan man minska antalet brister i mjukvaruarkitekturen och samtidigt spara in på den tid som annars måste läggas på manuella granskningar.

För att se till att en mjukvaruarkitektur efterföljs i ett projekt formulerar man ofta regler som utvecklarna sedan ska följa. För att se till att dessa regler följs kan de testas automatiskt vilket ger möjligheten att ge utvecklaren direkt feedback om de finns fel som måste rättas till. Sådana automatiska tester har i det här arbetet implementerats och utvärderats i ett mjukvaruprojekt på Robert Bosch AB.

Det finns en mängd olika verktyg och ramverk som är gjorda för att testa mjukvaruarkitektur. Vissa verktyg fokuserar på att testa vissa delar, till exempel beroenden mellan mjukvarukomponenter, medan andra fungerar som större ramverk med bredare testningsmöjligheter. Gemensamt är dock att dessa verktyg ofta kräver mycket underhåll och anpassning för att fungera i verkliga projekt. Detta gäller särskilt om ett projekt redan pågått under en tid innan ett sådant verktyg börjat användas. En annan metod kan därför vara att själv implementera arkitekturtester för det specifika projektet. Dessa tester kan på så vis vara helt skräddarsydda till den kod och de arkitekturregler som finns definierade i projektet istället för tvärt om.

I detta projekt implementerades ett antal tester som automatiskt kontrollerade att den kod som utvecklarna skrev följde utvalda arkitekturregler. Om några tester identifierade kod som bröt mot de definierade reglerna visades information om felet och hur det borde lösas i samma verktyg utvecklarna utnyttjade för att göra manuella granskningar. På så sätt kunde testerna införas utan att det innebar några stora förändringar för utvecklarnas arbetsprocess.

De implementerade testerna utvärderades i projektet och visade att manuell granskning av kod inte räcker till för att hitta alla regelöverträdelser. Resultaten visade att ungefär två tredjedelar av de överträdelser av arkitekturregler som våra tester identifierade missades under den manuella granskningen. Utvecklarna visade sig också vara positiva till de automatiska testerna och förstod den återkoppling de gav. Resultaten från arbetet tyder därför på att automatiska tester kan fungera som ett bra alternativ eller komplement till manuella metoder inom kontroll av mjukvaruarkitektur.