CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Eliciting Small Scale Architectures with Maintainability Scenarios

Master's thesis in Software Engineering

Aðalsteinn Axelsson

Department of Computer Science and Engineering
Division of Software Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018:NN

Eliciting Small Scale Architectures with Maintainability Scenarios

AÐALSTEINN AXELSSON

Department of Computer Science and Technology
Division of Software Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Eliciting Small Scale Architectures with Maintainability Scenarios
AÐALSTEINN AXELSSON

Master's Thesis 2018:NN
Department of Computer Science and Technology
Division of Software Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Eliciting Small Scale Architectures with Maintainability Scenarios
AÐALSTEINN AXELSSON
Department of Computer Science and Technology
Chalmers University of Technology
University of Gothenburg

Abstract

There are available methods for software architecture elicitation and evaluation. Many of them have been applied in industry with great success. These methods are well suited to bigger operations, and there have been many successful applications in large scale software production. These successful methods are less effective for small to moderate scale software production in comparison. In addition, existing elicitation methods frequently consider several quality attributes. In some cases, especially in smaller production, a specific quality attribute is the primary concern. There are few elicitation methods available that are designed for that context.
This study proposes an elicitation method designed for small to moderate scale software projects where achieving high maintainability is the primary concern. This method was named the Maintainable Architecture of Small Scale Elicitation Method or the MASSE method. This method consists of three phases each of which is comprised of several steps. These phases are executed iteratively until a satisfactory result is obtained. The phases are Presentation, Elicitation and Execution. During the Presentation phase the MASSE method is explained to relevant parties, during the Elicitation phase the requirements for the software architecture are elicited and during the execution phase the architecture is designed based on these requirements. The method utilizes Quality Attribute Scenarios as its main elicitation tool.
To test the effectiveness of the MASSE method it was applied in industry. The method was applied at an Icelandic engineering firm called EFLA, with relatively small software operations. There an existing program called EflaVefgatt needed to be redesigned to substantially increase its maintainability. Two iterations of the method were executed during which both the architecture design and the method were improved. Qualitative data was collected for both iterations through interviews. The interviews compared the existing software architecture with the architectures resulting from the two iterations of the MASSE method. The results were positive after the first iteration and after the improvements made during the second iteration they were more positive and deemed satisfactory. Subsequently the process ended as a success.

Keywords: Software Architecture, Software Architecture Elicitation, Quality Attribute Scenarios, Maintainability and Small Scale Software Production

Acknowledgements

I would like to thank Imed Hammouda for his initial input, Robert Feldt examinor and Hörður Þórðarson industrial advisor. Finally, I would like to give a special thanks to Magnus Ågren ,who was my daily supervisor, for the great amount of time and effort he invested in this project.

AÐALSTEINN AXELSSON, Gothenburg, June 2018

Contents

List of Figures x

List of Tables xi

1 Introduction **1**
- 1.1 Software Architecture and Maintainability . . . 1
- 1.2 Thesis Outline . . . 2

2 Related Work **3**
- 2.1 Quality Attribute Scenarios . . . 3
- 2.2 Architectural Concepts . . . 4
  - 2.2.1 Architectural Patterns . . . 4
    - 2.2.1.1 Broker Pattern . . . 4
    - 2.2.1.2 Service-Oriented Architecture . . . 4
    - 2.2.1.3 Microservices . . . 5
  - 2.2.2 Architecture Tactics . . . 5
    - 2.2.2.1 Split Module . . . 5
- 2.3 Methods and Concepts Relating to Software Architecture and Maintainability . . . 6
  - 2.3.1 Attribute-Driven Design . . . 6
  - 2.3.2 Architecture Trade-off Analysis Method . . . 6
  - 2.3.3 Architecture-Level Modifiability Analysis Method . . . 7
  - 2.3.4 Scenario-based Software Architecture Reengineering . . . 7
  - 2.3.5 Architecture Level Prediction of Software Maintenance . . . 8

3 Research Method **9**
- 3.1 Design Science Framework . . . 9
- 3.2 The Project Within the Framework . . . 10
- 3.3 Threats to Validity . . . 11
  - 3.3.1 Construct Validity . . . 12
  - 3.3.2 Internal Validity . . . 12
  - 3.3.3 External Validity . . . 12
  - 3.3.4 Reliability . . . 12

4 Case Company **13**
- 4.1 Databases and Web Solutions department . . . 13
- 4.2 EflaVefgatt . . . 14
- 4.3 Maintainability and EflaVefgatt . . . 15

5 Iteration 1 **17**
- 5.1 Problem Investigation . . . 17
- 5.2 Solution Design . . . 17
  - 5.2.1 Rationale for the Method . . . 18
  - 5.2.2 Method Description . . . 18
    - 5.2.2.1 Presentation . . . 19
    - 5.2.2.2 Elicitation . . . 19
    - 5.2.2.3 Execution . . . 19

5.2.3 The MASSE Method and Maintainability 20
5.3 Design Validation 20
5.3.1 Internal Validity 20
5.3.2 Trade-offs 21
5.3.3 External Validity 21
5.4 Implementation 21
5.4.1 Presentation 22
5.4.1.1 Stakeholders 22
5.4.1.2 Method Presentation 22
5.4.2 Elicitation 23
5.4.2.1 Business Drivers 23
5.4.2.2 Quality Attribute Scenarios 23
5.4.3 Execution 27
5.4.3.1 Architectural Approaches 27
5.4.3.2 Architecture 28
5.5 Evaluation 30
5.5.1 Evaluation Process 30
5.5.2 Results 32

6 Iteration 2 **33**
6.1 Solution Design 33
6.1.1 Rationale for Changes to the Method 33
6.1.2 Method Revision 34
6.1.2.1 Presentation 35
6.1.2.2 Elicitation 36
6.1.2.3 Execution 36
6.2 Design Validation 37
6.3 Implementation 37
6.3.1 Presentation 37
6.3.2 Elicitation 38
6.3.3 Execution 38
6.3.3.1 Architectural Approaches 38
6.3.3.2 Architecture 39
6.3.3.3 Architecture Validation 41
6.4 Evaluation 42

7 Discussion **43**
7.1 Future Work 44

8 Conclusion **45**

A Appendix **II**

List of Figures

4.1 Component diagram depicting the context of EflaVefgatt. 14

5.1 Architecture from Iteration 1. 30

6.1 Architecture from Iteration 2. 40
6.2 Detailed view of an example proxy. 41

List of Tables

5.1 QAS 1 25
5.2 QAS 2 25
5.3 QAS 3 25
5.4 QAS 4 26
5.5 QAS 5 26
5.6 QAS 6 26
5.7 QAS 7 26

A.1 General Modifiability Scenarios II
A.2 Iteration 1 Questionnaire and Results II
A.3 Iteration 2 Questionnaire and Results IV

1
Introduction

This section introduces various topics and concepts relevant to this research. The objective is to introduce the context of this research and explain its incentives.

1.1 Software Architecture and Maintainability

The importance of software architecture is increasing in modern software development. For example a lot of large system failures can be traced to the neglect of architecture principles [1]. Therefore organizations are giving consideration to software architecture practices. One of these considerations is the maintainability of software systems. The ISO 25010 standard [2] provides a definition of maintainability as the degree of effectiveness and efficiency with which a product or system can be modified. The maintainability of a system, determines the effort spent on its development after its initial release. It is established that only something between a fourth and a third of the effort put into a software system is spent towards actually creating the software system [3]. The rest of the effort is spent on maintaining the system. The greatest opportunity to reduce costs in software development is therefore in reducing maintenance costs.
To summarize, taking advantage of software architecture practices to create maintainable software systems can substantially decrease development costs. This is not unknown to the software industry and many companies adhere to mature software architecture practices. These practices include using software architectural tactics and patterns to guide the design of software architectures, but also adhering to mature processes. This would include software architecture evaluation methods, as well as software architecture elicitation methods. There is an apparent lack of research available on the relevance of the size of a software project to the effectiveness of applying an architecture evaluation method to its architecture. There is not much empirical data available to illustrate this point. Despite this there is substantial argumentation for prevalent methods being more effective for large scale software development. One of the leading methods in architecture evaluation is the Architecture Trade-off Analysis Method or ATAM which is proposed in [4]. This method involves bringing together a group of stakeholders as well as technically adept personnel. In addition, a strong leader experienced in applying the method is needed to guide the process. Similarly methods like Scenario-based Software Architecture Reengineering [5] and Architecture-Level Modifiability Analysis Method [6] require substantial experience in executing the method as well as multiple parties being involved and a lot of time be spent on elicitation. These methods will will be elab-

orated on in Chapter 2. Therefore, they are better suited for large scale software operations. Thus, there exists a certain lack of methods for small to moderate scale software development. In addition to this there is a lack of software architecture elicitation methods that target the maintainability quality attribute specifically. There are methods available that are designed to work with multiple quality attributes, but compromises are needed to accommodate all of them.
The purpose of this research is to design a method of eliciting software architectures that is suited to these constraints. Namely, creating a software architecture elicitation method suitable for creating software architectures for small to moderate operations, that when implemented result in highly maintainable software products. The possibility of using quality attribute scenarios to achieve this goal will be explored.
This is done in cooperation with a company in industry where this problem was encountered and a solution needed. Proposing the aforementioned method, evaluating it and creating a software architecture as a byproduct are the main objectives. The proposed method and its evaluation provide research value whilst the architecture provides direct benefits to the case company.

1.2 Thesis Outline

The thesis starts with Chapters 2 and 3 on related work and the research method respectively. This is followed by Chapter 4 on the case company under consideration. The research followed an iterative design science cycle as is described in the research method section. Chapters 5 and 6 describe the execution of these iterations. Finally there is a discussion of the results in Chapter 7 and a conclusion in Chapter 8.

2 Related Work

The purpose of this chapter is to introduce concepts related to the research being conducted. This will include research that has been conducted within the same domain as well as concepts that are used in the construction of design artifacts.

2.1 Quality Attribute Scenarios

Quality attributes can be defined as properties of a system that are measurable or testable and can be used to indicate how well a system meets the needs of its stakeholders [7]. Quality attribute scenarios (QAS) are a tool in software engineering that is typically used in the testing of software architectures and software systems with regard to quality attributes. Kazman [8] defines them as descriptions of a systems' proper response to certain stimuli. In the analysis of software architectures they are used to try and determine whether a candidate architecture is likely to meet its quality goals. They are also used to evaluate whether systems that have been produced meet their quality goals. As previously mentioned a QAS is a short description of a systems´ intended response to a certain stimulus. This description consists of six parts namely:

1. Source: the source of the stimulus.
2. Stimulus: some condition that affects the system and needs consideration.
3. Artifact: the specific part of the system that was affected by the stimulus.
4. Environment: the context in which the stimulus occurs.
5. Response: the action taken in response to the stimulus.
6. Response measure: the measure that is used to evaluate the system response.

QAS are associated with a specific quality attribute and what the six parts listed above are composed of varies depending on which quality attribute that is. In [7] a listing of general modifiability QAS is provided. This listing is provided in Table A.1 in the appendix. This table can be used as a guideline for creating modifiability QAS to ensure consistency. This table is not exhaustive however and in some cases assigning values that are not listed in the table for some or all of the six parts is necessary.

2.2 Architectural Concepts

This research entailed producing software architectures. These architectures relied on several known architectural principles and constructs. The purpose of this section is to introduce these architectural principles and constructs.

2.2.1 Architectural Patterns

In [7], an architectural pattern is described as a set of design decisions that are frequently encountered in industry, that has characteristics that can be identified to allow reuse and describes a class of architectures. Furthermore an architectural pattern should establish a relationship between:

- *A context.* A situation that commonly arises in the world that presents a problem.
- *A problem.* An appropriately generalized version of the problem that arises in the context.
- *A solution.* An appropriately abstracted solution to the problem described in the context.

Several patterns are frequently considered in the design of just one program. The design could be guided by a combination of several patterns or variations of patterns. Furthermore, the design of a subset of a program can be guided by one pattern whilst other subsets could be guided by other patterns. To summarize, this means that programs are not simply designed by choosing one architectural pattern but can take ideas from several.

2.2.1.1 Broker Pattern

A description of the Broker pattern is provided in [7]. The context of the Broker pattern is a situation where a system is composed of several services across several servers. The problem that arises in this situation is that these services need to exchange information with each other and service users somehow. Moreover the problem is finding a way so that they can exchange information without needing to know where the information comes from or how it is transferred.
The proposed solution is to add a component called a broker to the design. This component is used to separate the clients from the servers. It handles all communications between these two groups. The servers do not have to communicate directly with the clients. This allows services to be replaced without changes to the clients since communications between the clients and the broker are unchanged.

2.2.1.2 Service-Oriented Architecture

A description of the Service-Oriented Architecture (SOA) pattern is provided in [7]. The context is a situation where multiple services are provided and consumed by clients. Clients need to be able to use these services without knowing their exact implementations. The problem that arises in this situation is that a way to support

the interoperability of different services that can be running on different platforms is needed. Some way to find these services and combine them in useful ways is needed. The solution is the SOA pattern. In this pattern services and clients are different components. The services are components that are mostly independent and could be deployed in different systems and on different platforms. Interfaces are then used to allow the services to communicate with clients and each other.
To help the components communicate or to add to the structure of the system some additional components are often present. The most notable of these is what is called an enterprise service bus (ESB). An ESB is a type of broker like the one described above. It handles messages passing between the clients and the services. Additionally an ESB can often act as a translator and convert messages from one protocol to another. Message passing is usually done through SOAP, REST or asynchronous messaging.

2.2.1.3 Microservices

In recent years a special case of the SOA pattern called microservices has been surfacing [9]. The word microservices started appearing around 2011 [9]. Given its young age there is no formal definition of microservices and the word does not always have the same meaning to different parties. However there are certain characteristics that are commonly associated with microservices. In [9] Randy Shoup formerly of eBay and Google was quoted, saying that microservices have three main characteristics. Namely:

- Microservices have a well defined and limited scope that only encompasses a few functionalities that can be conveyed by an API that is small in size.
- The microservices are very modular, the difference being that they are not only modular at the program level but at the deployment level allowing independent deployment of each service.
- The microservices contain their own permanent storage and do not share it with other services.

In addition [9] states that microservices commonly communicate through simple interfaces and the pattern does not use an ESB to handle communications.

2.2.2 Architecture Tactics

In [7] tactics are defined as techniques that can be used to achieve system goals in terms of quality attributes.

2.2.2.1 Split Module

The split module tactic involves taking a module that is large in size and splitting it into two or more smaller modules [7]. This tactic is intended to reduce the cost associated with making a change to the module. The tactic is only successful if the split is chosen to reflect changes to the module that are likely to be undertaken.

2.3 Methods and Concepts Relating to Software Architecture and Maintainability

The purpose of this section is to introduce other available methods for eliciting and evaluating architectures.

2.3.1 Attribute-Driven Design

There are methods available for creating software architectures from quality attribute requirements. The most notable of these being Attribute-Driven Design (ADD) which is defined in [10]. This method selects certain requirements and uses them as architectural drivers. This means that the selected requirements guide the design of the architecture whilst secondary requirements are eventually satisfied but do not guide the design. The requirements chosen as drivers are likely to be both important to the stakeholders and difficult to satisfy from an architectural perspective. Applying this method with maintainability requirements as architectural drivers is a technique that can be used to create architectures for highly maintainable systems.

2.3.2 Architecture Trade-off Analysis Method

In addition to these methods there are also methods for evaluating software architecture candidates with regard to quality attributes. This category of methods often relies on QAS for evaluation.
The most notable software architecture evaluation method is the Architecture Trade-off Analysis Method commonly referred to as the ATAM method. In [4] ATAM is described as a method for evaluating software architectures in relation to quality attribute goals. The purpose is to expose potential flaws in the architecture that could prevent it from delivering on an organization's business goals. The core idea behind it is to create QAS along with the relevant stakeholders and use those scenarios to evaluate an existing architecture. Specifically carrying out the method requires going through nine steps in four phases. Namely:

Presentation

1. Present the ATAM
2. Present business drivers
3. Present architecture

Investigation and Analysis

4. Identify architectural approaches
5. Generate quality attribute utility tree
6. Analyze architectural approaches

Testing

7. Brainstorm and prioritize scenarios
8. Analyze architectural approaches

Reporting

9. Present results

Step 5 mentions a quality attribute utility tree. Such a tree is composed by first listing all relevant quality attributes as nodes. Their children are then categories of quality attributes who in turn have sub-categories or QAS as their children.

2.3.3 Architecture-Level Modifiability Analysis Method

Another notable method that evaluates software architecture candidates is the Architecture Level Modifiability Analysis or the ALMA method [6]. The main aspect that sets the ALMA method apart from ATAM is that it focuses on modifiability rather than general quality attributes. Similar to ATAM, this method is composed of multiple steps, namely the five steps:

- Goal selection
- Software architecture description
- Scenario elicitation
- Scenario evaluation
- Interpretation

Like ATAM the ALMA method uses QAS for the architecture evaluation. The method creators had found that modifiability analysis normally has one of three goals. Namely prediction of future maintenance cost, identification of system inflexibility or the comparison of two or more alternative architectures. The execution of the steps is based around the appropriate goal.

2.3.4 Scenario-based Software Architecture Reengineering

The evaluation method Scenario-based Software Architecture Reengineering is proposed in [5]. This is a method that is used to assess the quality of an architecture with respect to quality attributes and iteratively improve on them where the quality goals are not met. The method is described as having inputs of a requirements specification that has been updated, as well as the existing architecture. The output of the method is then an improved software architecture. The method works by iteratively executing four steps namely:

1. Incorporate new functional requirements in the architecture.
2. Software quality assessment.
3. Architecture transformation.
4. Software quality assessment.

The intention is that the method helps satisfy quality goals but not just functional goals as can be the case with other methods. In [5] the method is applied on a beer can inspection system. A total of 5 method iterations were executed with respect to maintainability and reusability. At the end of the process there had been some trade-offs seen but overall a satisfactory improvement was made to the architecture design in that case.

2.3.5 Architecture Level Prediction of Software Maintenance

The Architecture Level Prediction of Software Maintenance method is proposed in [11].It is a method used for predicting the average amount of effort that needs to exerted towards a maintenance task. These amounts can be used to compare architecture candidates. The method takes for possible inputs namely: the architecture design under consideration, the specification of requirements, any expertise from participants and possibly but not necessarily historical data. It works by executing a series of steps namely:

1. Identify categories of maintenance tasks
2. Synthesize scenarios
3. Assign each scenario a weight
4. Estimate the size of all elements.
5. Script the scenarios
6. Calculate the predicted maintenance effort.

In [11] the method is applied to a design of a software architecture, specifically for a haemo dialysis machine. The method successfully produces estimates of effort needed for maintenance tasks.

3 Research Method

This chapter discusses the design science framework that the research was conducted within. This consists of a description of the framework along with a discussion on how the research project at hand fits into the framework. This is followed up by a section on threats to validity.

3.1 Design Science Framework

This project is based in the Design Science (DS) methodology. Hevner [12] provides a framework for doing design science research. This framework explains how solutions to practical problems can contribute to the knowledge base for the space in which the research is conducted. Another adaptation of Design Science is provided by Wieringa in [13]. Wieringa's framework builds on Hevner's framework and elaborates on the difference between knowledge problems and practical problems, and the relationship between them.
Wieringa's framework defines these two kinds of problems from the point of view of the stakeholders. A practical problem is defined as the difference between the world as it is and the way the stakeholders would want it to be. A knowledge problem, on the other hand, is defined as the difference in the current and the desired knowledge base of the stakeholders. The author goes on to suggest speaking of practical problems that have solutions and knowledge questions that have answers. This is to prevent ambiguity.
Wieringa suggests an iterative cycle as the way to conduct Design Science research. This cycle is dubbed The Regulative Cycle. The reason it is iterative is to facilitate not only creating new artifacts but also improving upon existing ones. This in turn means it can be used to create an artifact and subsequently improve it until a satisfactory result is obtained. Each iteration is composed of several steps. These steps are:

1. Problem Investigation/Implementation Evaluation
2. Solution Design
3. Design Validation
4. Solution Implementation

Since it aims to understand the world without making any alterations to it the Problem Investigation is a knowledge question. The problem investigation step entails studying the problem at hand and can also include studying the possible conse-

quences of leaving it unsolved. In later iterations calling this step Implementation Evaluation, is more appropriate. During that step the problem can be reconsidered but evaluating the implementation from the previous iteration is the main objective.
The Solution Design is a practical problem. During this step a solution to the problem from the previous step is created. The author suggests that using the word solution is precarious, since the product of this step may not solve anything at all, despite that being the objective.
The Design Validation step is a knowledge step that attempts to speculate whether implementing the design properly will solve the problems for the stakeholders. Three knowledge questions are asked. The first one is of Internal Validity, which asks whether the solution will, in the given context, solve the problems encountered by the stakeholders. The second one is on Trade-Offs, which asks how slight alterations to the solution would impact its effectiveness. Finally, there is a question of External Validity that asks whether the solution would be as effective if implemented in a slightly altered context.
During the Implementation step a version of the solution is implemented. The implementation can then feed into the Implementation Evaluation step of the next iteration completing the cycle.

3.2 The Project Within the Framework

The project used the Wieringa framework. The steps of the Regulative Cycle were used as a guideline and the first step was Problem Investigation. The execution of this step as well as subsequent steps is described in chapters 5 and 6. However, at this point the problem can be summarized quickly so that the knowledge question it raises can be presented.
The research problem is that there is a lack of a method to elicit software architectures, that is specialized for small to moderate scale operations where maintainability is of high importance. This is not currently part of the knowledge base. The knowledge question raised is the first research question and will be referred to as RQ1:

RQ1. How can a software architecture be elicited and evaluated for a small system requiring high maintainability?

The next step is the solution design. The research question it raises is a result of the problem investigation as it asks whether quality attribute scenarios can answer the knowledge question asked in RQ1. The question is:

RQ2. Can quality attribute scenarios be used to drive software architecture elicitation, for systems that are small to moderate in size and need to be highly maintainable?

Wieringa describes the design validation as the process of answering three questions. Namely questions of internal validity, trade-offs and external validity. These questions are answered in detail as part of the iteration sections.

The implementation step was an application in industry involving an existing program with an existing architecture at the engineering firm EFLA as will be explained in more detail in chapter 4. The implementation involved producing a new architecture for that program.
Two iterations were executed and both iterations ended in Implementation Evaluation steps. The presentation in the Wieringa framework is slightly different since this step was considered as the last step of each iteration rather than the first step of the next. The reason being that evaluation was considered an important step and was therefore performed at the conclusion of both iterations rather than only after the first.
Since the methods´ purpose is the creation of software architectures, the method was judged by the architecture that it produced. The evaluation process for the method was therefore designed to compare an architecture that was the output of this method to an architecture created by other means. The comparison was between the architecture produced and the current architecture. To pass the evaluation the architecture that was elicited with the new method needed to achieve the system goals significantly better than the original architecture. This was reasoned to indicate that the proposed method was effective in eliciting software architectures in the studied case. The comparison of the two architectures was designed as a comparison of the existing program and a prototype of the proposed system.
In [14] an architectural prototype is defined as a set of executables created with the objective of investigating stakeholder concerns. It goes on to define two main classes of architectural prototypes. First there are exploratory prototypes which are generally used to refine requirements specifications or to discuss alternative approaches. Secondly there are experimental prototypes. These prototypes are used to measure certain aspects of a system. They can also be used to test certain qualities. The prototype used in the evaluation is best described as an experimental prototype.
In [14] it is explained that architectural prototypes are used to study architectural quality aspects but not directly provide functionality. The evaluation prototype was based on these concepts. It was decided it should be an incomplete implementation of the new architecture. It was to be capable of interfacing with the same entities as the original program but did not need to be functional. Questionnaires answered by relevant participants in the process were used to provide qualitative data for analysis.
The architecture description it implements, was to be detailed enough so that a prototype that fulfills these requirements could be created from it. The comparison between the two architectures was to involve an analysis. This analysis considered the quality attribute scenarios available from the elicitation process. The questionnaires, along with analytic reasoning would be used to reason about the quality of each architecture. The prototypes are described in detail in chapters 5 and 6.

3.3 Threats to Validity

This section discusses possible threats to validity for this research.

3.3.1 Construct Validity

When participants answered questionnaires measures were taken to ensure that interviewees understood any questions asked in the same way as was intended. However the respondents that were interviewed both formally and informally were often less focused on the research objectives at hand and had their minds focused on practical matters.
The approach taken was to use the output of the artifact for its evaluation. This was the best available approach but there is nevertheless a possibility that using the rigorous approach did not have as much of an effect on the output generated as was hoped. Redesigning the architecture of the system without using the elicitation method could also have yielded some positive results.

3.3.2 Internal Validity

The data collected was somewhat subjective since it entailed collecting expert opinions. The stakeholder, who was the main interviewee, was excited about this process and had big hopes for its outcome. There was a certain risk this might cause him to be overly positive. With this in mind, he was encouraged to try to be as objective as possible.

3.3.3 External Validity

The generic nature of the artifact together with the importance of maintainability in software production make this artifact potentially useful in a broad section of the software industry.
All the same, this research was conducted at one company with a limited number of interviewees. Therefore it would be a bold assumption to assume from this research that the artifact is applicable in such a large domain. Factors such as the lack of rigor in the software engineering practices at the case company could potentially have had a positive impact on the results.

3.3.4 Reliability

The questionnaire used was specifically tailored to the case at hand. This could make replicating the data collection difficult.

4
Case Company

The company whose problems led to this research being conducted is an Icelandic engineering firm called EFLA. EFLA operates primarily as a contractor for various projects in Iceland as well as abroad. EFLA has a staff comprised of engineers of various disciplines, technicians and project managers among others. EFLA is not a software company but some projects conducted at the company have led to a need for software developers. EFLA's Databases and Web Solutions department is an example of this.

4.1 Databases and Web Solutions department

As the name implies the department works with databases as well as creating web reports that present data. The reasons for its inception are that electrical engineers at EFLA were contracted to design, program and deploy Programmable Logic Controllers (PLC) for various customers. These controllers were used to control industrial equipment. These PLCs as well as control equipment were logging lots of data which presented an analytic opportunity. The need to systematically handle this data and create applications to view it led to the inception of the department. The data is presented in automated web reports. These reports use data from various sources including data logged in databases and data directly from Programmable Logic Controllers (PLCs). The data is displayed in various forms (such as tables, graphs, bar charts etc.) in front-end web applications. At the heart of all this is a program called EflaVefgatt. EflaVefgatt is a server program that is written in JavaEE. It acts as a proxy between the various data sources listed above and the front-end applications as well as providing several other services.
This program was initially small but its size and importance gradually grew over time. Since its growth was unexpected and gradual, it lacked a clear design plan. Therefore the resulting program is lacking in flexibility and problems have been encountered in maintaining and adding new components to it. EFLA wanted to redesign EflaVefgatt to address these difficulties. Therefore the people at EFLA started giving consideration to the software architecture of the system. EFLA have identified the maintainability of the current system to be a big source of problems. These problems centered around EflaVefgatt are the main topic of this discussion. The department is small and ad hoc approaches are common. Mature processes for developing software are not in place but the employees are interested in changing this.

4.2 EflaVefgatt

As previously explained, EflaVefgatt is a program involved in giving front-end applications access to data from databases as well as directly from Programmable Logic Controllers (PLCs). EflaVefgatt has other purposes but they are not the focus for this discussion.
EFLA has several projects that use EflaVefgatt as a proxy server between front-end applications and data sources. For security reasons, many of these projects are closed systems and do no networking with the outside world. Therefore a local instance of EflaVefgatt is used that only interacts with the components of that specific system. This is the reason that there is no central version of EflaVefgatt and all projects have their own version. This also applies where security concerns are not of high importance.
A generic project of this kind will consist of a front-end application, EflaVefgatt and some data-sources. EflaVefgatt can communicate in three ways with data sources. It can access databases directly or it can get data from PLCs either through a KEPServer or directly through an HTTP protocol. A KEPServer is a connectivity platform that can speak with various industry devices and manage them through one interface [15]. An integrated system for such a project is depicted in Figure 4.1.

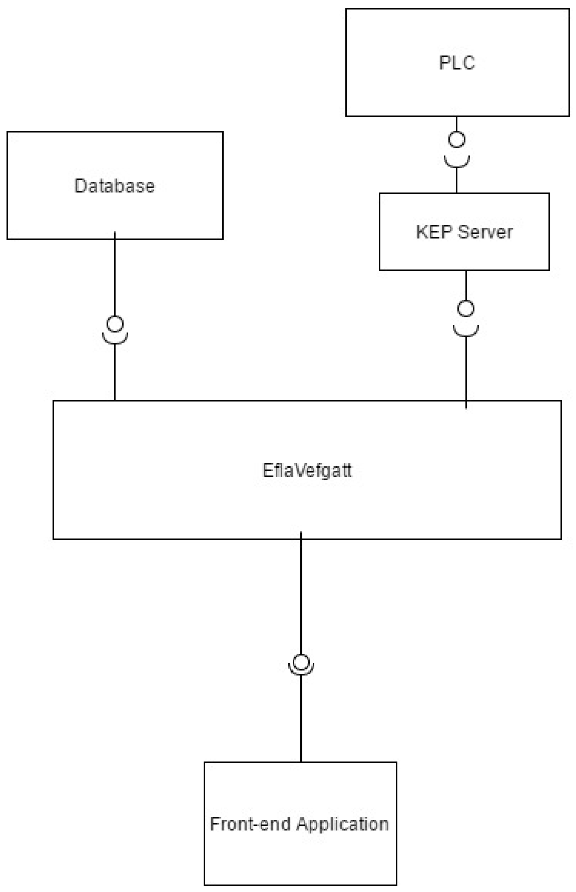

Figure 4.1: Component diagram depicting the context of EflaVefgatt.

Depending on the situation, a front-end application will communicate with a database,

PLCs or both and EflaVefgatt will handle the traffic. EflaVefgatt can also write to databases and it is common for it to write information from PLCs to databases.
Since EflaVefgatt is a JavaEE project it is composed of classes which in turn are arranged into packages. For EflaVefgatt each package is intended to encompass a group of related classes. The package relevant to this discussion is termed services. As the name implies this package is composed of several services. These services work together to read and write data as well as performing other related actions. There are lots of dependencies between these services. The front-end applications that use these services also depend on these services individually and commonly need to use many of the services in combination. There is no clearly defined contract between the front-end applications and EflaVefgatt in place. EflaVefgatt is also maintained continually and clear versions of it are not defined. Finally the services package of EflaVefgatt has grown to a large size and because of the heavy coupling between its services this has become a problem.
These issues combine into a maintainability problem. Changing the services is tedious since the developer has to consider the impact the changes might have on the other services as well as the front-end applications. This also means that creating new integrated projects with EflaVefgatt as a proxy server becomes more difficult since any adjustments to EflaVefgatt that are needed will require substantial work. This maintainability problem was identified as the main problem with EflaVefgatt by the relevant stakeholders.

4.3 Maintainability and EflaVefgatt

This section explains the concept of maintainability and what it means in the context of EflaVefgatt.
The ISO/IEC 9126 standard [16] has been used for many years to define quality attributes. In 2011 it was revised and replaced with the new standard ISO/IEC 25010 [2]. Despite that the 9126 standard is still used by some parties today. Both these standards split maintainability into several sub-qualities. The 9126 standard defines maintainability as the capability of the software product to be modified and splits it into the following sub-qualities:

- Analyzability
- Changeability
- Stability
- Testability
- Maintainability compliance

The 25010 standard defines maintainability as the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers and splits it into the following sub-qualities:

- Modularity
- Reusability
- Analysability

- Modifiability
- Testability

Despite having been replaced by the 25010 standard the 9126 can be used whenever considered appropriate. Therefore a discussion on both standards is warranted.
As was explained previously, the primary maintainability concerns are too much coupling and bulky system parts which together contribute to difficulties in modifying the program. This indicates that the updated 25010 standard is more appropriate. The reasons for that being that the standard has modularity and modifiability as sub-qualities. Modularity is defined as the degree to which a software system is composed of discrete components with low coupling. Improving the modularity of the system will therefore address the problems of high coupling and bulky system parts. Modifiability is defined as the degree to which a software system can be effectively changed without damaging existing qualities. This describes the reported difficulties in making changes to the program. Therefore the 25010 standard will be used to define maintainability for this project. Moreover, the sub-qualities modularity and modifiability.

5
Iteration 1

The purpose of this chapter is to report on the execution of the first iteration of the Wieringa design cycle. This includes an evaluation even though the Wieringa framework would put that in the next iteration.

5.1 Problem Investigation

What the stakeholders for this problem solving project wanted to achieve was to create a new and better architecture for an existing piece of software and subsequently evaluate it. Moreover they wanted to improve upon its maintainability.
A common process to achieve this would entail using the Attribute Driven Design method (ADD) [10] to create an architecture that improves upon a particular quality attribute. This method would then be used to elicit several architecture candidates. Once those candidates had been created an evaluation would follow. Using a respected method like ATAM [4] would be a good choice to execute the evaluation. For many projects this process can be successful.
In this case however that option was considered but ultimately it was decided that it was ill-suited to this problem. The reason for this is that the process described above is too time-consuming, expensive and requires the involvement of too many parties. For enterprise software systems, having software architects create several architecture candidates and then gathering a large group of stakeholders to hold lengthy meetings, is well worthwhile. A lot is at stake when large enterprise software systems are created and spending a lot of time and resources to prevent an unsuccessful project is economical.
In this case however the subject is a small to medium sized software project that is only worked on by a handful of developers. Gathering a group of stakeholders for a lengthy evaluation is therefore not reasonable. The problem is that a simpler way to elicit and subsequently evaluate architectures is needed.

5.2 Solution Design

In this section a method is proposed for creating architectures for software systems that place high importance on maintainability.

5.2.1 Rationale for the Method

In chapter 4 it was described how the maintainability problems with EflaVefgatt led to a decision to redesign it and evaluate the resulting architecture. Originally it was concluded that using a combination of ADD [10] to elicit the architecture and subsequently using ATAM [4] was too much work to be feasible in this case.
There is a lack of methods to elicit architectures that target maintainability for smaller software systems. Therefore it was decided to look for new ways. The idea for the new method came about when ATAM was considered. ATAM involves creating software architecture candidates, creating QASs to evaluate whether the architecture satisfies its goals and then either rejecting or accepting the candidates. The new method has a different approach to this. The question asked was, since the QASs that were elicited ultimately determine whether the architecture is accepted or not could creating the scenarios beforehand and using them to guide the design be a way to elicit architectures and at the same time implicitly evaluate them?
The intention was to establish a method that produces highly maintainable architectures. Therefore, some tool to ensure maintainability is needed. QAS are such a tool. Such a scenario is a type of requirement that can be used to test whether an architecture satisfies some quality goal. When everything is done properly the goal is satisfied if the architecture satisfies the scenario. If maintainability is chosen as the quality attribute, then maintainability scenarios, (maintainability quality attribute scenarios), will ensure that the architecture is in fact maintainable.
Some evaluation methods, including ATAM, use QASs for precisely this purpose. The intention here is to propose a method for eliciting architectures rather than just evaluating them. Therefore this method will elicit maintainability scenarios prior to creating architectures as was suggested previously. The specification of the maintainability scenarios will be used to guide the design. Eliciting architectures in this way ensures that they are maintainable since they satisfy the maintainability scenarios that were used to create them. Since maintenance requires modifying the system, elicitation and evaluation are highly intertwined which is the reasoning for this approach.

5.2.2 Method Description

The method that was designed was given the name Maintainable Architecture of Small Scale Elicitation Method and will be referred to in this text as the MASSE method.
ATAM is the leading method in the area of software architecture evaluation. This method consists of nine steps, each of which belongs to one of four phases. The MASSE method draws on the ATAM method with the major difference being that the MASSE method results in a new architecture whilst ATAM evaluates existing architectures. The MASSE method will also have steps that are parts of phases. There will be three phases and 5 steps. Namely:

Presentation

1. Present the method. The method is described to the involved stakeholders (typically customer representatives, the architect or architecture team, user

representatives, maintainers, administrators, managers, testers, integrators, etc.).

Elicitation

2. Elicitation of business drivers. Information is gathered from the involved stakeholders about their expectations for the system.
3. Generate a quality attribute utility tree. A quality attribute utility tree is elicited down to QAS level. The tree will begin at maintainability since other qualities are not considered in this context.

Execution

4. Propose architectural approaches. With the quality attribute utility tree from step 3 in hand architectural approaches that achieve these utilities are identified.
5. Propose an architecture. The proposed architectural approaches from step 4 are now used to create an architecture for the system in question.

These steps are performed iteratively until a satisfactory result is obtained. Over the lifetime of the software product it is expected that the architecture will need revising at which point these steps should be revisited. The phases will now be described in more detail.

5.2.2.1 Presentation

During this phase the MASSE method is described to the involved stakeholders and anyone participating in the process. The purpose of this phase is to make sure that all the participants understand the process. More specifically, that they understand how the process works, what the purpose of different steps is, what is to be gained from its execution and what their respective roles are in the process. This helps to ensure efficient participation from each party involved in the process.

5.2.2.2 Elicitation

During this phase QAS for the system being designed are elicited. First the project business drivers are elicited from the involved stakeholders. This is done to ensure that the overall project goals are understood and can guide the execution of the following steps. The purpose of this step is to have the goals from a business perspective elicited. This step is performed in the first iteration but can be skipped in later iterations. This is done through interviews with relevant stakeholders. Following the elicitation of the business drivers the QAS are written. This is done by working closely with the stakeholders, in particular the technical staff that will implement the architecture. General scenarios like the one in table A.1 in the appendix can be used as a guideline. When a consensus is reached on the QAS and a quality attribute utility tree is ready, the creation of the architecture can begin.

5.2.2.3 Execution

During this phase the architecture itself is created in two steps. First appropriate architectural approaches are identified. The purpose of this step is to structure the process. The approaches can include tactics, patterns or other established architectural approaches, that are likely to help satisfy the scenarios. Then when the approaches have been identified, they are used to systematically create an architecture for the system at hand. When the architecture has been created the process is completed.

5.2.3 The MASSE Method and Maintainability

When ATAM is executed a QAS tree is created. This is done to prioritize which quality attributes have the highest priority. This is intended to keep the focus on the highest priority attribute which should result in a bigger chance of a project being successful.
The MASSE method is designed with only maintainability in mind though and the various aspects that characterize other quality attributes were not considered in its design. Using it for other attributes is therefore less likely to be successful. The reasons for this are that maintainability is very iterative in nature. It revolves around the constant changing of a system and inevitably the changing of its design. The MASSE method is therefore designed to have multiple iterations that can take place at multiple times during the lifetime of the software. The architecture will therefore evolve with time as needed to preserve maintainability.

5.3 Design Validation

This section validates the design of the artifact by answering the three questions asked by Wieringa. Namely the questions of internal validity, external validity and trade-offs.

5.3.1 Internal Validity

The first question is that of internal validity. This question asks whether the design when implemented will solve the problems it was designed to solve.
There are two main characteristics to consider here. First, whether the artifact will produce architectures that are maintainable. Second, whether the artifact is suitable for small to moderate scale operations.
The method was designed using QAS [8]. They are a construct that has been shown to work well in predicting success with regard to certain quality attributes. They have not been tested to the same extent as an elicitation tool. However, they are capable of describing requirements with respect to quality attributes. The idea behind using QASs was to target the requirement that the artifact produce maintainable architectures.
Furthermore the maintainability of a system is an ongoing consideration for most software systems. Therefore the iterative nature of the MASSE method is particularly well suited to maintainability. Revisiting maintainability considerations regularly over the lifetime of a software product helps ensure that the product remains maintainable.

The second point was that the MASSE method needed to be suitable for small to moderate scale operations. To ensure this the MASSE method was designed to be simpler and less time consuming in execution than available alternatives.

5.3.2 Trade-offs

The trade-off question asks what impact changing the artifact slightly would have on its effectiveness if applied in the same context.
For this discussion we first consider changing the degree of complexity of the MASSE method. Lets assume the method was changed to include more steps, more detailed steps, and perhaps a direct evaluation. This could result in a certain improvement on the quality of the architectures it would produce. On the other hand, this would require more time be spent on the process and possible require more participants. This would be contrary to the methods´ intended design.
If we assume that the method was changed to be less defined, simpler in execution and to not include multiple iterations. This would result in shorter execution times but the process would approach becoming entirely ad hoc. This could potentially defeat the purpose in following a defined process in the first place.
Second, we consider not revisiting the architecture design at later times in the life of the software product. This could result in the software product produced by the method, having degrading maintainability. On the other hand though, revisiting the design regularly requires maintenance which is what having high maintainability is supposed to prevent. In the long run though not revisiting the method would result in a pile up of refactoring work. The cost of resolving this would most likely far exceed the costs saved by not revisiting the method.

5.3.3 External Validity

The external validity question asks how slightly changing the context that the artifact is applied in would affect its effectiveness.
For this discussion we first consider how the MASSE method would work if applied for large scale operations. Lets consider revisiting the architecture design regularly over the lifetime of a software product. This would provide a lot of supervision which is beneficial when a lot is at stake. It could potentially require that the architecture be redesigned which would require a lot of refactoring for a large system. This would not be achievable in all cases.
This could be the consequence of the second aspect we consider. This is the fact that the MASSE method prioritizes simplicity in execution. Large systems generally require a very thorough analysis since possible problem areas can be difficult to identify. Therefore, a shorter design time could be achieved but this could result in serious design flaws.

5.4 Implementation

The MASSE method was used to construct an architecture for an existing system with higher maintainability than the existing design. This section describes the execution of the various phases of the method.

5.4.1 Presentation

This section describes the presentation step of the first iteration. The section starts off with an analysis of the relevant stakeholders. Their expectations and involvement are the key drivers for this step and the steps to come. A description of the execution of the presentation step itself follows.

5.4.1.1 Stakeholders

EflaVefgatt is used as a component in other systems for various projects and customers. Its main purpose is to act as a part of integrated systems that display data from PLCs in web applications. It is also used to some extent for small internal projects at EFLA, for example to log the number of employees that eat lunch in the cafeteria each day. EflaVefgatt can be considered to have three groups of stakeholders. Namely:

1. Developers that work with the program directly. They also work directly with external customers.
2. External customers that use integrated systems with EflaVefgatt as a component.
3. Employees at EFLA that use integrated systems with EflaVefgatt as a component.

Developers that work with the program directly are integral to this project since they are the only stakeholders capable of changing EflaVefgatt. No other stakeholders have a better understanding of the project so this group plays a pivotal role in the process.
Both external and internal customers use EflaVefgatt as part of black box systems and can neither work with EflaVefgatt directly nor see how it is used. Consequently, changes made to EflaVefgatt are not visible to them unless accompanied by changes to the front-end. Moreover EflaVefgatt is deployed locally for each project and so each project can have a different version of EflaVefgatt.
Whenever work is done on an integrated system that relies on EflaVefgatt, EflaVefgatt is updated but updates are not pushed to customers otherwise. For those reasons customers would need help understanding the value of changes to EflaVefgatt and would have difficulty contributing to the elicitation of its problems. Therefore the diagnosis of the problem at hand only directly involved the developers. External as well as internal customers are aware that EflaVefgatt gets updated and generally understand the value of such updates. However consenting to general improvements was the extent of participation from these groups of stakeholders.

5.4.1.2 Method Presentation

The method was presented to four individuals. Namely the head of EFLA's Databases and Web Solutions department as well as three developers working at the depart-

ment. One of theses developers is the original creator of EflaVefgatt who will be referred to as the head developer in this discussion. He works closely with the program and the other two developers work with it to a lesser extent. The department head and the head developer requested a meeting to discuss the project. The method was presented to them during this meeting. The method was presented to the other two developers in individual sessions where the elicitation also took place. No customers were involved in the process. Therefore the method was only presented to these four individuals. The three developers all took part in the later stages of the process. The department head gave his consent on using the method and took part in the elicitation of business drivers. His involvement in later stages was limited.
The four involved individuals took well to the method. They found the steps of the method easy to follow and quickly understood the main aspects and objectives. Requirement elicitation processes at EFLA are not mature and the developers were very interested in exploring formal ways to elicit software specifications.

5.4.2 Elicitation

This section describes the elicitation phase of the first iteration. The section covers both the elicitation of business drivers and the elicitation of QAS.

5.4.2.1 Business Drivers

The elicitation of business drivers took place in a joint meeting with the department head and the head developer. The method presentation took place in that same meeting. EflaVefgatt has been used as a part of integrated systems for several projects and is intended to be used for future projects as well. Both individuals agree that adapting EflaVefgatt to new projects will require maintenance. The main business goal is therefore to reduce the amount of maintenance needed, thereby decreasing development costs. EFLA's customers will benefit from this indirectly but are not directly involved in the process and were not consulted.

5.4.2.2 Quality Attribute Scenarios

This section provides a discussion of the QAS that were created for this project. The scenarios themselves are provided along with an explanation of what the terms used for the scenarios mean in this context. The scenarios are complemented with a subsection on how these scenarios were written and why they are appropriate.
Several terms are used in the scenarios whose meaning could be ambiguous. In Chapter 4, it was explained that EflaVefgatt has several services that each handle some task. When the term service is used in the scenarios it refers to one of those services or similar services that could be created in the future.
EflaVefgatt is comprised of several modules. These are logical groupings of functionality where each group contains related classes that typically have dependencies to other classes in the same group. The only module of concern here is the services module. However, it is assumed that other service modules can be added to the project and that the current service module can be split.

EflaVefgatt can act as a proxy between Front-End applications and various data sources including databases and PLCs. These data sources are referred to in the scenarios as data sources but are not limited to the currently available data sources. In addition it is assumed that an available data source has the necessary means in place to communicate with EflaVefgatt unless stated otherwise.
The system refers to EflaVefgatt itself and not to a complete system, including Front-End applications, data sources and so forth.
A direct dependency to a service is a dependency to a specific service in a module. This refers to interfacing between a module and a service in another module. If the dependency was indirect it would interface directly with the module that encompasses the service.
The scenarios were elicited through semi-structured interviews with the three individuals who are the most involved with EflaVefgatt. Since the program is not handled directly by end users the individuals who were interviewed are all software developers. The involvement of each of these developers with the program differs substantially.
First off, the least involved developer uses EflaVefgatt indirectly through a front-end application that is inconveniently also named EflaVefgatt. That is a web application that was created for internal use within EFLA. He has some understanding of the program but is rarely involved in back-end development. Therefore his input was limited. Second, there is the individual who wrote the aforementioned front-end application. On occasion he has used EflaVefgatt to communicate with data sources but does not use any other services. Finally, there is the head developer of EflaVefgatt who has used all parts of EflaVefgatt for one or more of his projects.
These developers were all interviewed individually. A simple questionnaire was used as a basis for each discussion but the interviewees were encouraged to go off course and express any ideas they came up with outside the questionnaire. The architecture of the existing system was considered during this process and used to determine what qualities were currently off and needed improving upon. The output of these discussions is summarized below.
Two main problem areas were identified by the developers. First there were connections to data sources. The current version of EflaVefgatt allows connections to three types of data sources. The developers think that more types will be wanted in the future. This requires maintenance since mechanisms for communicating with new types of data sources will need to be added. In addition services and front-end applications are often connected to some data source and doing so requires maintenance.
Scenarios were written to accommodate these needs. Second, there are the services in the project. Too much coupling in the service module was identified as the primary maintainability concern. The services are all bundled together in one module without clear boundaries. The developers want to simplify making changes to a service without affecting other system components. They also want to simplify the process of adding a new service module to the system. Scenarios were also written to accommodate these needs.
There was a consensus among the developers that the main incentives for a new architecture design are these issues and that other minor concerns should not be ad-

dressed specifically. Therefore the number of scenarios were kept low in an attempt to channel all efforts towards the desired goals.
The scenarios can all be categorized as modifiability or modularity scenarios and therefore a utility tree is not presented graphically. The tree has three levels. Maintainability is the root, modifiability and modularity its children and their children are QAS 1-5 and QAS 6-7 respectively.
Below are the scenarios created for this project. The pairs of scenarios 1 and 6 as well as 2 and 7 have the same stimulus but the difference in response measures caused them to be split for clarity. The scenarios are somewhat coarse grained and all of them only require an action to be finished within one day. The relevant parties were somewhat insistent on not narrowing this down to smaller intervals arguing that this is how they normally work. This was respected in constructing the scenarios.

Table 5.1: QAS 1

Portion of Scenario	**Value**
Source	A developer
Stimulus	has created and wants to integrate a
Artifact	new module into the
Environment	system.
Response	The integration is completed
Response Measure	within one day.

Table 5.2: QAS 2

Portion of Scenario	**Value**
Source	A developer
Stimulus	has created and wants to integrate a
Artifact	new service into the
Environment	appropriate module.
Response	The integration is completed
Response Measure	within one day.

Table 5.3: QAS 3

Portion of Scenario	**Value**
Source	A developer
Stimulus	wants to take into use a
Artifact	new REST protocol
Environment	at design time.
Response	A class to handle communications through that REST protocol
Response Measure	is completed within one week.

Table 5.4: QAS 4

Portion of Scenario	Value
Source	A developer
Stimulus	wants to connect some service to a
Artifact	data source
Environment	at design time.
Response	The service is connected to the data source
Response Measure	within one day.

Table 5.5: QAS 5

Portion of Scenario	Value
Source	A developer
Stimulus	wants to connect some front-end application to a
Artifact	data source
Environment	at design time.
Response	The application is connected to the data source
Response Measure	within one day.

Table 5.6: QAS 6

Portion of Scenario	Value
Source	A developer
Stimulus	has created and wants to integrate a
Artifact	new module into the
Environment	system.
Response	The module is integrated and has
Response Measure	no direct dependencies to services in other modules.

Table 5.7: QAS 7

Portion of Scenario	Value
Source	A developer
Stimulus	has created and wants to integrate a
Artifact	new service into
Environment	the appropriate module.
Response	The service is integrated and has
Response Measure	no dependencies to services in other modules.

5.4.3 Execution

This section describes the execution phase of the first MASSE iteration. The section covers both an analysis of the architectural approaches used and a description of the resulting architecture.

5.4.3.1 Architectural Approaches

When first setting out to solve the problems at hand the developers at EFLA were interested in the possibility of a design using the microservices pattern. This pattern is very popular in the Icelandic software community at the moment and the head developer had attended conferences where the pattern was a hot topic. Therefore the microservices architecture was the first architectural approach to be considered. Microservices were introduced in chapter 2 and there it was established that they had three main characteristics. These can be summarized as

1. They are limited in size and together group a very small number of related functionality.
2. They are modular not only at the program level but the deployment level as well.
3. Each microservice contains its own permanent storage and does not share it with other services.

The QAS that were elicited in a previous section highlight EflaVefgatt's reported lack of modularity. This is apparent since they require fast integration of both services and modules. The microservice pattern is highly modular and would therefore address this problem. However, if the three characteristics described above are examined it is clear that the microservice pattern is not directly applicable to this design problem.
The first characteristic that each service should be kept small is applicable since this results in higher modularity. The existing architecture satisfies this condition to a certain extent and it was decided that this characteristic could play a role in the final design.
A separate instance of EflaVefgatt is deployed for each project that contains it and no centralized version exists. Having a central version of EflaVefgatt is difficult to achieve and not necessarily practical either. The integrated systems containing EflaVefgatt are for security reasons often not networked with the outside world and having a centralized version of EflaVefgatt would require reconsidering that approach which could agitate some customers. Therefore, having services that can be deployed independently does not satisfy any stakeholder requirements.
Furthermore EflaVefgatt is used to access very centralized data from both databases and PLCs. The services all connect to these centralized data sources and having them contain their own data would result in lots of duplication and would require frequent updates to the data contained in each service.
To summarize the microservices pattern is not directly applicable to this system. However, the ideas of splitting a program into services and keeping each one small can be used to achieve the goals at hand. The existing program is very service ori-

ented in nature. That coupled with these ideas, led the participants to consider that there might be characteristics to traditional service-oriented architecture patterns that could be applied to this problem.
A traditional view of the service-oriented architecture pattern is given in [7]. The context of that pattern is a situation where multiple services are consumed by multiple clients. These services can be running on different platforms and some way for the clients to access these services without substantial knowledge of their implementations is required.
This context description fits the problem at hand well since the problem has data accessible from both databases and PLCs and clients need to access their data with limited knowledge of what happens in the background.
The solution provided by the pattern is the traditional SOA architecture where services and clients are different components. An Enterprise Service Bus (ESB) is used as a broker between all the different components.
Given how well this pattern fits to the problem at hand it was decided that the SOA architecture would play a large role in the architecture design.
The description of the maintainability problems encountered with EflaVefgatt involves there being too many dependencies between the various services as well as to the different clients. A major aspect of this being that clients have a separate connection to each service they communicate with. They also communicate with each service through a separate interface
The solution in the SOA pattern is to separate clients and services and connect them through an ESB as was described. Having the ESB in place means the clients only have one service endpoint to communicate with, resulting in a drastic reduction in dependencies. This would therefore help maintenance substantially.
In addition to patterns the split module tactic was considered. The services package is entirely contained within one module. Despite being composed of multiple services it can be considered a cohesive module to some extent as will be explained in more detail in the description of the resulting architecture.

5.4.3.2 Architecture

The analysis of the architectural approaches unveiled that the SOA pattern would be a good fit for the problem at hand. Therefore, the design of the new architecture was led by the SOA pattern. The end result is depicted in Figure 5.1.
The biggest change from the existing architecture is that there is an ESB in place that handles all communications between the data sources, services and the clients. The ESB has a module called the broker module. This module contains classes that each implement at least one of the interfaces to the service modules, the data sources or the clients. All communications between the various entities must go through the broker. The broker should contain as little functionality as possible and be limited to the logic needed to help the various entities communicate.
There are two new modules namely the databases and PLC modules. These are modules that communicate with data sources directly and that communicate with the services through the broker. These are resource layers that contain no business logic and only serve to pass messages between data sources and the broker.

The split module tactic was applied to the service module. The service module is composed of several services with lots of direct dependencies between them. These services are not independently deployable. Consequently, it was decided that splitting the module would be beneficial. Furthermore a constraint was placed that services would not be allowed to directly depend on services in other modules.
The services module was subsequently split into three modules namely: the data fetching module, the data logging module and the action performing module. The architecture requires that there be no direct dependencies between the entities that communicate with the broker. It should be clarified that the modules as wholes constitute entities and a service does not. Therefore, having direct dependencies between services within modules does not break that rule. The existing system has dependencies across these modules so refactoring is needed to split this module. It is assumed that all services contained in a specific module communicate with the broker through the same interface. This is one reason to split the module. It was decided to allow direct dependencies within modules, as a way to make it more realistic to implement this new design. There was a certain fear that too much refactoring would be required in order to remove all dependencies and that only going halfway was more likely to be achievable.
The data fetching and data logging modules group services that, fetch data from databases and PLCs, and services that, log data to databases from PLCs, respectively. The new architecture requires some changes be made to both these modules. The existing system has the services communicating directly with the data sources but placing the broker in between requires that the resource functionality be moved to separate components in the database and PLC modules. The business logic should however remain in the service modules and the broker is a layer in between these components. The resource logic in the databases and PLC modules should be limited to what is absolutely necessary to communicate with the data sources. Doing it this way separates concerns and maintains the ESB as the heart of the system.
This design has six interfaces that are implemented in the broker module. These interfaces bridge between the broker and the three service modules, two data source modules and the clients. It is assumed that each service in the service modules communicates directly with the broker. The same can be said for all classes in the data source modules. However, there are separate interfaces for each module. Then there is a separate interface for communications with front-end clients.
The service modules as well as the data source modules have several implemented interfaces that are then consumed by the broker module. The data source modules are not composed of multiple distinct services but are rather each contained within one class. This results in a single interface being implemented by each and both are consumed by the broker. For the services modules there is one interface implemented by each provider service. All of these interfaces are consumed by the broker. It is assumed that over the lifetime of this software product the number of interfaces consumed by the broker will increase with a growing number of services.

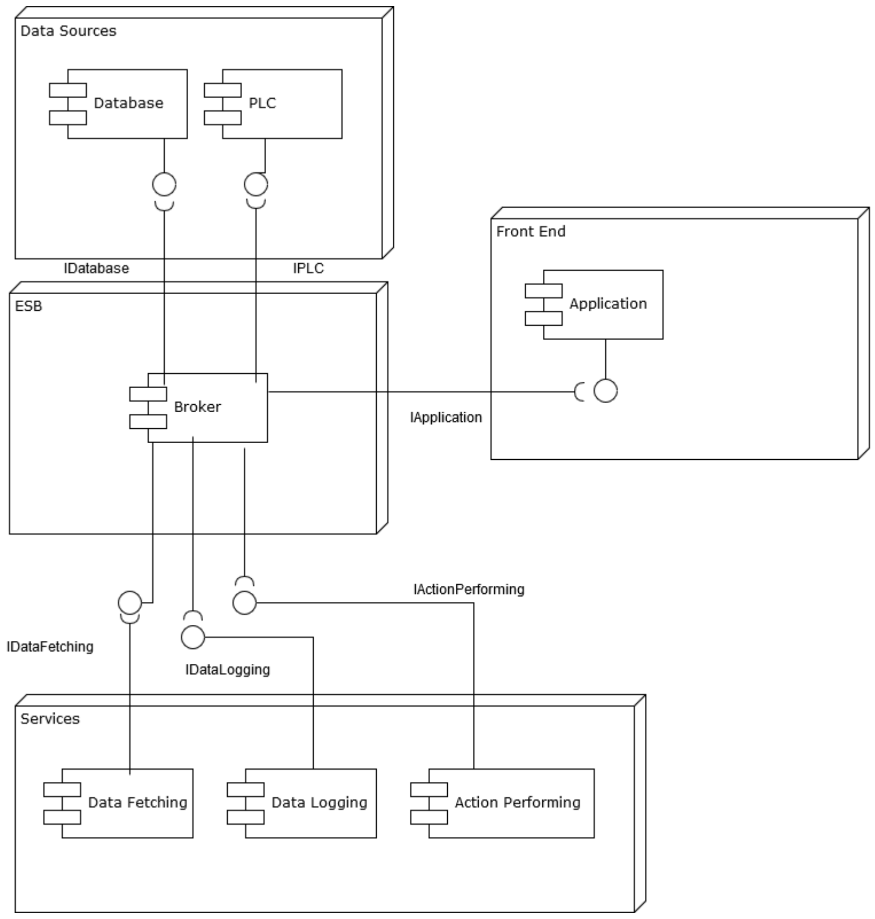

Figure 5.1: Architecture from Iteration 1.

5.5 Evaluation

The research framework that this work is based in requires that there be an evaluation of the implementation that is the output of the research. This is integral since the next iteration relies on this information to make changes to the design and perhaps the implementation process. This section will define the evaluation process used for the project and provide the findings.

5.5.1 Evaluation Process

Before moving on to evaluating the design, a description of the evaluation process that was defined will be provided. A prototype of the architecture produced during implementation was used to evaluate the architecture. Since the method that was designed produces an architecture as its output the quality of the method can be judged by the quality of such architectures. The quality of the architecture can then be determined by the quality of its implementation, in this case a prototype.
The prototype that was created had the design of the architecture from the implementation section and was written in JavaEE. This prototype was a conceptual prototype rather than a working prototype. What this means is that the classes

and interfaces that were part of the architecture design were created. However they were not implemented. The reason for this is that the system serves a lot of different clients which require different resources out of the ones available. I was tasked with creating the prototype. I could not get access to all of these clients and even if that had been the case, the work involved in determining which services were being provided to each client and what the expected outcome was, would have been extensive and out of the scope of this project. Since there was very little documentation and no unit tests available, it was decided that implementing a working prototype could not be achieved and the emphasis was therefore placed on the structure.
The evaluation of the prototype aimed to determine whether it was in fact more maintainable than the original system design. Given that the prototype was not a working prototype it was decided that a qualitative approach to evaluation was better suited than a quantitative one.
The approach taken was to create a questionnaire to be answered. In [17] a guide to creating an effective questionnaire is provided. This guide describes nine steps to creating a questionnaire. These steps are:

1. Define your research question and study population.
2. Decide how the questionnaire will be administered.
3. Formulate your questions.
4. Formulate the responses.
5. Design the layout.
6. Pre-pilot the questions and layout.
7. Pilot study–test validity, reliability, acceptability.
8. Design your coding scheme.
9. Print questionnaire.

These steps were used to guide the process of writing the questionnaire but the process was adjusted when appropriate. Some of the steps were not appropriate since they assumed that a large sample was being considered and that responses were not guaranteed. In this case however, the sample only consisted of one respondent and a response was guaranteed.
In [17] closed questions are defined as questions that have a finite number of allowed answers. This is opposed to open questions where the respondent can write his own answers. A closed question can have Yes/No answers, multiple choices or possibly have the respondents order the answers based on applicability. The questions can ask the respondent to answer how much he agrees to a statement by answering with: strongly disagree, disagree, neutral, agree, strongly agree, or something compatible. Additionally it is possible to have the respondent give analogue answers by marking where on a line between two opposite answers his opinions lie.
For this research it was decided to go with closed questions. The outcome was a questionnaire of 17 questions. These questions were all structured in a similar way. Each question targeted a specific architectural decision and asked about the impact said decision would have on a specific QAS. Furthermore each question asked the respondent to compare the new architecture design to the existing design. These questions were limited to architectural decisions and QAS pairings where the decision

could affect the scenario significantly. The 17 questions do therefore not compare every decision to every scenario.
These questions are therefore somewhat in line with closed questions where the respondent is asked to what degree he agrees with a statement. For this case a total of three options was decided on. These three options were that a decision could have a significant positive impact, no significant impact or a significant negative impact. A total of three options was decided on since a respondent to this questionnaire would not have enough information to judge the impact of an architectural decision with more accuracy. The alternatives were the same for each question and their exact wording was:

1. The change is a significant improvement. +1p
2. There is no significant difference. 0p
3. The design was significantly better before the change. -1p

These answers compare the new design to the original one. The answers each had a score as indicated at the end of each question. They have the unit of p which is simply short for point, but this unit was used to remove ambiguity between the scores and other numbers in this discussion. The sum of the scores should be calculated and the more positive the value the more positive the result is. The purpose of scoring the questions is to provide a metric that can be used to interpret the results.

5.5.2 Results

The questionnaire was answered by the head developer. Other people were involved in this process but the head developers´ knowledge and involvement in the project far exceeded everyone else's. Furthermore, other individuals were not available to obtain the information required to give well informed answers to the questionnaire. Therefore it was decided that the best result would be obtained with the head developer as the sole respondent. This is in line with the fact that the elicitation method is intended for small scale operations. As the respondent was not directly involved in the creation of the new architecture itself he was in a position to give an objective view.
These questions were presented to the respondent in a shared online document. The document was opened to the respondent for editing once the questionnaire was complete and closed again once he had answered it.
Before the questionnaire was opened to the respondent, the details of the new architecture design were presented to him during a meeting. The respondent was presented with relevant documentation that was made available to him so that he could review it whilst responding. This was augmented with a presentation explaining the design and the purpose of each design decision.
The entire questionnaire along with response scores is provided in the appendix. The final score obtained was +8p. The range of possible results was from -17p to +17p. Therefore a score of +8p is a positive result. Some questions had negative scores but this was to be expected since some architectural decisions required trade-offs and only one iteration was completed.

6
Iteration 2

The purpose of this chapter is to report on the execution of the second iteration of the Wieringa design cycle. This iteration was somewhat shorter since some parts were covered adequately in the first iteration. The most notable being that no problem investigation is needed, since the research framework does not require it for this iteration.

6.1 Solution Design

In the second iteration the solution design process was a refactoring process and the design artifact of the previous iteration was improved. This section includes discussions on a revised version of the artifact as well as the arguments for why this version is an improvement.

6.1.1 Rationale for Changes to the Method

During the first iteration a first version of the MASSE method was coined. During this process and when the next iteration was considered several ideas on how to improve the method came up. Iterations that succeed the first iteration are frequently refereed to collectively as later iterations in this text. In addition, the iteration that succeeded the iteration under discussion at each point is frequently referred to as the previous version in this text.
It immediately became apparent that having a fixed version of the method for every iteration would be counter productive and that variations of the method for later iterations should be described. The first variation regards the Presentation phase. When executing successive iterations with a short time in between iterations this phase plays a smaller role since relevant parties should already be familiar with it. This phase can therefore be kept short and should probably only need to explain the difference in later iterations to the first one.
The second variation regards the elicitation of business drivers during the Elicitation phase. The business drivers should be unaffected by earlier iterations and therefore do not need to be revisited and that step can be skipped. The original version of the method implied that this step would play a smaller role in later iterations. The difference here is that a firmer stance is taken and the step does not need to be revisited at all under normal circumstances.
The third variation is that the QAS should be revisited. They should not be rewritten from scratch but thought should be given to whether they may need to be edited

or rewritten to some extent.
The fourth variation is that it should be explicitly noted that the architecture should not be rewritten from scratch during each iteration but rather be a redesign of the output of the previous iteration.
As well as adding variations to later iterations the step of generating a quality attribute utility tree needs a minor modification. Since the Quality attribute utility tree only has a branch for maintainability calling it a tree can be confusing. It would be more accurate to say that the maintainability scenarios should be elicited and then categorized based on which sub-attribute they belong to.
It was pointed out that the description of step one in the presentation phase implied that a lot of parties would be involved in the process. This could be repelling for projects that are smaller in size. Due to this the method description needs to clarify that some of these parties may be involved but it is not necessary to involve nearly all of them.
Furthermore, the execution phase was found to be quite extensive and that breaking the steps down to more steps and possibly adding more steps would be an improvement. An idea for that was to add a final step after the architecture has been created where the architecture is matched up against the scenarios. This would be analytic reasoning for how this new architecture fulfills the scenarios.
The method of the previous iteration did not specify what constitutes a small piece of software. It had been assumed that a small piece of software would be a project having less than 100000 LOC. This was to be explicitly stated in the method description.
Finally after the iteration was complete, it was considered that the approach to design architectures entirely from maintainability scenarios without considering other quality attributes could be risky. The solution proposed for this was that important scenarios for other quality attributes would be elicited as well. These scenarios would not guide the design of the architecture but be used for a sensitivity analysis once the architecture had been completed. This was suggested after the process was completed and was not tested but is all the same included in the method description.

6.1.2 Method Revision

This section provides an updated version of the MASSE method that was created in this process. A complete description of the method is given rather than just listing the changes from iteration 1. This may seem somewhat repetitive but it is necessary so that a complete description of the revised method is made available.
The MASSE method is an elicitation method suitable for relatively small pieces of software. Generally this will be software with 100000 LOC or less. The MASSE method consists of three phases namely: Presentation, Elicitation and Execution. Each of these phases is comprised of one or more steps. The method is intended to be used iteratively. A succession of iterations is executed until a satisfactory result is obtained. The first iteration differs from later iterations in some ways. These differences are described later when the steps are listed.
It is assumed that the architecture will be revised later in the life of the software product. At that point the process is executed from the first iteration until a sat-

isfactory result is obtained once again. Doing this iteratively over the life of the software product contributes to the product preserving high maintainability. The phases and their steps can be summarized as follows:

Presentation

1. Present the method. The method is described to the involved stakeholders (typically customer representatives, the architect or architecture team, user representatives, maintainers, administrators, managers, testers, integrators, etc.). In later iterations this step is less important and its main purpose is to describe parts of the process that are skipped or different from the first iteration.

Elicitation

2. Elicitation of business drivers. Information is gathered from the involved stakeholders about their expectations for the system. This step is not necessary for later iterations and should be skipped under normal circumstances.
3. Elicit maintainability quality attribute scenarios. These scenarios should be grouped based on their sub-attributes.

Execution

4. Propose architectural approaches. With the quality attribute grouping from step 3 in hand architectural approaches that help satisfy these scenarios are identified.
5. Propose architecture. The proposed architectural approaches from step 4 are now used to create an architecture for the system in question. In later iterations the result of the previous iteration will be redesigned as needed.
6. Validate architecture. The architecture is measured up against the scenarios through analytic reasoning.

The phases will now be described in more detail.

6.1.2.1 Presentation

During this phase the method is described to the involved stakeholders and anyone participating in the process. These parties can be from any of the groups described in the description of step one depending on the situation. However, it is advised that the number of people involved be kept low to prevent overhead from rising. It is necessary though, to have someone knowledgeable on the stakeholder goals participate as well as someone that works with the software.

The purpose of this phase is to make sure that all the participants understand the process. More specifically, that they understand how the process works, what the purpose of different steps is, what is to be gained from its execution and what their respective roles are in the process. This helps to ensure efficient participation from each party involved in the process. This step is less important in later iterations at which point its main purpose is to explain which parts of the process are skipped or changed from the first iteration. When performing multiple successive iterations this step will gradually become unnecessary.

6.1.2.2 Elicitation

During this phase QAS for the system being designed are elicited. First the project business drivers are elicited from the involved stakeholders. This is done to ensure that the overall project goals are understood and can guide the execution of the following steps. The purpose of this step is to have the goals from a business perspective elicited. This is done through interviews with relevant stakeholders. During later iterations this step can be skipped. The reasons for this are that the business drivers should not be affected in any way by previous iterations. Furthermore, since the iterations should be executed in close succession the business drivers are unlikely to have changed over time. Therefore, in later iterations this step should be skipped entirely under normal circumstances. Following the elicitation of the business drivers the maintainability QAS are written. This is done by working closely with the stakeholders, in particular the technical staff that will implement the architecture. General scenarios like the one in table A.1 in the appendix can be used as a guideline. Once the maintainability scenarios have been written other quality attributes should be considered. Quality attribute scenarios should be written that address only the most important concerns that do not involve maintainability. When a consensus is reached on the QAS and a maintainability quality attribute grouping is ready, the design, or redesign in later iterations, of the architecture can begin.

6.1.2.3 Execution

During this phase the architecture itself is created and validated in three steps. First appropriate architectural approaches are identified. The purpose of this step is to structure the process. The approaches can include tactics, patterns or other established architectural approaches, that can help satisfy the scenarios. Then when the approaches have been identified, they are used to systematically create an architecture for the system at hand. In later iterations this will be a redesign of the output of the previous iteration rather than a new design from scratch. In some cases the resulting design will differ from the previous one enough to be called a new design. However the architecture from the previous iteration should always be the starting point in later iterations. The final step is to measure the architecture up against the maintainability scenarios. Doing this once the architecture has been created helps ensure that the involved parties did not lose sight of their original goals when designing the architecture. If this is found to be the case the previous step can be revisited. Once the architecture has been measured up against the maintainability scenarios the other scenarios are considered. This is done to try and spot whether any other quality attributes are at risk of being greatly affected negatively. If the impact to other quality attributes is considered reasonable the iteration is complete otherwise, the previous step should be revisited.

6.2 Design Validation

The design validation from the first section still holds for what was discussed there. The solution design text for this iteration provides arguments for the changes that were made to the artifact. Given all this it was not deemed necessary to go through an entire validation at this stage.

6.3 Implementation

Since both the research method and the design artifact are iterative processes a discussion to remove ambiguity when speaking of iterations is warranted. The research method goes through iterations to produce and improve upon a design artifact. The design artifact in this case is an iterative method that produces another design artifact, namely a software architecture. This means there is inevitably some nesting of iterations that can cause confusion.
The execution of the research method involves running one or more iterations of the method it produces. In the past research method iteration a single iteration of the MASSE method was executed. Despite the first MASSE method iteration differing somewhat from later iterations it was decided that the execution of the second MASSE method iteration would take place as part of the second research method iteration. This means that a second iteration of the first version of the MASSE method was not executed nor a first iteration of the second version.
The reasons for this are that after the first method iteration had been executed it was apparent that the method could be improved upon. These improvements mainly concerned listing in what ways later iterations should differ from the first one.
When being implemented for the first time it was found that some aspects needed to be different. These differences were then channelled into the improved version of the MASSE method. The first iteration description of the new version of the MASSE method remained comparable to the original though.
From this it was deducted that it was not necessary to repeat the first MASSE method iteration in the second research method iteration. Moreover it did not seem logical to have a second MASSE method iteration as part of the first research method iteration. This was because potential improvement areas had already been identified in the first version of the method and it seemed logical to make appropriate adjustments before moving on.
Therefore, this section provides a discussion on the execution of the second method iteration using the second version of the MASSE method. In this section method iterations are referred to simply as iterations and should not be taken to mean research methods unless stated otherwise. Additionally the method should be taken to mean the second version of the method unless stated otherwise.

6.3.1 Presentation

As prescribed by the method this phase was quite short for the second iteration. The differences were described informally to the head developer but no meetings were held. There was less involvement from other parties in this phase than in the previous iteration as a result of the relative triviality of it. The head developer along with the author of this work were the main contributors during this phase and the remainder of the iteration. Other parties were consulted when needed but were generally not involved.

6.3.2 Elicitation

As was prescribed by the method this phase was also somewhat shorter than in the previous iteration. There were no special circumstances that required revisiting the business drivers. Therefore that step was skipped entirely.
The revised version of the method emphasizes that the QAS should not be assumed to be immutable. Therefore the scenarios were revisited.
This was done in two parts. First the existing scenarios were examined and a discussion was held on whether they were still applicable and whether any changes should be made to them. Secondly it was considered whether any new requirements had been discovered that would need to be elicited as scenarios.
The results were that no alterations were deemed necessary and no new scenarios were added either. Given that the program under investigation is of intermediate size this can be expected in a lot of cases. Even though it did not result in any changes this step enlightened the participants and asserted that the elicitation was successful to begin with.

6.3.3 Execution

The execution phase for the second iteration includes discussions of architectural approaches and the new architecture design. Furthermore, the added step of architecture validation is discussed.

6.3.3.1 Architectural Approaches

The architectural approaches that were identified were mostly as responses to observed improvement areas, in the architecture from the first iteration, that needed to be addressed. A discussion on that design revealed that the splitting of the service module from the previous iteration was considered incomplete. What was meant by this was that even though the appropriate services had been grouped according to function and dependencies between different groups removed, the module was still to some extent one cohesive unit, and the split had minimal impact. To change this, it was suggested that some way of making each module more unique would be beneficial. Adding proxies between the broker and the modules was suggested as a way to do this.
It was also discussed that the difficulties in connecting EflaVefgatt to front-end applications had not been adequately addressed. An idea to address this was to implement a strict versioning system. This would mean that an unchanging interface would be between broker and clients, referred to as a contract. This would mean that changes could be made to specific versions of the systems without potential impacts on all other integrated projects having to be considered.

6.3.3.2 Architecture

The new architecture was a redesign of the architecture from the previous iteration as was prescribed in the method. The previous architecture design remained the same for the most part but two notable changes were made. A full description of the new architecture is not provided in this section. Instead the two major changes to the architecture are described. These changes correlate with the approaches listed in the previous step.

The first change to the architecture was that proxies were added between the broker and each of the service modules as depicted in Figure 6.1. After this change the service modules can each be considered a separate entity functionally, as opposed to being only conceptual groupings of services. Another result of this change is that the broker now only consumes one interface from each proxy rather than a separate instance from each providing service. Each proxy then only needs to consume the interfaces of the services contained in its respective module. The broker is significantly less bulky as a result. This is illustrated further in 6.2 showing an example proxy. There it can be seen that the proxy has one interface with the broker but has multiple interfaces with different services. Prior to this iteration these services interfaced directly with the broker.

The second change was that a versioning system was put in place. The versions would be numbered in the format "x.y". The x would be the major version and change whenever a change is made to the contract. The y would only change for significant changes to the program that do not change the contract. When updating front-end applications no change should be required unless the contract has been changed. Otherwise, an update would simply require a new version being connected to it.

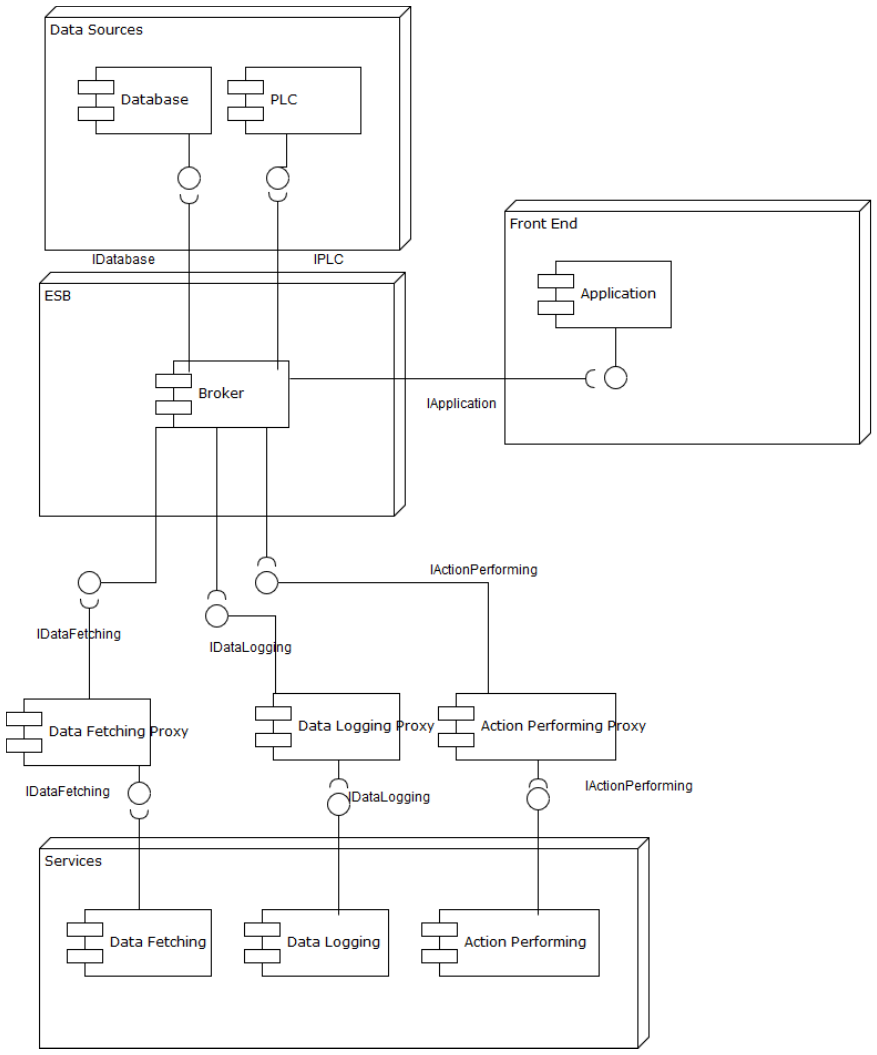

Figure 6.1: Architecture from Iteration 2.

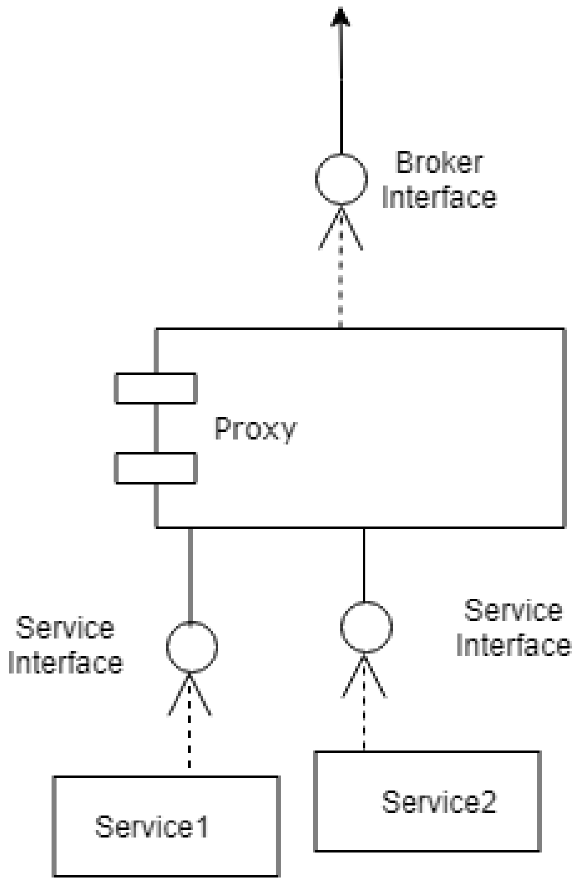

Figure 6.2: Detailed view of an example proxy.

6.3.3.3 Architecture Validation

Before the redesign took place there were two major issues. First, there was the need to fully execute the splitting of the modules. Second, there was the opinion that connecting front-end applications was still quite difficult.
The second problem is well described by QAS5 on connecting the program to a front-end application. The first problem relates to all the scenarios except QAS5. The reasoning for this is that these are the QAS that the split module tactic was aimed at satisfying in the first place.
The new architecture design took two actions targeted at these two issues. First, it tackled the problem of splitting the module by adding unique characteristics to each module. Namely by adding proxies between the broker and services modules. This change has a clear purpose and solves the splitting problem. Second, the problem of connecting the program to front-end applications was tackled by adding a versioning system. This change also has a clear purpose and solves the front-end connection problem.
To summarize, the problems encountered were solved by the architecture and the decisions made had clear purposes. They measure up well against their respective scenarios, and the analysis consequently indicates that the execution was successful.

6.4 Evaluation

The evaluation for the second iteration used the same evaluation process as the first iteration. The questions asked in this iteration were only related to the architectural decisions that were new to this iteration and the answers from the first iteration were taken to still hold for previous decisions. As before the head developer was the respondent. The entire questionnaire along with scores can be found in the appendix.
Given that this iteration had 7 questions, the range of possible total scores ranged from -7p to +7p. The total score for this iteration was +7p. The final score was therefore the highest obtainable one. There was a great deal of satisfaction at the outcome of iteration 2 so this result was not surprising.
At this point a total score for the whole process can be calculated. This can be done by summing the scores from both iterations. The scores for the split module decisions from iteration 1 are removed since they do not apply to the final version. This gives a total of 20 questions giving a point range from -20p to +20p. The final score calculated this way was +14p.
This high total score is a very positive result and indicates that the MASSE method was highly successful. This indicates that the design artifact in question is of high quality.

7
Discussion

The purpose of this study was to propose a method for eliciting architectures for small software systems requiring high maintainability. RQ1 asked how architectures could be elicited for small systems requiring high maintainability. RQ2 then asked whether QAS could be used to elicit architectures for such systems. Therefore, a method, named the MASSE method, that used QAS to elicit architectures, was proposed and iteratively improved.
The MASSE method, being a possible solution to the problem in RQ1 could also provide an answer to the question asked in RQ2. Two iterations where the method was used were executed. The first iteration resulted in a version of an architecture elicitation method. This first version described three phases, each consisting of one or more steps. These phases were to be executed iteratively until a satisfactory result was obtained. Later in the life of the software product it was expected that the phases would be revisited, whenever the architecture needed revising. This method was then put to use and an architecture was created. This architecture was then evaluated.
The first iteration yielded positive results and the method seemed to be effective. The resulting architecture was compared with the architecture it was meant to replace, through the collection of qualitative data. Subsequently it was deemed a significant improvement. The first iteration also revealed some possible improvement areas for the method which fed into the second iteration.
During the second iteration the method was improved. The biggest change being, that it had become apparent during the first iteration, that the first method iteration was unique. This led to the first iteration being defined separately. The results of this iteration were also positive with the evaluation of the architecture resulting in a perfect score.
RQ2 asked whether QAS could provide a way to elicit architectures, specifically for small systems with requirements for high maintainability. A method was proposed that successfully elicited an architecture for such a system. This indicates an affirmative answer to RQ2.
RQ1 asked how architectures, for systems like the one described in RQ2, could be elicited. Since the answer to RQ2 was in the affirmative, QAS provide one solution to RQ1.
The positive results of this research indicate that this method could be a good alternative to traditional elicitation methods in some cases. It is recommended that this method only be applied to systems with the specified requirements. Applying the method to, larger systems or systems where another quality attribute is important, is not recommended. The reasons being that it was not designed for those contexts

nor tested in them.
In this work an existing piece of software was used as a basis for the process. This is in accordance with the design of the method since an existing architecture should be revisited during the life of the software. Then being redesigned in order to preserve maintainability. The method is also intended to work when constructing new software from scratch. This was not tested during this work and the execution relied on the existing architecture to determine what measures needed to be taken. If an architecture was being created from scratch it would be less clear where the quality problems lie which could complicate the process.
The evaluation of the method was entirely qualitative in nature. Furthermore, the collection of qualitative data was limited to a survey answered by one individual. The method is designed to be used over the lifetime of a software product and the architecture design should be revisited later in the life of a software product. Given the time limitations for this research, the method did not create an architecture from scratch, reapplying the method later in the life of the same software product. The method was applied to an existing software product which is somewhat like applying it later in the life of a software product. This software product was not designed using the method originally though.

7.1 Future Work

Future research on this topic would therefore include applying the method fully. This would entail creating an architecture for an entire system using the method, subsequently reapplying the method later in the life of the same product. This should be done several times for the most accurate results. Performing more design science iterations for a case like the one at EFLA would be useful to improve the method. To achieve this it would particularly useful to study a situation where quantitative data was more easily obtainable. This would give much better information on the effectiveness of changes.
The idea of a sensitivity analysis, of other quality attribute scenarios than maintainability, was floated at the end of the process. This was not performed as part of the implementation steps of the iterations. Future work would include executing the method whilst including such an analysis. This could prevent the method from resulting in unusable architectures due to unconsidered risks.
The method has additionally not been empirically compared to other methods for effectiveness. It would be valuable to collect quantitative data in addition to qualitative data on architectures produced by the MASSE method, comparing that to the prevalent methods of the industry. Comparing their effectiveness for producing both large scale and small scale architectures could help show the need for the MASSE method.

8 Conclusion

This project set out to solve a design problem and at the same time give an answer to a research question. The topic of both related to defining an architectural elicitation method suitable for a special situation. Namely a situation where a small software system with high maintainability is required.
Such a method was proposed. This method used QAS to elicit the architectures iteratively. This method was applied in industry where a system of the described type was needed. This successfully produced an architecture for the system. The method was revised and subsequently improved over the course of two iterations. The results of the evaluations for each iteration were very positive and indicated that the method had been effective.
At the start there were three main objectives described for this research. Namely proposing a method, evaluating that method and producing a software architecture as a byproduct. All of these objectives were achieved. A method was proposed, put to use and subsequently evaluated. This produced an architecture as a byproduct as was intended.
Despite the limitations of the accuracy of the evaluation of this method, a new method to elicit architectures has been created. There is some uncertainty as to how generalizeable the method is given limitations to the research data. However the data available suggests it is a viable alternative to existing methods of eliciting architectures, when applied in the right context.

References

[1] P. Clements, R. Kazman, and M. Klein, *Evaluating software architectures: Methods and case studies*, English. Boston, [Mass.]: Addison-Wesley, 2002, ISBN: 9780201704822;020170482X;

[2] *Iso - international organization for standardization*, Jun. 2016. [Online]. Available: `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35733`.

[3] P. Bhatt, W. K, G. Shroff, and A. Misra, "Influencing factors in outsourced software maintenance", English, *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–6, 2006.

[4] R. Kazman, M. Klein, and P. Clements, "Atam: Method for architecture evaluation", English, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2000-TR-004, 2000. [Online]. Available: `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=5177`.

[5] P. Bengtsson, J. Bosch, I. för programvaruteknik och datavetenskap, and B. T. Högskola, "Scenario-based software architecture reengineering", English, 1998, pp. 308–317, ISBN: 1085-9098.

[6] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet, "Architecture-level modifiability analysis (alma)", English, *The Journal of Systems Software*, vol. 69, no. 1, pp. 129–147, 2004.

[7] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*, English, 3. Upper Saddle River, N.J: Addison-Wesley, 2012, pp. 63, 119–120, ISBN: 9780321815736;0321815734;

[8] R. Kazman, M. Gagliardi, and W. Wood, "Scaling up software architecture analysis", English, *Journal of Systems and Software*, vol. 85, no. 7, pp. 1511–1519, 2012;2011;

[9] C. Carneiro Jr., T. Schmelmer, S. (service), B. (.-b. collection), and S. (.-b. collection), *Microservices from day one: Build robust and scalable software from the start*, English. Berkeley, CA: Apress, 2016.

[10] *Attribute-driven design method*, Mar. 2017. [Online]. Available: `http://www.sei.cmu.edu/architecture/tools/define/add.cfm`.

[11] P. Bengtsson and J. Bosch, "Architecture level prediction of software maintenance", English, 1999, pp. 139–147, ISBN: 0769500900;9780769500904;

[12] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research", English, *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[13] R. Wieringa, "Design science as nested problem solving", English, ACM, 2009, pp. 1–12, ISBN: 9781605584089;1605584088;

[14] J. E. Bardram, H. B. Christensen, and K. M. Hansen, "Architectural prototyping: An approach for grounding architectural design and learning", English, IEEE, 2004, pp. 15–24, ISBN: 076952172X;9780769521725;

[15] *Kepserverex.* [Online]. Available: `https://www.kepware.com/en-us/products/kepserverex/`.

[16] *Iso - international organization for standardization*, Sep. 2012. [Online]. Available: `http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749`.

[17] A. Williams, "How to write and analyse a questionnaire", English, *Journal of orthodontics*, vol. 30, no. 3, p. 245, 2003.

A
Appendix

Table A.1: General Modifiability Scenarios

Portion of Scenario	**Possible Values**
Source	End user, developer, system administrator
Stimulus	A directive to add/delete/modify functionality, or change a quality attribute, capacity, or technology
Artifact	Artifacts Code, data, interfaces, components, resources, configurations, . . .
Environment	Runtime, compile time, build time, initiation time, design time
Response	One or more of the following: • Make modification • Test modification • Deploy modification
Response Measure	Cost in terms of the following: • Number, size, complexity of affected artifacts • Effort • Calendar time • Money (direct outlay or opportunity cost) • Extent to which this modification affects other functions or quality attributes • New defects introduced

Table A.2: Iteration 1 Questionnaire and Results

1. In your opinion how does the addition of a broker impact the integration of a module into the system as described in QAS1?
   Score: -1p
2. In your opinion how does the removal of dependencies between services in different modules impact the integration of a module into the system as described in QAS1?
   Score: +1p
3. In your opinion how does the addition of a broker impact the integration of a service into the system as described in QAS2?
   Score: +1p

4. In your opinion how does the removal of direct dependencies between services in different modules impact the integration of a service into the system as described in QAS2?
   Score: 0p
5. In your opinion how does the splitting of the service module and grouping related services in smaller modules impact the integration of a service into the system as described in QAS2?
   Score: +1p
6. In your opinion how does the addition of a broker impact taking a new REST protocol into use as described in QAS3?
   Score: +1p
7. In your opinion how does grouping data fetching services into their own module impact taking a new REST protocol into use as described in QAS3?
   Score: 0p
8. In your opinion how does the addition of a broker impact connecting a service to a data source as described in QAS4?
   Score: +1p
9. In your opinion how does the splitting of the service module and grouping related services in smaller modules impact connecting a service to a data source as described in QAS4?
   Score: 0p
10. In your opinion how does the addition of a broker impact connecting a front-end application to a data source as described in QAS5?
    Score: +1p
11. In your opinion how does grouping data connection services into a separate module impact connecting a front-end application to a data source as described in QAS5?
    Score: +1p
12. In your opinion how does the addition of a broker the integration of a module without direct dependencies to other modules into the system as described in QAS6?
    Score: +1p
13. In your opinion how does the removal of existing dependencies between services in different modules impact the integration of a module without direct dependencies to other modules into the system as described in QAS6?
    Score: +1p
14. In your opinion how does the splitting of the existing service module into several different modules impact the integration of a module without direct dependencies to other modules into the system as described in QAS6?
    Score: -1p
15. In your opinion how does the addition of a broker impact the integration of a service into a module without direct dependencies to services in other modules as described in QAS7?
    Score: +1p
16. In your opinion how does the splitting of the service module and grouping related services in smaller modules impact the integration of a service into a

module without direct dependencies to services in other modules as described in QAS7?
Score: -1p

17. In your opinion how does removal of direct dependencies between services in different modules impact the integration of a service into a module without direct dependencies to services in other modules as described in QAS7?
Score: +1p

Table A.3: Iteration 2 Questionnaire and Results

1. In your opinion how does how does the splitting of the service module and grouping related services in smaller modules that communicate with the broker through proxies impact the integration of a module into the system as described in QAS1?
Score: +1p
2. In your opinion how does how does the splitting of the service module and grouping related services in smaller modules that communicate with the broker through proxies impact the integration of a service into the system as described in QAS2?
Score: +1p
3. In your opinion how does how does the splitting of the service module and grouping related services in smaller modules that communicate with the broker through proxies impact taking a new REST protocol into use as described in QAS3?
Score: +1p
4. In your opinion how does how does the splitting of the service module and grouping related services in smaller modules that communicate with the broker through proxies impact connecting a service to a data source as described in QAS4?
Score: +1p
5. In your opinion how does how does the splitting of the service module and grouping related services in smaller modules that communicate with the broker through proxies impact the integration of a module without direct dependencies to other modules into the system as described in QAS6?
Score: +1p
6. In your opinion how does how does the splitting of the service module and grouping related services in smaller modules that communicate with the broker through proxies impact the integration of a service into a module without direct dependencies to services in other modules as described in QAS7?
Score: +1p
7. In your opinion how does how does the addition of a versioning system involving a strict contract between the broker and front-end applications impact connecting a front-end application to a data source as described in QAS5?
Score: +1p