



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG



Assessment Methods When Choosing a Software Architecture Alternative

Software Maintainability Prediction

Master's thesis in Software Engineering and Technology

OMAR THOR OMARSSON

MASTER'S THESIS

Assessment Methods When Choosing a Software Architecture Alternative
Software Maintainability Prediction

Master's Thesis within the *Software Engineering and Technology* programme

OMAR THOR OMARSSON

SUPERVISOR, CHALMERS

Riccardo Scandariato

SUPERVISOR, FUGLAR ltd.

Helgi Einarsson

EXAMINER

Eric Knauss



CHALMERS

Department of Computer Science and Engineering

Division of Software Engineering

CHALMERS UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Gothenburg, Sweden 2017

Assessment Methods When Choosing a Software Architecture Alternative
Software Maintainability Prediction

Master's Thesis within the *Software Engineering and Technology* programme
OMAR THOR OMARSSON

© OMAR THOR OMARSSON, 2017

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Sweden
Telephone: +46 (0)31-772 1000

Chalmers Reproservice
Gothenburg, Sweden 2017

ABSTRACT

A software architecture is commonly viewed as the blueprint of a software system and consists of a set of structures that are needed to reason about the system at hand. These structures comprise the necessary software elements to implement the system and explains relations among them. Having the right software architecture when developing a software system is very desirable for any kind of a software engineer, especially when systems get large and complex. There does not exist a single software architecture that can be considered efficient for all software development scenarios as the software architecture needs to be chosen based on the given circumstance.

This study particularly investigates how to choose between software architectural alternatives solitarily based on their maintainability quality characteristic using both quantitative and qualitative approaches. The goal being to conclude which approach seems more appropriate when predicting the maintainability of a software architecture.

The software used for the investigation in the thesis is called KRIA and three software architectural alternatives were proposed for the system, and corresponding software prototypes implemented. The main instruments used to document these architectures are component and deployment diagrams along with some textual explanations. A ground truth measurement method is put forth to determine which of the three software architectural alternatives is truly the most maintainable and both the quantitative measuring and qualitative questioning evaluation method approaches are described thoroughly. The ground truth measurement method was the most extensive method as it required much preparation to ensure fairness and moreover, relies on tangible software prototypes corresponding to the architectures. It also involves quantitative time measurements. Therefore, the quantitative approach used throughout the thesis in facts relates to the ground truth method as it also requires software prototypes but was considered much easier to apply on the case. Finally, the qualitative approach was considerably easier to apply as it did not require development effort to construct software prototypes, i.e. only depended on the software architectures themselves.

After obtaining data results from the ground truth measurement method, and both the quantitative and qualitative evaluation methods from the case, it was concluded that the quantitative measuring method gives more reliable results than the qualitative questioning method for the purpose of predicting the maintainability for the case. Both approaches were however found to be useful depending on the circumstance. Therefore, even though the quantitative approach gave more promising results to predict maintainability, the qualitative questioning method proved to be useful to obtain information regarding the stakeholders interest and motivation (business drivers) for the undercoming software architectural change.

Keywords: *Software architecture, maintainability prediction, software design improvement, eliciting software architectures, quantitative and qualitative evaluation methods.*

PREFACE

This report is the result of a master's thesis research work, which was carried out as a completion of the master's program in Software Engineering and Technology at Chalmers University of Technology. The thesis was conducted in cooperation between Chalmers and Fuglar ltd., an Icelandic software firm that specialises in software solutions for pension funds. The thesis was carried out over a period of 20 weeks, between January and June 2017, and corresponds to 30 credits.

I would like to thank my academic supervisor Riccardo Scandariato for his invaluable guidance and share of knowledge throughout the study. His ambition for me as a software engineer really motivated me throughout the thesis work and I am very grateful to have had such an experienced and respected mentor.

I would also like to thank my examiner Eric Knauss for sharing his knowledge and experience in the field, for showing great interest in the study objectives and encouraging me in the work.

Furthermore, I would like to thank my supervisor at Fuglar ltd., Helgi Einarsson for always being available to give his advice and knowledge in the area and for his professional attitude concerning the academic perspectives of the thesis work.

Finally, I would like to thank the software developers and employees at Fuglar ltd. for their support. I also want to express my gratitude to Fuglar's managers for always providing me with essential material and sponsoring me throughout the thesis work.



Omar Thor Omarsson
Gothenburg, May 2017

TABLE OF CONTENTS

ABSTRACT.....	i
PREFACE.....	ii
LIST OF FIGURES	v
LIST OF TABLES	vi
LIST OF BOXES	vii
LIST OF ABBREVIATIONS.....	viii
1 INTRODUCTION	1
1.1 Problem statement	1
1.2 Purpose and aim	2
1.3 Research question.....	3
1.4 Method	3
2 THEORY	4
2.1 Software architecture – What is a software architecture?	4
2.2 Tools for documenting a software architecture	4
2.2.1 Component-and-connector view (C&C).....	4
2.2.2 Allocation view	5
2.3 Architectural patterns	6
2.4 Architectural tactics.....	6
2.5 Architectural patterns relevant for the case study	6
2.5.1 Layered pattern	7
2.5.2 Multi-tier pattern.....	7
2.5.3 Client-server pattern.....	7
2.5.4 Service-Oriented Architecture (SOA) pattern	8
2.6 Architectural tactics relevant for the case study.....	8
2.6.1 Split module	9
2.6.2 Increase semantic coherence.....	9
2.6.3 Encapsulate	9
2.6.4 Restrict dependencies.....	9
2.7 Comparing software architectural alternatives - Quality attributes	9
2.7.1 ATAM – Architecture Tradeoff Analysis Method	10
2.8 Quantitative and qualitative architecture evaluation methods	11
2.8.1 Quantitative methods	11
2.8.2 Qualitative methods	11

3	FUGLAR’S CASE	12
3.1	KRIA’s main functionality and features	12
3.2	Software architectural alternatives	13
3.2.1	KRIA’s original software architecture (A-ORIG) – Problem area	13
3.2.2	KRIA’s SOA software architectural alternative (A-REST)	16
3.2.3	KRIA’s web-based software architecture (A-WEB)	20
4	CASE STUDY APPROACH	24
4.1	Why case study?	24
4.2	Case study design	24
4.3	Preparation for data collection	25
4.3.1	Building the ground truth – Implementation time measurement	25
4.3.2	Qualitative approach	27
4.3.3	Quantitative approach	33
4.4	Collecting evidence	34
4.5	Analysis of collected data	34
4.6	Reporting	34
5	RESULTS	35
5.1	Implementation time - Ground truth measurement	35
5.2	Questionnaire - Qualitative questioning	36
5.3	LOC - Quantitative measuring	38
5.4	Interpretation of the results	41
5.5	Threats to validity	42
6	CONCLUSION	43
6.1	Discussion & conclusion	43
6.2	Future work	44
	References	46
	Appendix	49
	Appendix A – Software architectural alternatives	49
	A.1 A-ORIG	49
	A.2 A-REST	51
	A.3 A-WEB	54
	Appendix B – The ground truth measurement method	57
	Appendix C – The questionnaire for the qualitative questioning method	58

LIST OF FIGURES

Figure 1: Graphical notation of the main elements of a typical UML component diagram.....	5
Figure 2: Graphical notation of the main elements of a typical UML deployment diagram.....	6
Figure 3: Example application form from KRIA’s original software system	13
Figure 4: Main component diagram for KRIA’s original architecture (A-ORIG).	14
Figure 5: Deployment diagram for KRIA’s original architecture.	15
Figure 6: Main component diagram for KRIA’s RESTful service architectural alternative...	17
Figure 7: Deployment diagram for KRIA’s RESTful service architectural alternative.	19
Figure 8: Main component diagram for KRIA’s web-based architectural alternative	21
Figure 9: Deployment diagram for KRIA’s web-based architectural alternative.....	22
Figure 10: Example quality attribute utility tree.....	29
Figure 11: Average implementation time value (histogram).	36
Figure 12: Qualitative questioning results (histogram).	38
Figure 13: LOC percentage for each software architectural alternative.	41
Figure 14: An example application form from the original system.....	50
Figure 15: The Delphi IDE.	50
Figure 16: The web-API for the RESTful web-service.	52
Figure 17: Microsoft Visual Studio IDE – used to develop the RESTful web-service.	53
Figure 18: PL/SQL IDE – used for the database management (e.g. stored procedures).	53
Figure 19: User’s ADFS sign-on page.....	54
Figure 20: Part of an application form from the A-WEB prototype – presents a list.....	55
Figure 21: A-WEB prototype GUI – delete, modify and add new entry to a list.....	55
Figure 22: WebStorm IDE – used to develop the ReactJS/Redux website in A-WEB.	56

LIST OF TABLES

Table 1: IDEs used to implement KRIA’s original software application.	16
Table 2: IDEs used to implement KRIA’s RESTful service architectural alternative.	20
Table 3: IDEs used to implement KRIA’s web-based architectural alternative.	23
Table 4: Architectural approaches analysed according to a quality attribute scenario.	30
Table 5: Architectural risks.	30
Table 6: Sensitivity points.	31
Table 7: Trade-off points.	31
Table 8: Maintenance tasks for KRIA and their corresponding weight values.	35
Table 9: Implementation order and time measurements.	35
Table 10: Average implementation time value for each software architectural alternative.	36
Table 11: Qualitative questioning results.	38
Table 12: LOC measurement results for A-ORIG.	38
Table 13: LOC measurement results for A-REST.	39
Table 14: LOC measurement results for A-WEB.	40
Table 15: Maintenance tasks for KRIA and their corresponding weight values (detailed).	57
Table 16: Detailed maintenance task description.	57

LIST OF BOXES

Box 1: Component diagram's main elements.....	5
Box 2: Deployment diagram's main elements.....	5
Box 3: RESTful protocol constraints.....	8
Box 4: Quality attributes from the ISO/IEC 9126 standard.....	10
Box 5: ATAM steps.....	10
Box 6: CRUDL functions	12
Box 7: Case study steps	24
Box 8: Steps used for the ground truth measurement method	26

LIST OF ABBREVIATIONS

ADFS: Active directory federation service.

ALPSM: Architecture level prediction of software maintenance.

ATAM: Architecture tradeoff analysis method.

API: Application programming interface.

A-ORIG: KRIA's original architecture, see relation to Chapter 3.2.1.

A-REST: A RESTful service architectural alternative for KRIA, see relation to Chapter 3.2.2.

A-WEB: A web-based architectural alternative for KRIA, see relation to Chapter 3.2.3.

C&C: Component-and-connector software architectural view.

CRUDL: Types of software functions, i.e. create, read, update (edit), delete, list.

IDE: Integrated development environment.

IIS: Internet information service.

LOC: Lines of code.

ODAC: Oracle data access components.

OS: Operating system.

REST: Representation state transfer (a communication protocol).

SOA: Service-oriented architecture pattern.

SQL: Structured query language (database queries).

UML: Unified modelling language.

1 INTRODUCTION

Purpose and objectives

Software systems have become essential for the modern-day society and people constantly rely on them. When software systems evolve, and move from being small manageable artefacts to being complex entities, their design structure (software architecture) becomes a crucial factor for their success. However, the choice of a software architecture is a difficult decision to make as typically several quality attributes [1] need to be assessed and there can be synergies between them. Therefore, pros and cons of architectural alternatives in relation to relevant quality attributes need to be evaluated to make a good and accurate decision. A large proportion of all software systems are continuously being evolved as improvements need to be implemented and new requirements are constantly being put forth. Therefore, nowadays most software development is conducted in relation to existing systems rather than new ones. This has made the maintainability quality attribute [1, p. 10] very important to consider when evaluating and choosing a software architecture.

There are two main types of evaluation methods that can be used when making an assessment and choosing between software architectures, i.e. quantitative measuring and qualitative questioning [2, p. 640]. These two approaches are explained thoroughly in upcoming chapters but fundamentally, quantitative methods specifically rely on numerical data that is then transformed into usable statistics whilst qualitative approaches specifically rely on subjective data (that can in fact sometimes be transformed into quantitative data). In general, quantitative methods have in the past been considered more hard science [3] than qualitative methods and are therefore often considered to provide more reliable and trustworthy results. But is this necessarily the case when predicting the maintainability quality characteristic of a software system, i.e. might for that quality attribute the subjective aspect which is gained with qualitative questioning weigh more when making the assessment?

The general idea behind this study is to investigate this issue and seek the truth in this respect. Making an assessment of which type of method seems to prevail when predicting the maintainability quality characteristic of a software system requires the use of a case study. The thesis uses a case study from industry, an Icelandic software firm Fuglar Ltd., a leading firm in software solutions for Icelandic pension funds. The company has a large software system, KRIA, which was in need of a robust architectural modification and therefore a choice between software architectures had to be made.

1.1 Problem statement

When software firms and institutions have to make choices between software architectures for their software systems they often rely on data collected from a set of evaluation methods and corresponding metrics. Namely, the data collected from the evaluation methods is used to predict and make an assessment of quality characteristics that the system will fulfil in order to meet expectations.

As previously mentioned, two basic classes of evaluation methods/techniques exist, i.e. questioning and measuring. Firstly, most questioning techniques are qualitative methods and rely on subjective data such as opinions and observations from conducting interviews, scenarios, or checklists. This data can often be quantified to further reason, and generalise about the phenomena being investigated. Secondly, the measuring techniques involve

conducting quantitative measurements to generate numerical data that can be transformed into usable statistics [4], [5].

In general, both types of methods have proved to contain appropriate sets of evaluation methods but depending on the software quality attribute being evaluated, one type is usually considered more appropriate than the other [1], [6]. For example, the ISO/IEC 9126 software standard categorises software quality attributes into six characteristics: Functionality, reliability, usability, efficiency, maintainability and portability [1]. When predicting, and making an assessment e.g. on the efficiency quality attribute from the standard, quantitative measurements clearly seem more appropriate as they can be used to measure the time behaviour (involves numerical data) of a system in relation to the amount of resources being used. On the contrary, qualitative questioning seems more appropriate to predict the usability of a system as usability involves making an assessment of the capability of the software system to be understood, learned, used and to attract the user. For other quality attributes, it is not always obvious which type of evaluation methods should be used. Even for some, a set of both quantitative and qualitative methods may seem appropriate. However, it can become problematic to choose between software architectures if both types of evaluation methods are being relied on. This has already proved to be problematic when conducting research using both methods as inconsistency between qualitative and quantitative data has caused unanticipated difficulties in interpreting research findings [7].

As previously mentioned, Fuglar Ltd. is a software firm that had an architectural problem with its software system KRIA. The problem related to how monotonous the system had been programmed in the past as significant amount of its logic was contained within one programming language, i.e. Delphi (fat-client approach) [8]. Since some of KRIA's data and business logic may have to be shared with other software clients in the future, this form of programming conventions had become a rising issue that needed to be solved by implementing a new software architecture for the system. KRIA is constantly being evolved and therefore the maintainability characteristic of the system was important to consider and predict when choosing the new architecture for the system.

A set of both questioning and measuring evaluation methods may seem appropriate predictors for making an assessment of the maintainability quality attribute. For example, metrics such as code size measurements (counting lines of code - LOC) and/or code complexity measurements (McCabe cyclomatic code complexity [9]) can be used to conduct the quantitative measuring. However, these types of metrics obviously do not only require documentations of the software architectures but also require tangible software prototypes to be present. Addressing aspects such as future potentials, suitability of software environments and development tools etc. needs to be performed by qualitatively questioning the stakeholders (developers) and here the evaluation is usually only based on the architectures themselves (prototypes not needed). Discovering which type of evaluation method seems more appropriate for the purpose of predicting the maintainability quality characteristic of a software system and its software architecture was the core problem that the thesis aims to give answers to.

1.2 Purpose and aim

The purpose of the study is twofold. Firstly, as the software system KRIA was used as a case in a case study research throughout the thesis, two proposed software architectures and corresponding prototypes were implemented with the intention to solve Fuglar's aforementioned problem. The study therefore directly assists Fuglar Ltd. to make an assessment of the maintainability quality characteristic of their architectural alternatives. Secondly, making this assessment of the suitability of the two previously mentioned classes of evaluation methods

allows other software firms and institutions in a similar situation to decide which type of evaluation method to use when predicting the maintainability of their software systems. Also, it opens a possibility for other researchers with different case studies to investigate similar phenomena.

1.3 Research question

Throughout the master's thesis work the following research question was investigated:

- RQ: Which type of evaluation method (quantitative measuring vs. qualitative questioning [2, p. 640]) yields more reliable results when predicting the maintainability quality characteristic of a system and its software architecture? – Specifically, the quantitative measuring method involved using metrics to assess the code (from prototypes), and qualitative questioning focused on the suitability of software environments and development tools etc.

1.4 Method

The investigation was initialised by analysing KRIA's original software application and its corresponding software architecture. This analysis was done in order to get a good understanding of Fuglar's problem and make it possible to efficiently point out and structure two new architectural alternatives for KRIA. After defining and documenting the two proposed software architectural alternatives and the system's original software architecture, corresponding prototypes were implemented to represent a simple proof-of-concept of how the system functions for each architectural alternative.

This domain specific investigation lead to a case study that aimed to investigate two types of evaluation methods, quantitative measuring and qualitative questioning. The purpose was to answer the previously stated research question, i.e. to determine which type of evaluation method seems more reliable when predicting the maintainability quality characteristic of a software system. Both types of evaluation methods were therefore defined accurately and a ground truth measurement method was put forth to determine which of KRIA's software architectural alternatives is truly the most maintainable. Therefore, each method aimed to make an assessment on the maintainability quality characteristic of the software architectural alternatives. The ground truth method was the most superior method to predict maintainability as it required actual maintenance tasks to be implemented in the software prototypes that corresponded to the proposed architectures. It also required implementation time measurements to be undertaken and was therefore considered more related to the quantitative measuring approach rather than the qualitative questioning approach. The quantitative measuring method involved measuring the lines of code needed to construct the software prototypes and therefore also relied on tangible software prototypes to represent the software architectural alternatives. However, the qualitative questioning method simply relied on the software architectures as questions were formulated with the stakeholders. The intention was to construct a questionnaire specifically aimed to assess the maintainability quality characteristics of the three architectures.

After defining the methods, data collection was conducted by applying the evaluation methods on Fuglar's case (the three software architectural alternatives). The results from the two types of evaluation methods were compared with the ground truth measurement results. Finally, an assessment regarding the reliability of the two aforementioned classes of evaluation methods could be made and a conclusion was drawn. The method is explained further in Chapter 4.

2 THEORY

Theoretical background

This chapter contains a summary of the theoretical background relevant for the case study research being carried out. To begin with, a general definition of a software architecture is given and an explanation of how architectural patterns and tactics are used to construct software architectures. The main tools when documenting software architectures are then introduced. Next, the general idea behind specific design principles, patterns and tactics that are used to construct the architectural alternatives for KRIA in the next chapter (Chapter 3 – Fuglar’s case) are then explained. This is followed by a section explaining how software architectures can be compared by using quality attributes and metrics from the ISO/IEC 9126 software standard [1]. Finally, a thorough distinction is given between the two evaluation methods being investigated (quantitative and qualitative).

2.1 Software architecture – What is a software architecture?

When software systems evolve and become complex artefacts, they do not only possess problems related to complex algorithms and data structures. Rather, the design structure of these systems becomes a vital aspect as it specifies how entities within the system should be organised and designed. The design structure of a software system is called a software architecture and has to do with making fundamental structural choices for a software system based on common software architectural patterns. One sufficient and broad definition of a software architecture is given by L. Bass, P. Clements, and R. Kazman [5, p. 4]:

“The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.”

As one can imagine, a single software architecture does not fit for every occasion and therefore a good software architecture for one software system can be a very bad alternative for another system. Having the right software architecture is often a crucial factor of a system’s success and having the wrong one can have disastrous consequences [10].

2.2 Tools for documenting a software architecture

When documenting a software architecture, a set of elements are explained as they exist in the system by using architectural views. An architectural view helps the architect to describe the system at hand from a specific perspective. It is however a matter of opinion whether the view entails detailed enough information for the given circumstance [11].

The thesis uses two main types of views to describe the software architectural alternatives (see Chapter 3 – Fuglar’s case), i.e. C&C and allocation views. They are explained separately in the following two subsections.

2.2.1 Component-and-connector view (C&C)

The C&C view is often represented with a UML component diagram. It describes where software concerns (components, sometimes called “classifiers”) are stored in the system and

how data travels between those components. A component diagram consists of four important elements described in Box 1 [12], [13] (graphical notation is given in Figure 1).

Box 1: Component diagram's main elements.

-
- *Component*: An element of computation and data storage.
 - *Interface*: Describes the features that a component provides (usually represented as one or more operations that other components can rely on).
 - *Port*: A distinct interaction point of its component and explains how a component interacts with its environment.
 - *Connector*: An interaction mechanism among components. It is used to connect two components via a shared interface.
-

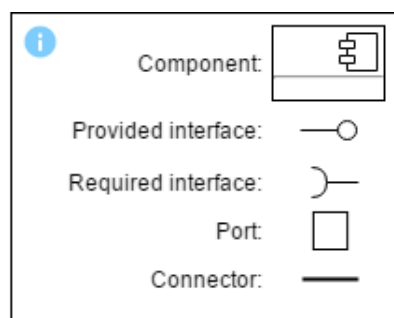


Figure 1: Graphical notation of the main elements of a typical UML component diagram.

2.2.2 Allocation view

An allocation view is often represented with a UML deployment diagram. The diagram is used to describe where the software components (from the C&C view) are physically deployed at runtime. Therefore, component and deployment diagrams are closely related and the diagrams should be consistent (i.e. all components from the component diagram should be present on the deployment diagram and vice versa) [14]. The main elements used for drawing deployment diagrams are explained in Box 2 [15], [16] (graphical notation is given in Figure 2):

Box 2: Deployment diagram's main elements.

-
- *Artefact / Component*: The components from the corresponding component diagram. These typically exist on an execution environment node which runs on a device node.
 - *Execution environment node*: A software computing resource that runs on a device node. Provides an executable software environment on the device node, for example a PC or a mobile application.
 - *Device node*: A physical computing resource of a system, for example a PC computer or a mobile phone.
 - *Association*: Used to represent the communication path between nodes. The communication path can entail a <<stereotype>> which describes the type of the connection.
-

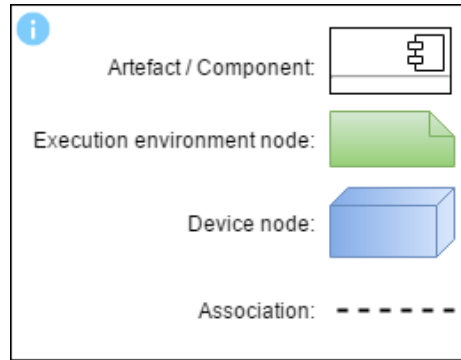


Figure 2: Graphical notation of the main elements of a typical UML deployment diagram.

2.3 Architectural patterns

Reoccurring software architectural problems in different domains have resulted in a need for creating common architectural patterns that can then be reused to solve similar future problems over different domains. These patterns capture hard-won architectural knowledge and provide a set of strategies for solving similar problems that may exist in other software systems. A software architect must find ways to make an architectural pattern fit for his software system. The architectural pattern is therefore tailored to fit a specific context so that the fundamental idea is reached to solve the common known software architectural problem [5].

2.4 Architectural tactics

A software tactic is an architectural concept that is very related to software patterns. Both architectural patterns and tactics are ways to capture design structures that have proven to be successful and good, and can therefore be reused. However, software tactics are much simpler than patterns. They usually just consist of a single structure or computational mechanism as they are only meant to address a single architectural element of a specific quality attribute. Tactics therefore often serve as “building blocks” of design when formulating an architectural pattern and software patterns typically consist of several software tactics. Therefore, tactics allow the architect to have a more precise control when implementing the architectural design of a system [5].

2.5 Architectural patterns relevant for the case study

In the next chapter (Chapter 3 – Fuglar’s case), several software patterns were relied on to construct the software architectural alternatives for the case under study, KRIA. These patterns are explained individually in the following subsections.

The majority of the following software architectural patterns have been invented and built around one common software architectural principle called “Separation of concerns”. This principle involves separating a software system into distinct sections where each section is used to address a specific concern. The idea is to avoid maintaining different types of software concerns within a single software component (such as in a single programming file). This has many benefits as it makes it easier to follow other important programming principles such as the “Don’t repeat yourself” principle which states that duplication in logic should be eliminated. By using architectural patterns that involve separating software concerns, the system becomes more maintainable and less tightly coupled [17].

2.5.1 Layered pattern

The layered pattern is a convenient architectural pattern for a complex system that needs to be evolved in several independent portions. It introduces units, called layers that are used to achieve this separation of concerns.

A layer should be responsible for a specific software concern and each layer group modules that offer a cohesive set of services. Among layers there should be a unidirectional allowed-to-use relationship, i.e. if layer A is allowed to use modules in layer B, B should not be allowed to use modules in layer A. However, depending on the circumstance, it can sometimes be reasonable to make minor exceptions.

By using the layered pattern the developers have a clear documented separation of concerns, making it possible to develop and maintain the system's modules independently. The downside of using the layered pattern is of course that it can be costly to design and build the layers. Furthermore, the system's complexity can increase, especially if the layering is not designed in an appropriate manner [5, pp. 205–210].

2.5.2 Multi-tier pattern

The multi-tier pattern is related to the layered pattern but it is important to not confuse tiers with layers. The layered pattern uses layers to group units of implementation while the multi-tier pattern uses tiers that strictly represent runtime entities. Still, tiers can be used to group components of similar functionality and in that case the pattern is considered a C&C pattern. However, the most common use of tiers is when they are defined in relation to the computing environment on which the software will run, making it an allocation pattern. For example, in an enterprise system the client tier would not be running on the same computer as the database machine (i.e. two tiers: client and database). Therefore, the main purpose of using the multi-tier pattern is to distribute the deployment of a system's infrastructure to distinct sections (tiers) that can e.g. be owned by different organisations.

The components groupings into tiers can be based on a variety of criteria, for example based on the type and purpose of the component, or sharing an execution environment etc. Between tiers there may exist many kinds of communication protocols for them to be able to communicate, and for some tiered patterns a call-return communication can exist in one direction but an event-based notification in the other.

For simple systems, it can be an overhead to use the multi-tier pattern as the system simply becomes more complex than necessary and the up-front and ongoing cost is not justifiable. However, the multi-tier pattern is ideal for complex systems where software artefacts may have to be distributed between several organisations [5, pp. 235–238].

2.5.3 Client-server pattern

The client-server is an appropriate software architectural pattern in a situation where there exist several different distributed clients that wish to access the same shared set of resources and/or services. Instead of having each client implementing the same logic repeatedly (fat-client approach [8]), clients interact by sending requests to common services that a one central server or multiple distributed servers provide.

By having a set of shared resources and services, it becomes easier to manage, modify and reuse software modules within a system as they are stored in a one common location (or several unique small locations). The main disadvantage of using the client-server pattern is that the

shared server is often a performance bottleneck as it can be serving several clients at the same time. Similarly, the shared server can be a single point of failure. Making decisions as to whether to locate the system's functionality in the client or server component can also be costly and hard to change if the system has already been built [5, pp. 217–219].

2.5.4 Service-Oriented Architecture (SOA) pattern

The service-oriented architecture pattern has very similar characteristics as the client-server pattern. The main difference lies in the interaction style as the client-server pattern uses a request-response style, i.e. the client sends a request to the server which replies with a response. However, SOA imposes no interaction style restriction [18]. Therefore, it can be concluded that all client-server architectures are SOA but not necessarily vice versa.

In a typical SOA architecture, several services are offered by service providers that are then consumed by service consumers, such as client applications or other services. The main purpose of using a SOA pattern is to support interoperability of distributed components that are running on different platforms, possibly written in different programming languages and can be provided by different organisations over the Internet [5, pp. 222–226].

There are several basic types of connector mechanisms used in SOA. One is called SOAP, a wide-spread protocol for allowing web-services to communicate. However, the protocol that the thesis uses in the next chapter (Chapter 3 – Fuglar's case) is called "Representation state transfer" (REST) and has become very popular. It allows the consumer to send a non-blocking HTTP request to the service component. The HTTP request can be on four commonly known formats (POST, GET, PUT, DELETE) to indicate the main characteristics of the request [5, pp. 223 & 108], [19].

REST aims to support many of the aforementioned software patterns as it proposes six constraints, see Box 3.

Box 3: RESTful protocol constraints [19].

- *Uniform interface:* The service provider should be easy to understand (self-descriptive messages etc.).
 - *Stateless:* The service provider does not store a specific state for the client consumer, the request itself contains the state (if necessary) in e.g. parameters, body or headers.
 - *Cacheable:* Consumers should be able the cache responses.
 - *Client-server:* Separation of clients and servers (see the client-server pattern above).
 - *Layered system:* The client should not ordinarily be able to tell whether it is directly connected to the end server or an intermediary (see the layered pattern above).
 - *Code on demand (optional):* Servers should be able to extend or customise functionality coming from the client by transferring it into executable code.
-

2.6 Architectural tactics relevant for the case study

As the thesis highly focuses on the maintainability quality attribute, some of the well-known architectural tactics that directly address the maintainability attribute must be mentioned. The following tactics defined by Len Bass et al. [5] were considered relevant for the thesis' case study research.

2.6.1 Split module

The split module tactic simply suggests that a software module should be refined into several smaller modules as it reduces the average cost of possible future changes [5, p. 123]. All of the aforementioned software architectural patterns are built around this tactic, i.e. the layered pattern uses layers, the multi-tier pattern uses tiers, the client-server pattern separates the client software from the service provider and the SOA pattern suggests that modules should be separated on multiple services.

2.6.2 Increase semantic coherence

The increase semantic coherence suggests that if two unrelated responsibilities are stored in a single software module, they should be separated into two modules [5, p. 123]. This is very related to the majority of the aforementioned software patterns, e.g. the layered pattern can use each layer to handle a specific software responsibility and the SOA pattern can similarly use services.

2.6.3 Encapsulate

The encapsulate tactic proposes a way for a module to interact with other modules through an API. The interfaces should be abstract to the modules that are using them [5, p. 123]. This tactic is good to use when constructing the server in a client-server pattern and the web-services in SOA.

2.6.4 Restrict dependencies

The restrict dependencies tactic suggests that modules should be restricted when interacting with, or depending on other modules [5, p. 124]. The layered pattern is a typical software pattern that is built around this tactic as it suggests that a layer is only allowed to use layers below itself.

2.7 Comparing software architectural alternatives - Quality attributes

In order to determine whether one software architecture is a better fit for a software system than another, several important aspects come into play. The ISO/IEC 9126 software standard defines six quality characteristics that a software system typically must take into account for the purpose of being usable and competitive to other systems. When comparing software architectures, the relevant quality characteristics from the standard can be used for the purpose of evaluating which architecture is a better fit for the system. This involves using relevant metrics to measure quality attributes. A direct quote from the ISO/IEC 9126 software standard [1] is given in Box 4 which briefly explains each of the six important quality attributes.

Box 4: Quality attributes from the ISO/IEC 9126 standard (direct quote from ISO/IEC 9126 [1]).

- *Functionality*: The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions.
 - *Reliability*: The capability of the software product to maintain a specified level of performance when used under specified conditions.
 - *Usability*: The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions.
 - *Efficiency*: The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.
 - *Maintainability*: The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.
 - *Portability*: The capability of the software product to be transferred from one environment to another.
-

In Chapter 4 – Case study approach, a further discussion about the maintainability quality attribute is given and e.g. a ground truth measurement method is put forth.

2.7.1 ATAM – Architecture Tradeoff Analysis Method

Quality attributes are important to consider when choosing a software architecture but there also exist well defined evaluation approaches to analyse software architectures with respect to the aforementioned quality attributes. One is called Architecture Tradeoff Analysis Method (ATAM), a leading method in the area of software architecture evaluation [20] as it provides insight into how quality attributes interact with each other and how they trade-off against each other [2], [21]. The fundamental steps when conducting ATAM are listed in Box 5 below.

Box 5: ATAM steps [21].

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. Analyse architectural approaches.

Testing

7. Brainstorm and prioritize scenarios.
8. Analyse architectural approaches.

Reporting

The qualitative approach used in the thesis relates to ATAM, however, the business drivers are specifically aimed at maintainability characteristics of the software architectural alternatives as this case study research specifically investigates the maintainability quality attribute. ATAM can still undoubtedly be used when evaluating software architectures with respect to

maintainability as the method is quite broad scope (in the sense that it can be used regardless of the quality attribute being investigated). For this reason, an example for Fuglar's case is given in the qualitative section of Chapter 4 – Case study approach. The example shows how one maintainability scenario could be evaluated by using the fundamentals of ATAM, such as an utility tree etc., and how the qualitative approach used in the study relates to the method.

2.8 Quantitative and qualitative architecture evaluation methods

In the past, the notion of evaluation has been used with many different definitions. A quite comprehensive definition given by the Joint Committee on Standards for Educational Evaluation [22] states that an evaluation is:

“A systematic investigation of the worth or merit of an object”.

This implies that the goal of an evaluation is to discover how valuable the object being evaluated is for some given context [6].

In many cases, using a metric is enough to measure how efficiently a quality attribute is being met by a software system. However, depending on the quality attribute, quantitative data collection does not always give the full picture or may not be relevant. This is where qualitative evaluation and data comes into play as it can be more suitable for evaluating some of the quality attributes [1], [23].

2.8.1 Quantitative methods

Quantitative research and methods specifically rely on numerical data that can be transformed into usable statistics. Quantitative methods are often related to the positivist paradigm where the positivist believes that only a single measurable reality exists. In the past, quantitative methods have been considered to be a formal, objective and deductive approach to solve problems [3, pp. 35–36]. Namely, their findings can be easily generalised from the study sample to a larger target population as the numerical data is used with respected statistical tests [24], [25] etc.

2.8.2 Qualitative methods

Qualitative research is usually considered more informal than quantitative research as it is based on subjective data such as people's opinions. It aligns with the naturalistic paradigm where there can be multiple interpretations of the reality, in contrast to quantitative methods where only a single measurable reality is believed to exist. The findings are often hard to generalise beyond the specific study sample [3, pp. 35–36].

3 FUGLAR'S CASE

Software architectural alternatives proposed for the case

In this case study research, three software architectural alternatives were defined and their corresponding software prototypes implemented. This chapter explains the fundamental aspects of each architectural alternative using both component and deployment diagrams, and the corresponding prototypes are discussed. All three software architectural alternatives describe the same software system, KRIA, and they all aim to provide the same user functionality. Still, their internal structure and runtime environments are quite different.

Before individually discussing each architectural alternative, KRIA as a software system is briefly introduced.

Note, this chapter strictly discusses domain specific concerns regarding KRIA as a software system. Software architectures from any software system could have been used to conduct the research investigation but KRIA was considered a good case study candidate.

3.1 KRIA's main functionality and features

KRIA is a very robust and widespread pension fund system, particularly designed for, and used by, large pension funds in Iceland. The system has been under constant development by Fuglar Ltd., an Icelandic software firm, for the past 20 years and the system's evolution is expected to be ongoing in the future. The system consists of a few hundred application forms that often have very similar software functionality. A typical application form in KRIA provides the user with the five features (CRUDL), see in Box 6.

Box 6: CRUDL functions.

1. A *create* function, that allows the user to add an entry to the list.
 2. A *read* function that allows the user to read specific information about a selected entry from the list.
 3. An *update / edit* function, that allows the user to edit an entry from the list.
 4. A *delete* function, that allows the user to delete a specific entry from the list.
 5. A *list* function, that presents the user with contents from a single or multiple (joined) database tables.
-

These five features are explained in relation to a GUI in Figure 3, which contains an example application form from KRIA's original software system.

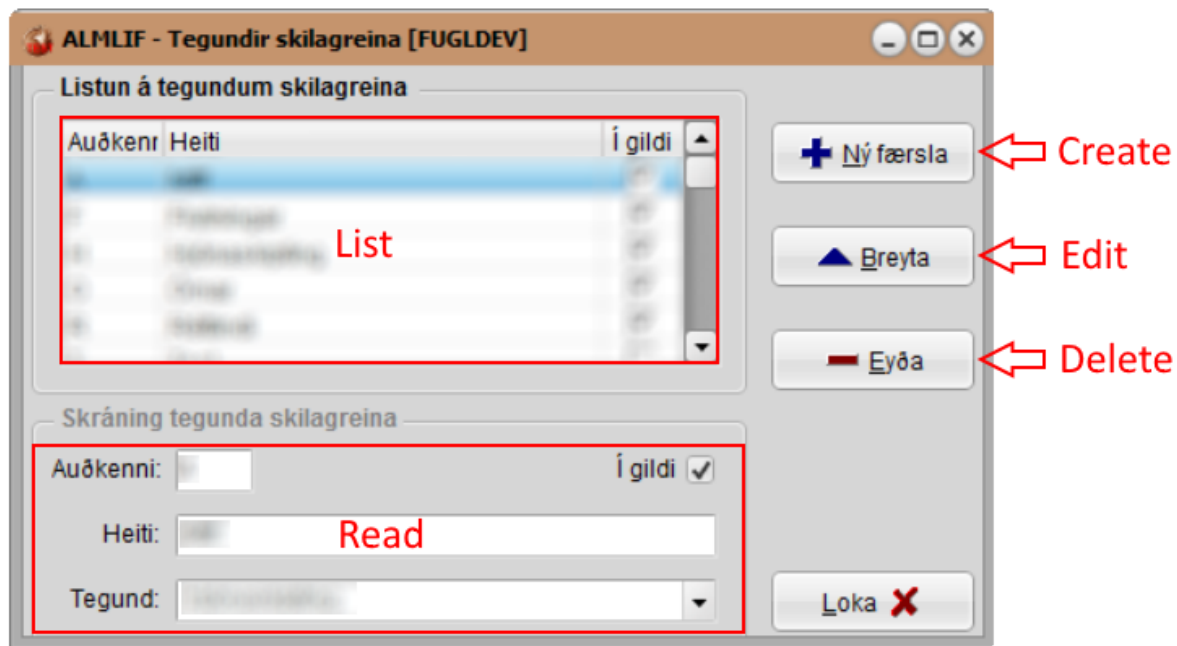


Figure 3: Example application form from KRIA’s original software system. The red elements (texts) in the figure represent the five key features that each application form typically entails.

3.2 Software architectural alternatives

In the following subsections, KRIA’s original architecture is documented and two new software architectural alternatives are proposed for the system.

3.2.1 KRIA’s original software architecture (A-ORIG) – Problem area

This first software architecture for KRIA (A-ORIG) was constructed differently from the second two architectural alternatives as it is based on an existing system, KRIA’s original system. Also, there was no need for the researcher to implement a tangible software prototype for this architectural alternative as a working software application already existed.

KRIA’s original software architecture is a typical fat-client [8] approach to software developing, the reason being since most of the software concerns are dealt with and stored in a single client application using Delphi source code. Even though the system is highly dependent on contents from a database, the database SQL queries and most of the business logic is stored in the client application along with the graphical user interface (GUI). For example, each application form (like in Figure 3) in the system is typically defined in a single programming file, regardless of what software aspect is being handled, i.e. GUI, business- and data logic is closely intertwined.

The component diagram in Figure 4 explains this software architectural alternative by using only two top-level components, i.e. the “DelphiClient” component and the “Database”.

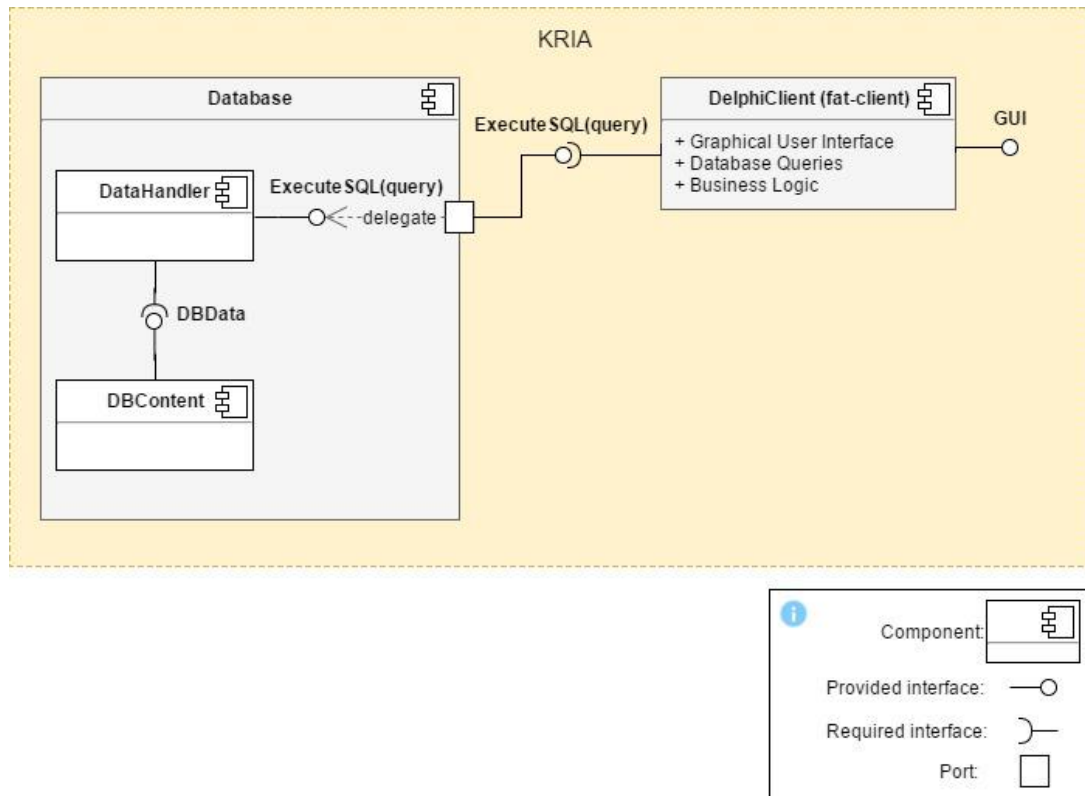


Figure 4: Main component diagram for KRIA's original architecture (A-ORIG).

Explanation of each top-level component in Figure 4:

1. DelphiClient (top-level component):

For each application form within the original system the GUI, database queries and business logic are defined in a single source file (.pas file and a corresponding .dfm file, which is autogenerated). For example, a user's sign-in form would be programmed in the following manner:

- The source code needed to represent the GUI, such as the username and password fields would be stored in a single "login.pas" file and a corresponding "login.dfm" file.
- The necessary SQL database queries (e.g. to check if a user exists) would also be stored in the "login.pas" file. The queries are then sent to another component that executes the queries and returns data to the "DelphiClient" component.
- The business logic would also be stored in the "login.pas" file. It defines logic in regard to what should happen if the user enters wrong / correct user credentials etc.

This is obviously a very monotonous way of programming and the software concerns are closely intertwined. Therefore, all of these software concerns are represented in this single top-level component and its main purpose is to provide a GUI interface (Windows desktop application) to an external entity (the user) for it to be able to interact with the system. As can be observed, the only thing missing from this client component is the actual data coming from the database, although the component stores the database queries. For the component to receive necessary data from the database it uses an interface "ExecuteSQL(query)" which takes as an input a SQL database query and returns the expected output to the caller component.

2. Database (top-level component):

This component is in charge of executing the SQL queries and stores the system's database content. The component can be divided into the following two internal components:

- a. *DataHandler (internal component)*: Executes database queries that come from the client component. The "ExecuteSQL(query)" interface returns the expected data to the "DelphiClient" if the query was executed successfully.
- b. *DBContent (internal component)*: Contains definitions of database tables, such as table names, columns info, foreign and private key relationships etc. It also contains the actual data such as row entries in the database tables. This data is provided to the data handler component through the "DBData" interface.

As the main purpose of this component is to handle the system's database content, it provides a single interface "ExecuteSQL(query)" from its internal "DataHandler" component that allows other components (such as the "DelphiClient" component) to access the data from the database by using SQL queries.

As the component diagram only has two top-level components that need to be deployed separately, the deployment diagram in Figure 5 uses two corresponding device nodes.

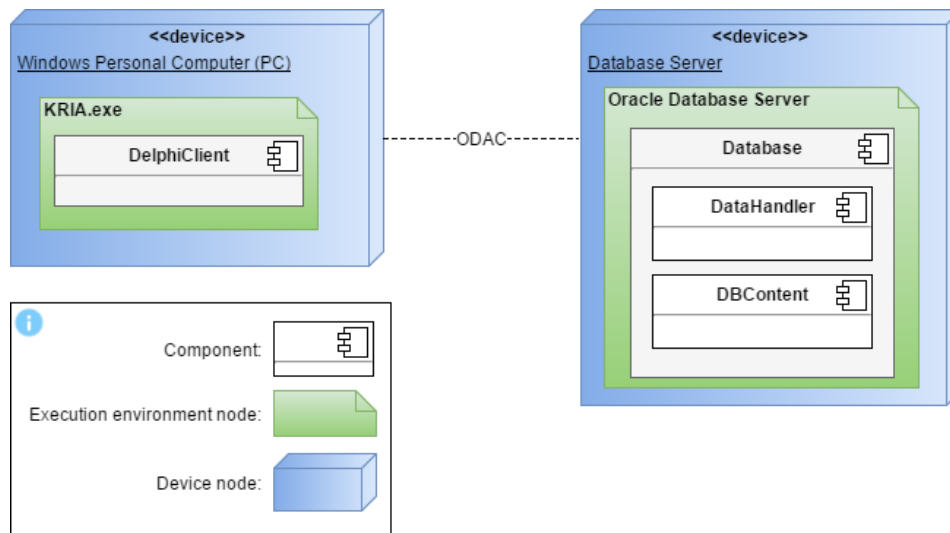


Figure 5: Deployment diagram for KRIA's original architecture.

Explanation of each device node in Figure 5:

1. Windows Personal Computer (PC):

The Windows personal computer node stores the Delphi client application that must be installed on the user's computer before he/she uses the system. The user runs the system by executing a run-file called "KRIA.exe" which starts up KRIA. As the "DelphiClient" component contains a significant part of the system's software logic, this desktop application is relatively large.

2. Database Server:

The database server stores the Oracle database that the client depends on and contains all data information required by the client. The Delphi client application interacts with the database by using Oracle data access components (ODAC) [26].

As can be observed from the deployment diagram, this software architecture is built around the client-server software architectural pattern and the split module tactic. However, as most of the

data logic (SQL-queries) is still defined in the Delphi client application, only limited part of the database management can be shared through the database server. Therefore, this is not a typical client-server approach and also a weak usage of the split module tactic. For example, if another client needs to use the database server, that client has to re-define or define a new set of database queries and business logic.

Prototype – Proof of concept:

Implementation of a software prototype was not necessary for this architectural alternative as there already existed a working software application. Table 1 shows in which development environment each top-level component was implemented. Appendix A.1 contains a more thorough explanation regarding the GUI and the software code for this prototype (software application).

Table 1: IDEs used to implement KRIA's original software application.

Component	Development environment (IDEs)
DelphiClient	Delphi [27]
Database	PL/SQL [28]

3.2.2 KRIA's SOA software architectural alternative (A-REST)

This initial proposed software architectural alternative for KRIA aims to make an extensive modification to the original system on the architectural level. This involves moving the database queries and business logic from the client application to a separate web-service, allowing the system to support interoperability of distributed components running on different platforms that are written in different programming languages.

This architectural alternative is built around several software architectural patterns and tactics (previously introduced in Chapter 2 – Theory). Firstly, the SOA pattern is achieved as the main service provider is to be implemented with a web-service and the database server still runs separately from the client's software. This obviously concludes that the split module and the increase semantic coherence tactics are efficiently used. Secondly, this is also in line with the client-server architecture but unlike A-ORIG, the client software now communicates with the database through an intermediary web-service. Thirdly, it can be concluded that the multi-tier pattern is achieved to some extent as three runtime entities are used (see the device nodes on the deployment diagram in Figure 7), where each entity is responsible for a single software concern, i.e. the client software is responsible for the GUI, the web-service for the business logic and the database server for the data management. Furthermore, the layered pattern is followed which indicates that the restrict dependencies tactic was used.

The component diagram in Figure 6 explains relationships between components for this software architectural alternative and how software concerns are now stored in the system.

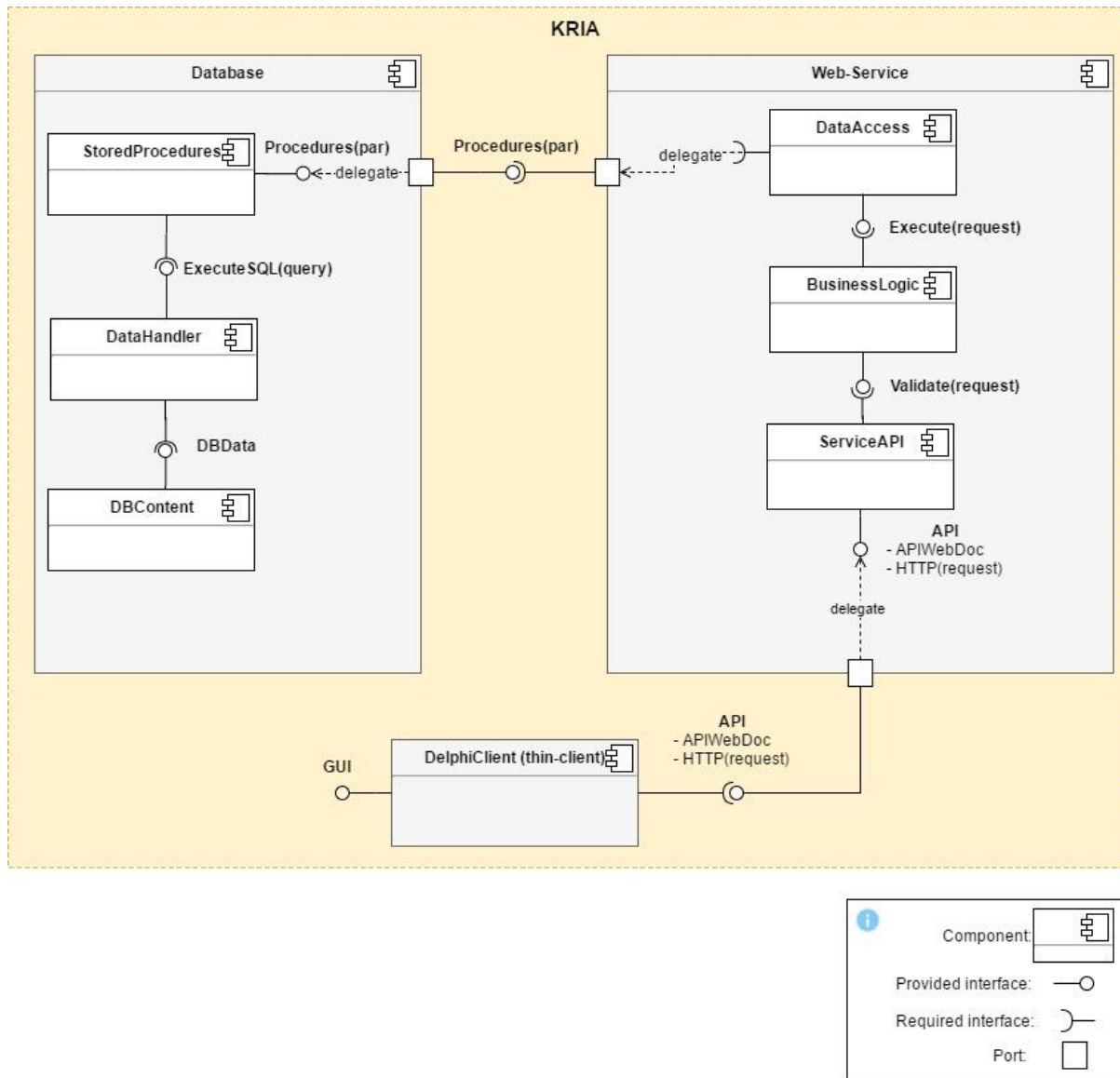


Figure 6: Main component diagram for KRIA's RESTful service architectural alternative (A-REST).

Explanation of each top-level component in Figure 6:

1. DelphiClient (top-level component):

The main purpose of this component is to address a single software concern, i.e. the graphical user interface (GUI) for the client application. The client application is still programmed in Delphi but unlike the A-ORIG architecture, this Delphi client component only implements the GUI. As the component still needs to display data from a database it relies on an "API" interface from another component. When displaying data, the component sends a request to the "HTTP(request)" interface and waits for a response. When the response is received, the component is in charge of displaying the data to the user through the "GUI" interface.

2. Web-Service (top-level component):

The "Web-Service" component for this architectural alternative was implemented with a RESTful web-service (see explanation of prototype below). Therefore, despite the fact that other types of web-services (such as SOAP) exist, the focus is on RESTful characteristics. This "Web-Service" component serves as an intermediary between the

client and the database component. The component handles several software concerns and is divided into three internal-components:

- a. *ServiceAPI (internal component)*: This component represents the interface of the web-service which contains a set of HTTP (e.g. GET and POST) methods. The component does not only provide the HTTP methods but also a documentation for each HTTP method. A developer that seeks to use the REST service for his/her software client can use this documentation as it e.g. explains the methods and the parameters necessary to make an appropriate HTTP request to the web-service.
- b. *BusinessLogic (internal component)*: This component is in charge of validating the HTTP requests (including parameters) coming from the client application and ensures that the returned data is in the right format. If the validation is successful the component uses the “Execute(request)” interface that forwards the request to another component that executes the request. The results are then forwarded back to the caller component.
- c. *DataAccess (internal component)*: If the “BusinessLogic” component successfully managed to validate the incoming HTTP request, this component calls the external database component and asks for a corresponding stored-procedure to be executed. When a response is received, the results are forwarded to the “BusinessLogic” component which forwards the results to the “ServiceAPI” component which then finally forwards the results back to the client (caller) component.

3. Database (top-level component):

This component is in charge of the entire database management. Its implementation is very similar to the A-ORIG architecture. The database queries have however now been moved from the Delphi source code to stored procedures which are now stored there along with the database. The component is divided into the following three internal components:

- a. *StoredProcedures (internal component)*: This component provides a set of database stored procedures, a centralised SQL logic that can be used to execute data requests to the database.
- b. *DataHandler (internal component)*: Executes SQL database queries that come from the stored procedures component. The “ExecuteSQL(query)” interface returns the expected data to the stored procedures component if the query is executed successfully.
- c. *DBContent (internal component)*: Contains definitions of database tables, such as table names, columns info, foreign and private key relationships etc. It also contains the actual data, such as rows in the database tables etc. This data is provided to the “DataHandler” component through the “DBData” interface.

The deployment diagram in Figure 7 contains three device nodes and describes where the components from Figure 6 are stored at runtime.

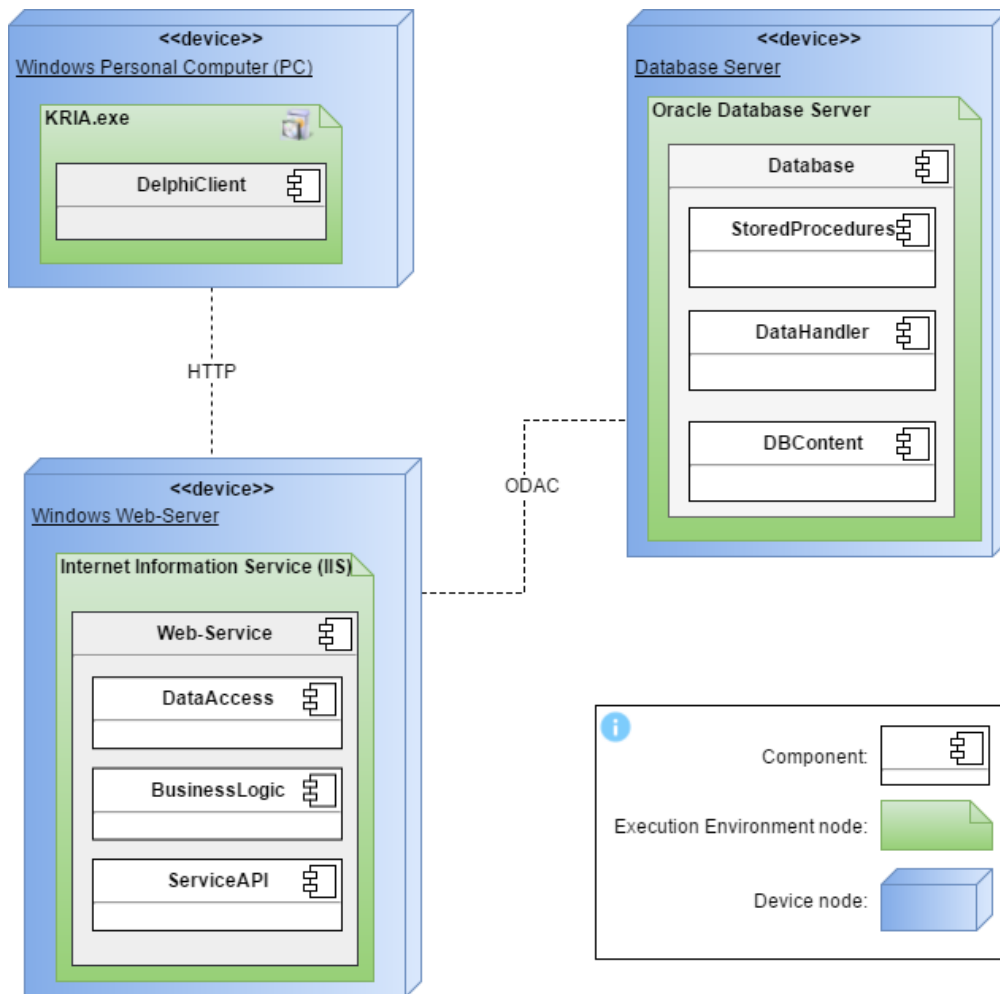


Figure 7: Deployment diagram for KRIA's RESTful service architectural alternative.

Explanation of each device node in Figure 7:

1. Windows Personal Computer (PC):

The “Windows Personal Computer” node stores the Delphi client application that must be installed on the user’s computer before he/she uses the system. The user runs the system by executing a run-file called “KRIA.exe” which starts up KRIA. The “DelphiClient” component is now significantly smaller in comparison with A-ORIG as much of the logic has been moved to the web-service. Therefore, this client application is quite small.

2. Windows Web-Server:

The “Windows Web-Server” is a Windows server machine that runs a Microsoft Internet information service (IIS) which hosts the web-service. The Delphi client application uses a HTTP protocol to make requests to the web-service.

3. Database Server:

The “Database Server” stores the database and contains all data information required by the client. The web-service interacts with the database by using ODAC [26].

Note, the internal implementation of the “Web-Service” component can be considered related to the “Layered Pattern” which does not strictly apply on runtime entities but simply defines separate units of implementation. This also applies for the internal components of the “Database” component.

Prototype – Proof of concept:

A simple proof-of-concept software prototype was implemented for this software architectural alternative. Similar to A-ORIG, the client software is implemented as a Windows desktop application by using Delphi source code. However, the intermediary web-service was implemented with Microsoft Visual Studio as it provides good support for implementing a RESTful web-service. It was also considered a good candidate for Fuglar Ltd. as the company in general specialises in Microsoft software solutions (is currently a Microsoft's Silver partner).

Several application forms from the original system were implemented (similar to the application form in Figure 3). A more detailed explanation of this software prototype can be found in Appendix A.2 but Table 2 explains the development environment each top-level component was implemented in.

Table 2: IDEs used to implement KRIA's RESTful service architectural alternative.

Component	Development environment (IDEs)
DelphiClient	Delphi [27]
Web-Service	Microsoft Visual Studio [29]
Database	PL/SQL [28]

3.2.3 KRIA's web-based software architecture (A-WEB)

This architectural alternative makes an extensive modification to the original system on the architectural level in a similar manner as for the A-REST alternative, i.e. by using a web-service as an intermediary between the client's software and the database engine. Therefore, it follows similar software patterns and tactics as the A-REST alternative. However, this software architectural proposal is also different on the GUI level as it proposes a new way of developing the client software, i.e. it uses a website instead of a Windows desktop application. If needed the website can still co-exist with the Windows desktop application developed in Delphi, i.e. they can share the web-service.

Using a website approach has several benefits. Firstly, the user authentication for the Delphi client approach (both A-ORIG and A-REST) is conducted directly through the database management, i.e. registered users and their credentials are stored in the database component. Microsoft has developed an Active Directory Federation Service (ADFS) [30] software component, a claim-based access-control authorisation model to maintain application security that can run separately on a Windows server machine. This component gives users single sign-on access to the system and provides secure sharing across an extranet of identity information among trusted business partners (known as a federation). Therefore, this pre-built Microsoft component is an ideal solution to authenticate and manage KRIA's users in the future as it can be used to protect both the website and the web-service, given that both the website and web-service are developed in ASP.NET. Secondly, both the website and the intermediary web-service are not strictly dependent on a single development environment (IDE), like the Delphi approach. However, they can still be developed in the same development environment, e.g. Microsoft Visual Studio and therefore the Delphi development kit can be dropped altogether. Thirdly, the Delphi approach specifically implements a Windows desktop application. However, a website simply requires a web-browser and can therefore be used regardless of the device's OS as long as it has web-browser support and an internet connection. Finally, deployment is efficient for developers as the website is hosted on a centralised server that the clients access through their web-browsers. The component diagram in Figure 8 explains

relationships between components for this software architectural alternative and how software concerns are now stored in the system.

Note, the “Web-Service” and the “Database” components are identical from the A-REST architectural alternative and therefore their internal components are hidden from this component diagram.

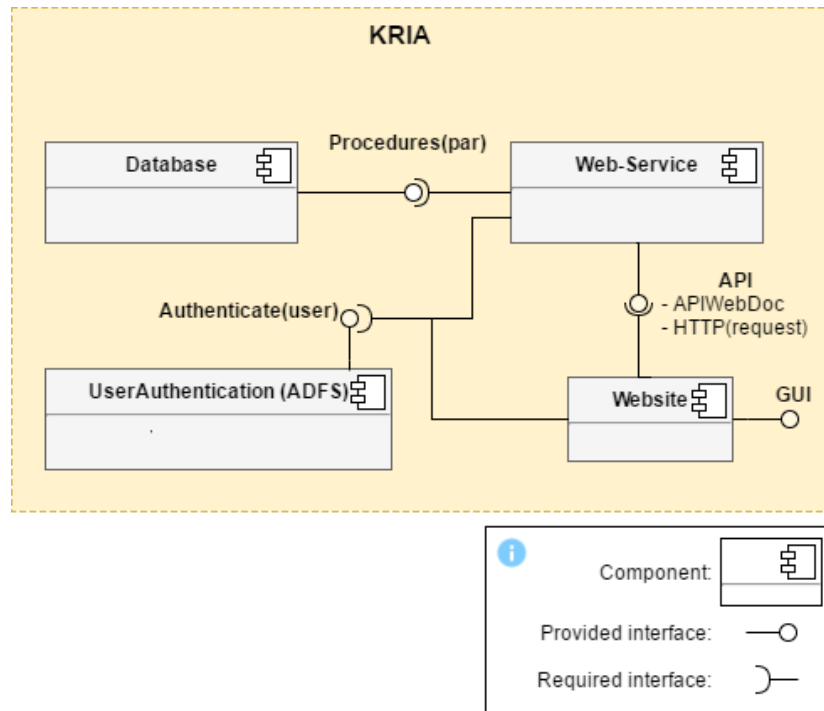


Figure 8: Main component diagram for KRIA's web-based architectural alternative (A-WEB).

Explanation of each top-level component in Figure 8:

1. **Web-Service (top-level component):**

Similar to the A-REST architectural alternative, this component contains an implementation of a RESTful web-service that serves as an intermediary between the client's application and the database.

2. **Database (top-level component):**

Similar to the A-REST architectural alternative this component is in charge of the entire database management.

3. **Website:**

This component replaces the Delphi client. In the thesis the corresponding software prototype was developed in ASP.NET with Microsoft Visual Studio and contains mainly C#, HTML, CSS and JavaScript (ReactJS/Redux) code needed to implement the GUI for the client. The website mimics the features from KRIA's original desktop application.

4. **UserAuthentication (ADFS):**

The original system (A-ORIG) authenticates users using manual SQL database queries and business logic. This “UserAuthentication” component uses an ADFS [30] component developed by Microsoft and is now responsible for KRIA's user authentication and management.

The deployment diagram in Figure 9 contains five device nodes and describes where the components from Figure 8 are stored at runtime.

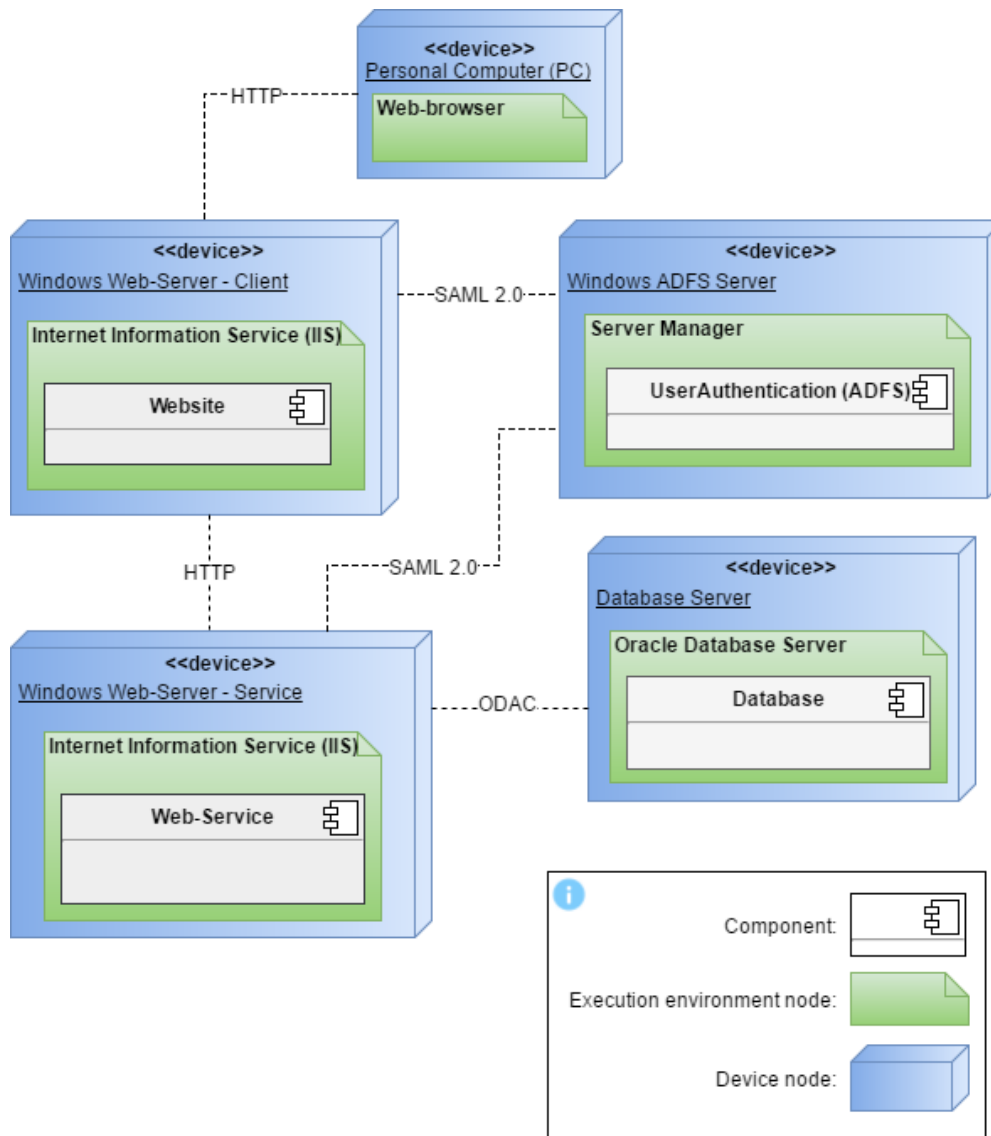


Figure 9: Deployment diagram for KRIA's web-based architectural alternative.

Explanation of each device node in Figure 9:

1. Personal Computer (PC):

The “Personal Computer” device node has a web-browser installed that can be used to access the system's GUI. The user runs the system by entering the correct URL that will link (DNS) to KRIA's application website.

2. Windows Web-Server - Client:

The “Windows Web-Server – Client” device node represents a Windows server machine which runs an IIS, responsible for hosting the “Website” component.

3. Windows Web-Server - Service:

The “Windows Web-Server – Service” device node is a Windows server machine that runs an IIS, responsible for hosting the “Web-Service” component. The client website uses an HTTP protocol to make requests to the REST service.

4. Windows ADFS Server:

This device node is a Windows server machine that has an ADFS server installed. It stores the “UserAuthentication” component. Both the client website and the web-service authorise users against this ADFS server.

5. Database Server:

The “Database Server” device node stores the “Database” component and contains all data information required by the client. The REST service interacts with the database by using ODAC [26].

Prototype – Proof of concept:

A simple proof-of-concept software prototype was implemented for this software architectural alternative. Similar to A-REST the “Web-Service” component was implemented as a RESTful web-service in ASP.NET, Microsoft approach. In order to maintain consistency the “Website” component was also implemented with ASP.NET. However, as KRIA can be considered a relatively large application it was decided to implement this prototype also with a third-party JavaScript library called ReactJS (with Redux) to ensure that modern web-development conventions were upheld. As the WebStorm IDE [31] has good support for writing ReactJS source code it was used along with Visual Studio when implementing this prototype.

Table 3 shows in what development environment each top-level component was implemented and a further explanation regarding this prototype is given in Appendix A.3.

Table 3: IDEs used to implement KRIA's web-based architectural alternative.

Component	Development environment (IDEs)
Website	Microsoft Visual Studio [29], WebStorm [31]
Web-Service	Microsoft Visual Studio [29]
UserAuthentication	- Not relevant (through ADFS [30])
Database	PL/SQL [28]

4 CASE STUDY APPROACH

Case study method and design

The research method approach applied throughout this master's thesis is based on the case study research methodology [32], as already indicated in the previous chapters. The aim of this chapter and its subsections is to thoroughly explain how the study was built around this methodology and the methods used in each required processing phase.

The case under study was Fuglar's situation when choosing a new software architecture for their software system KRIA. The three software architectural alternatives from the previous chapter (Chapter 3 – Fuglar's case) were under investigation and the objective was to make an assessment of their maintainability characteristic by using both quantitative and qualitative evaluation methods.

4.1 Why case study?

This case study was classified as an interpretive case study as it attempts to understand a phenomenon through the participants' interpretation of the context. Furthermore, some of the objects of the study could not be isolated and controlled, i.e. one could not only look at the proposed software architectural alternatives and quantitative results, they exist in a bigger context [32]. Two software prototypes were implemented to demonstrate corresponding architectural alternatives for KRIA. These prototypes are important for the case study for data collection purposes, which makes the study related to another research methodology, i.e. action research, with its purpose of "influence or change some aspect of whatever is the focus of the research" [33]. However, as the thesis uses the prototypes and KRIA's original software system to study effects of a change (on the maintainability quality attribute) the study was classified as case study research [32].

4.2 Case study design

Case study is a systematic research method approach and as for most empirical studies several steps must be undertaken. Per Runeson and Martin Höst [32] explain five major processing steps when conducting case study research. The steps are listed in Box 7.

Box 7: Case study steps.

1. *Case study design:* Objectives are defined and the case study is planned.
 2. *Preparation for data collection:* Procedures and protocols for data collection are defined.
 3. *Collecting evidence:* Execution with data collection on the studied case.
 4. *Analysis of collected data.*
 5. *Reporting.*
-

This case study research was divided into five activity phases that correspond to the previously mentioned processing steps. This subsection and the previous chapters have covered and

discussed the first step “Case study design” by listing up the objectives of the study and plan. The following subsections thoroughly declare the remaining activity phases.

4.3 Preparation for data collection

In order to efficiently make a data collection, the two types of evaluation methods (quantitative measuring and qualitative questioning) are defined concretely (see in subsections below) and the ground truth measurement method is put forth. However, it is not less important to establish the software architectural alternatives for the case under study (KRIA) in order to conduct the data collection with the well-defined evaluation methods. Therefore, three software architectural alternatives for KRIA were defined as can be observed in the previous chapter, Chapter 3 – Fuglar’s case. The qualitative questioning method could be applied based only on the three documented architectures, however the LOC quantitative measuring method used in the thesis had to rely on corresponding software prototypes in order to evaluate the maintainability aspect of the architectural alternatives. Therefore, the main purpose of constructing the tangible software prototypes for the architectural alternatives was to prepare the quantitative data collection but note that there may exist other potential quantitative methods to predict maintainability that do not specifically require software prototypes to be present.

The explicit definition of both the quantitative and qualitative evaluation methods is not only crucial for the data collection, but also for the research in general as the purpose of the study was to compare the two types of evaluation methods. The following three subsections thoroughly explain the ground truth measurement method and the two types of evaluation methods being investigated.

4.3.1 Building the ground truth – Implementation time measurement

The idea of predicting quality characteristics of a software system is not new and there exists a survey on software architecture analysis methods [2]. The survey discusses several valid architectural analysis methods that can be used and typically each method is geared to assess a single or several quality attributes. One of the methods, “Architecture Level Prediction of Software Maintenance” (ALPSM), makes an assessment specifically on the maintainability quality attribute. The method involves describing maintenance tasks/scenarios that may occur in a system and finally their maintenance effort can be predicted [2, p. 647].

The maintainability of a software system is defined as the capability of the system to undertake modifications [1]. Therefore, the method used in the thesis to determine the ground truth as to which of KRIA’s three architectural alternatives (A-ORIG, A-REST and A-WEB) is truly better with regards to maintainability is very similar to ALPSM, i.e. it involves measuring how long it actually takes to make a modification in each of the three software environments. Namely, the method involves the processing steps described in Box 8.

Prototypes corresponding to the software architectures were used in the thesis when applying the ground truth measurement method on the case. These prototypes were also needed for the LOC quantitative measuring method (see next section) as has already been discussed but the ground truth method was significantly harder to apply as it required remarkably more preparation.

1. *Define maintenance tasks:*

Maintenance tasks are defined. The comprehensiveness of these tasks depends on the software domain under study as they should be appropriate for the software system being investigated. To achieve good accuracy, several tasks should be defined but this also depends on the system's domain. For KRIA's maintainability assessment four maintenance tasks were defined, see Table 15 in Appendix B.

An example maintenance task for KRIA: "A new column needs to be added to an application form list (see in relation to Figure 3)".

The above maintenance task is typical for KRIA and requires the developers to e.g. add a new column to a data table in the database, add a new column in the GUI, and make proper modifications to the existing CRUDL functions with respect to the new column field.

2. *Weight assignment for maintenance tasks:*

After defining the maintenance tasks, each task needs to be assigned a weight value based on its probability of occurring. This can be done in several ways but for KRIA's case study a scale from 1-5 was used. A maintenance task that was considered very unlikely to occur was assigned the weight 1 while a task that was very likely to occur was assigned the weight 5. This weight assessment was done in collaboration with KRIA's stakeholders (developers).

3. *Time measurements:*

Finally, the time it takes a developer to accordingly implement these maintenance tasks in each of the software environments is measured. However, some maintenance tasks take a relatively long time to implement regardless of the software architecture being used. Therefore, to limit the actual implementation time factor of an individual maintenance task, the architectural alternative having the longest implementation time for the given maintenance task is assigned the time value 1,0 which is then multiplied by the task's corresponding weight value. The remaining software architectural alternatives get a proportion of that value respectively to their time measurement value for the given maintenance task. To demonstrate, consider the following example:

A maintenance task has been assigned the weight value 3. For A-ORIG it takes 85 minutes to implement the task, for A-REST it takes 50 minutes and for A-WEB it takes 60 minutes. As it takes longest to implement the task for A-ORIG (85 minutes), it is assigned the time value $1,0 * 3 = 3,0$ for that architectural alternative. For A-REST, $\frac{50 \text{ minutes}}{85 \text{ minutes}} \approx 0,59$. Therefore, the time value for A-REST is $3 * 0,59 \approx 1,76$. Finally, for A-WEB, $\frac{60 \text{ minutes}}{85 \text{ minutes}} \approx 0,71$. Therefore, the time value for A-WEB is $3 * 0,71 = 2,13$.

As can be concluded from the example, regardless of how long it actually takes to implement the given maintenance task (in minutes), the weight factor is the one that decides the task's impact on the implementation time value. The software architectural alternative having the shortest average implementation time value serves as the ground truth to determine which of the software architectural alternatives truly is the most maintainable.

The researcher and author of the thesis was responsible for implementing the maintenance tasks in each of the three software environments for KRIA as he was not more familiar with one environment than the other. As each task had to be implemented three times (once for each software architectural alternative), the researcher's learning curve of implementing the same maintenance task several times can influence the result. However, to reduce this undesirable effect, the tasks were implemented in a certain order, i.e. if the first maintenance task was to be implemented in A-ORIG that same task would not be implemented in the second architectural alternative until the remaining tasks had been implemented at least once in each of the architectural alternatives (environments). Also, to ensure fair measurement, a test implementation was conducted for each task in each software architectural alternative. The main reason for having a test run was to guarantee a stable implementation pace for the actual time measurements and limit the influence of the researcher's programming background. Also, a time measurement was redone if an irrelevant programming error occurred when conducting the time measurement (simply to ensure stable programming pace for each architectural alternative).

The entire list of maintenance tasks (including weight values) can be found in Table 15 in Appendix B.

4.3.2 Qualitative approach

Several potential qualitative approaches exist that can be used when choosing a software architecture for a system. In this study, a qualitative questioning method is used, which relates to the ATAM method (previously introduced in Chapter 2 – Theory). As already discussed in Chapter 2 – Theory, ATAM in general does not exclusively focus on the maintainability quality attribute (like ALPSM). Rather, ATAM is normally used as a trade-off analysis method where multiple software quality attributes that are considered important by the stakeholders are being evaluated and investigated. Still, the qualitative questioning method approach used in this study relates to ATAM as it depends on the stakeholders' involvement and is also built around some of ATAM's mandatory processing steps [21]. In the following two subsections, ATAM will first be discussed specifically for the case (Fuglar Ltd.), i.e. an example is given mainly focusing on one maintainability scenario which demonstrates how ATAM could have been used on the study's case. Secondly, as ATAM can be considered a quite comprehensive method, the main qualitative method used in the study is the qualitative questioning method which relates to some extent to the ATAM method as described in the later subsection.

4.3.2.1 ATAM approach, example

ATAM can either be used for any desired combination of quality attributes or with a focus on only a single quality attribute, e.g. maintainability. The following example shows how one quality attribute scenario relevant for Fuglar's case can be evaluated by using ATAM.

ATAM step-by-step demonstration for Fuglar's case (see in relation to Box 5):

Presentation

1. *Present the ATAM*

In this step, the evaluation leader describes the fundamentals of ATAM to the participants [20] (in this case Fuglar's stakeholders).

2. *Present business drivers*

In this step, the project manager (or the system's customer) describes the main business goals and what will then be the primary architectural drivers [20]. As has already been discussed, Fuglar's main business drivers for its software system KRIA relate to

maintainability. However, maintaining appropriate level of performance is also important, as well as keeping good user security and usability. Below, Fuglar's main business drivers are stated based on discussions with the stakeholders.

- The system needs to be easy for the developers to modify and change (maintain) according to customer desires in the future.
- The system must entail appropriate response time (performance).
- Users should at all time be safely authorised to the system (security).
- The system must be easy to use and understand (usability).

3. *Present architecture*

In this step, the architectural candidates are presented for the system at hand (with a special focus on the previously stated business drivers) [20]. The candidates used for Fuglar's case can be observed in Chapter 3 – Fuglar's case.

Investigation and analysis

4. *Identify architectural approaches*

In this step the architectural approaches are identified by the architect but are not analysed [20]. The main architectural approaches for the three software architectural alternatives of KRIA are the following (see in relation to Chapter 3 – Fuglar's case):

- A-ORIG
 - General: Windows app, fat-client approach, single layer.
 - Pattern: Client-server (pattern).
- A-REST
 - General: Windows app, thin-client approach, RESTful web-service.
 - Patterns: Client-server, SOA, Multi-tier, Layers.
 - Tactics: Split module, increase semantic coherence, restrict dependencies.
- A-WEB
 - General: Website, thin-client approach, RESTful web-service, ADFS.
 - Patterns: Client-server, SOA, Multi-tier, Layers.
 - Tactics: Split module, increase semantic coherence, restrict dependencies.

5. *Generate quality attribute utility tree*

The main purpose of creating a utility tree is to directly and efficiently translate the business drivers of the system into concrete quality attribute scenarios using a top-down mechanism [21]. As this demonstration example is solely meant to illustrate how ATAM could have been used for Fuglar's case, only one maintainability quality attribute scenario was formulated (as maintainability is of highest interest for Fuglar). Still, the utility tree in Figure 10 contains leaf nodes with the other important quality attributes that should also be considered when choosing a software architectural alternative for KRIA. Note, if the ATAM method would have been fully applied in the case, several quality attribute scenarios would have been formulated for each of the stated quality attributes (replacing the yellow squares of the utility tree in Figure 10).

The scenarios of a quality attribute utility tree are usually prioritised according to two dimensions, i.e. the importance of the node to ensure the system's success and the risk posed by the achievement of the node [20]. In the quality attribute utility tree below a scale High (H), Medium (M), Low (L) is used for prioritisation purposes. The

highest/best priority node would therefore have the rank (H,L), i.e. meaning that the node has high importance for the system's success and has low risk of being achieved. On the contrary, a node having the rank (L,H) would have the lowest/worst prioritisation order.

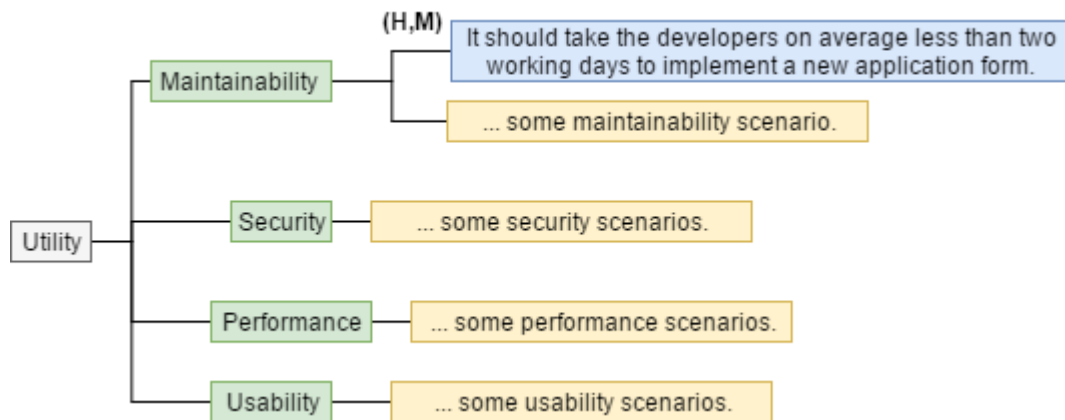


Figure 10: Example quality attribute utility tree.

Notice, the maintainability quality scenario (see blue square in Figure 10) is very similar to the first question used in the qualitative questioning method (see questionnaire in Appendix C). This was intentional as when asking the stakeholders the questions from the questionnaire, several of them expressed their thoughts about the importance of this first question. Therefore, this scenario was assigned a high importance (H) for the system's success and a medium importance (M) regarding the risk of achievement.

6. Analyse architectural approaches

Based on the priority ranking of the scenarios from the previous step, architectural approaches (from step 4) are elicited and analysed with the intention of achieving one or more utility tree sub-factors [21]. The scenarios are investigated with respect to their risks, sensitivity points and trade-off points. In Table 4 a demonstration is given of how the maintainability scenario from Figure 10 would be handled with ATAM, by taking into account each of the proposed software architectural alternatives from Chapter 3 – Fuglar's case. Evidently, if conducting ATAM to a full extent, more scenarios would be analysed in a similar fashion (starting with the high-priority scenarios).

Table 4: Architectural approaches analysed according to a quality attribute scenario.

Scenario	It should take the developers on average less than two working days to implement a new application form.			
Attribute	Maintainability.			
Environment	Normal workload on staff (developers).			
Stimulus	A pension fund (client) requests a new application form to be implemented.			
Response	Two working days on average to implement the requested form.			
Architectural alternative (A-ORIG)	Decision	Risk	Sensitivity	Tradeoff
	Fat-client	R1	S1	T1
Reasoning	<ul style="list-style-type: none"> - KRIA's fat-client approach allows the developers to work faster as they have to use fewer software tools (IDEs) when developing. - Also, as there are relatively few software layers, the developers may find it easier to respond to the required maintainability task. 			
Architectural diagrams	See Figure 4 and Figure 5.			
Architectural alternatives (A-REST, A-WEB)	Decision	Risk	Sensitivity	Tradeoff
	Thin-client	R2	S2	T2
	Multi-tier	R3	S2	T3
	Layers	R4	S2	T4
Reasoning	<ul style="list-style-type: none"> - Both A-REST and A-WEB suggest using a thin-client approach which can make it easier for the developers to push new updates, i.e. the developers may not have to update the client application manually. - Adding tiers and layers can make the code easier to understand and maintain. 			
Architectural diagrams	See Figure 6, Figure 7, Figure 8 and Figure 9.			

Table 5: Architectural risks.

R1	Using a fat-client approach can make the code hard to read and understand. It can also make it difficult for several developers to work together on a change. Furthermore, maintaining SQL queries as strings in the client application can open loopholes for SQL injection.
R2	Using a thin-client approach can make it more difficult for the developers to quickly react to the client's maintainability request as the developers have to modify several implementation layers.
R3	Having too many tiers can make the system hard to understand and deploy.
R4	Having too many layers can make the code hard to understand and possibly result in replication of functionality.

Table 6: Sensitivity points.

S1	Storing a big portion of the system's source code in the client application requires a relatively large hard drive on the client's computer. For the client, this can be a bottleneck.
S2	Storing a large part of the system's source code in a different location than the client application makes the system more vulnerable and dependent as it depends on external software components to function accurately. For example, a short downtime of a required service can have a significant impact on the system's performance.

Table 7: Trade-off points.

T1	By using a fat-client approach, the code can be difficult to understand (maintainability reduced) but as there only exist few software layers (only two, see component diagram in Figure 4), the system's performance should benefit.
T2	A thin client approach can have a negative influence on the system's performance but its maintainability characteristic can increase.
T3	Adding run-time tiers can have negative influence on the system's performance but its maintainability characteristic can increase.
T4	Grouping units of implementation into specific software layers can have negative influence on the system's performance but its maintainability characteristic can increase.

Testing

The Testing phase, which is often called the second phase of ATAM entails two steps "Brainstorm and prioritise scenarios" and "Analyse architectural approaches" [21]. This phase is important as a new, larger set of scenarios is elicited and prioritised with the entire stakeholder group. These scenarios then serve as test cases to confirm the analysis performed so far.

As this example only uses one quality attribute scenario to demonstrate ATAM for Fuglar's case, this testing phase of ATAM will not be demonstrated here. Still, this phase is equally (if not more) important than the previous steps as if there have been formulated several scenarios to choose from, this phase assures that the architect is meeting the stakeholder's demands and business goals.

Reporting

In this step, the collected information and findings from using ATAM are presented back to the stakeholders.

4.3.2.2 The qualitative questioning method

As one can imagine, conducting ATAM to a full extent can be a quite difficult task as it is a comprehensive method and requires the involvement of many stakeholders. Therefore, in this study a more lightweight qualitative method is used, which similarly requires the involvement of several stakeholders to evaluate architectural alternatives based on their opinions.

The qualitative questioning method has to do with creating a well-established questionnaire which is then used to ask several developers/designers about subjective maintainability aspects regarding the three software architectural alternatives proposed for KRIA. The questions from the questionnaire are formulated based on the business drivers that are explained by the project manager and other relevant stakeholders. Maintainability serves as Fuglar's main motivation for the change and therefore the questions mainly target maintainability aspects as can be

observed from the questionnaire in Appendix C. Instead of prioritising scenarios as is the procedure with ATAM, the qualitative questioning method uses the questions from the questionnaire to prioritise the proposed architectural alternatives (see alternatives in Chapter 3 – Fuglar’s case). The maintainability quality characteristic and the suitability of each software architectural alternative is therefore determined based on relevant stakeholder’s opinions, i.e. developers, managers etc.

Several steps must be taken when creating and using a questionnaire. Initially the research question and the study population must be well defined. The questions in the questionnaire must then be formulated and the response options defined [34].

The subject that the questionnaire seeks to investigate is based on the questionnaire’s research question, which is not the same as the case study research question previously stated in Chapter 1, the introduction chapter. The research question that the questionnaire seeks to answer involves predicting and making an assessment of which of KRIA’s three software architectural alternatives is the most maintainable.

The study population used was Fuglar’s team of developers/designers working on KRIA’s robust modifications. They were each interviewed separately and the questions from the questionnaire used.

The stakeholders were involved when formulating the questions for the questionnaire, similar to the scenario generation step in ATAM. The focus was on maintainability characteristics and the questions mainly involved aspects such as how modern and efficient the software architectural alternatives are, e.g. whether the integrated development environment (IDE) is better for one architectural alternative than the other (e.g. do the IDEs provide auto code completion etc.). Furthermore, questions were asked such as which architectural alternative seems to provide good future potentials, i.e. one was mainly programmed in Delphi while another mainly in C#, ASP.NET. The developers could therefore have opinions on which alternative has good future potentials based on these programming environments. Also, questions about the architectural structure of the alternatives were asked such as: “Is the code easy to understand in the software architectural alternative (e.g. since it has been separated into layers etc.)?”.

The response options for each question consisted of predefined multiple-choice responses, i.e. closed questions [34]. For KRIA’s case, the respondent was asked to position the three software architectural alternatives in relation to the question being asked. For example, if a question asked was: “Which of the three software architectural alternatives has a good development community?” the respondent had to prioritise the alternatives in the following manner:

- Good architectural alternative (+1p): _____
- Reasonably good architectural alternative (+0,5p): _____
- Bad architectural alternative (-1p): _____

The options allowed in each field being: *A-ORIG / A-REST / A-WEB*.

If the respondent’s opinion is that the development community is good for A-ORIG he puts that option in field number 1. If he thinks that the development community is reasonably good for A-ORIG he puts that alternative in field number 2 etc.

Note, if the respondent only knows that A-ORIG is a good architectural alternative for the given question but does not know where to position the last two alternatives, he can skip them and just answer the question for one architectural alternative. Similarly, if he thinks that both A-ORIG and A-REST are good alternatives for the given question he puts them both in field number 1.

The results are finally summarised by calculating the points (p) for each software architectural alternative. To demonstrate, consider the following example:

If A-ORIG was considered a good alternative for two questions ($2 * 1p = 2p$), a reasonably good alternative for 3 questions ($3 * 0,5p = 1,5p$) and a bad alternative for 2 questions ($2 * (-1p) = -2p$), the result would be $2p + 1,5 + (-2p) = 1,5p$. Since the respondent can skip answering a question for an architectural alternative that he/she is unfamiliar with, the final point value is divided by the total answer rate for the given alternative. In this example, the answer rate is $2 + 3 + 2 = 7$. Therefore, the final result would be $\frac{1,5p}{7} \approx 0,21p$.

These points (p) are then calculated with respect to each architectural alternative being investigated. The alternative having the highest number of points (p) is then considered to be the most maintainable.

The entire questionnaire may be found in Appendix C and the interview results are presented in the next chapter, Chapter 5 – Results.

4.3.3 Quantitative approach

There does not just exist a single quantitative metric that can be used to measure the maintainability quality characteristic of a software system. In fact, compound metrics even exist. For example, the “Maintainability Index” (MI) was introduced at the International conference on software maintenance in 1992 [35], which can be used to predict the maintainability by combining four metrics: Halstead Volume, Cyclomatic Complexity, Source Lines of Code and Comments Ratio [36].

Compound metrics such as the MI metric are debatable but possibly very good and accurate measurement methods to predict the maintainability of a software system. However, it can be quite difficult and time consuming to conduct the four underlying measurements to calculate the MI and studies have already shown that the four underlying metrics are directly correlated with the code size (lines of code) [37]. Researchers have therefore pointed out that simply measuring the lines of code (LOC) may give very similar results as MI [37]. Studies have also warned against using LOC as a software size measure [38] and therefore it was considered interesting to use Fuglar’s case to further investigate the LOC quantitative measuring method, i.e. if LOC measurements can be used to efficiently predict the maintainability of a software system instead of simply serving as a size measurement metric. Therefore, LOC was solitarily used as the quantitative evaluation method in this case study research for the purpose of quantitatively predicting the maintainability quality characteristic of KRIA’s software architectural alternatives.

The LOC measurement method involves counting the source lines of code that are needed to implement tangible software systems. Therefore, files from the two software prototypes that were implemented for KRIA, and files from the original software system were used as they correspond to the software architectural alternatives in Chapter 3 – Fuglar’s case. The same five application forms from each of the three software architectural alternatives (prototypes) were selected and the source files needed to implement these forms were gathered. The total source lines of code from these files were calculated with respect to each software alternative. Since one software prototype can rely on several source files for its implementation of a single application form (including e.g. GUI code, data queries etc.), it is implied that the LOC measurements for one architectural alternative may have taken into account significantly more/less source files than another.

The entire list of files and the LOC within each file can be found in Tables 12-14 (in Chapter 5 – Results), including the total amount of LOC for each software architectural alternative. The LOC measurements took into account all present lines in the software files, i.e. comments, empty lines and of course the actual source code lines.

4.4 Collecting evidence

After defining the aforementioned data collection methods, the methods were applied on the case. The data evidence (results) can be found in Chapter 5 – Results.

In a case study it is important to use several data sources to limit the effects of one interpretation based on a single data source [32]. Therefore, data triangulation [39] was used for each evaluation method. Firstly, for the ground truth measurement method, four different maintenance tasks/scenarios were defined and conducted in each of the three software environments. Secondly, for the qualitative questioning method, several different developers were interviewed to achieve data triangulation. Finally, for the LOC quantitative measuring method, source files needed to implement five application forms were used as data sources.

4.5 Analysis of collected data

After collecting the data evidence, a conclusion chapter was written (see Chapter 6 – Conclusion). The conclusion chapter contains an analysis of the collected data and an assessment is given on the two classes of evaluation methods (quantitative and qualitative) based on the data results. Also, the difficulty level of applying the evaluation methods is discussed and what each method requires to be present before it can be appropriately applied. This is in line with the research question and the objectives of the study.

4.6 Reporting

The final step when conducting a case study is reporting. This involves writing about each step of the case study and discuss its findings, i.e. for this case study: Had the two types of evaluation methods (quantitative measuring and qualitative questioning) been successful for predicting the maintainability of the system etc.

The thesis itself is the instrument used for the reporting.

5 RESULTS

Measurements results

This chapter contains the results from the ground truth measurement method (implementation time), the qualitative questioning method (questionnaire) and the quantitative measuring method (LOC). Not only are tables used to present the results but histograms and pie charts are also provided. Each chart has one architectural alternative (coloured in green) which represents the most maintainable alternative according to the given measurement method. After presenting the results from the evaluation methods, an interpretation section is provided and finally, threats to validity are discussed.

5.1 Implementation time - Ground truth measurement

Table 8 contains the maintenance tasks used for the ground truth measurement method. A more comprehensive description for each maintenance task can be found in Table 15 in Appendix B. Table 9 shows in which order the maintenance tasks were implemented and the actual time measurements (including the time values). Finally, Table 10 contains the average time value for each software architectural alternative.

Table 8: Maintenance tasks for KRIA and their corresponding weight values.

ID	Maintenance task – Title	Weight
T1	Add a new list column.	5
T2	Create a new application form.	3
T3	Remove an existing list column.	1
T4	Add a new search field.	2
Total weight:		11

Table 9: Implementation order and time measurements.

Order	Task ID	Architectural alternative	Implementation time (sec)	Time value
1	T1	A-ORIG	590	1,92
2	T2	A-REST	3294	2,79
3	T3	A-WEB	685	0,96
4	T4	A-ORIG	237	0,89
5	T1	A-REST	1535	5,00
6	T2	A-WEB	3538	3,00
7	T3	A-ORIG	242	0,34
8	T4	A-REST	530	2,00
9	T1	A-WEB	1265	4,12
10	T2	A-ORIG	2825	2,40
11	T3	A-REST	713	1,00
12	T4	A-WEB	426	1,61

Table 10: Average implementation time value for each software architectural alternative.

Architectural alternative	Average time value
A-ORIG	1,39
A-REST	2,70
A-WEB	2,42

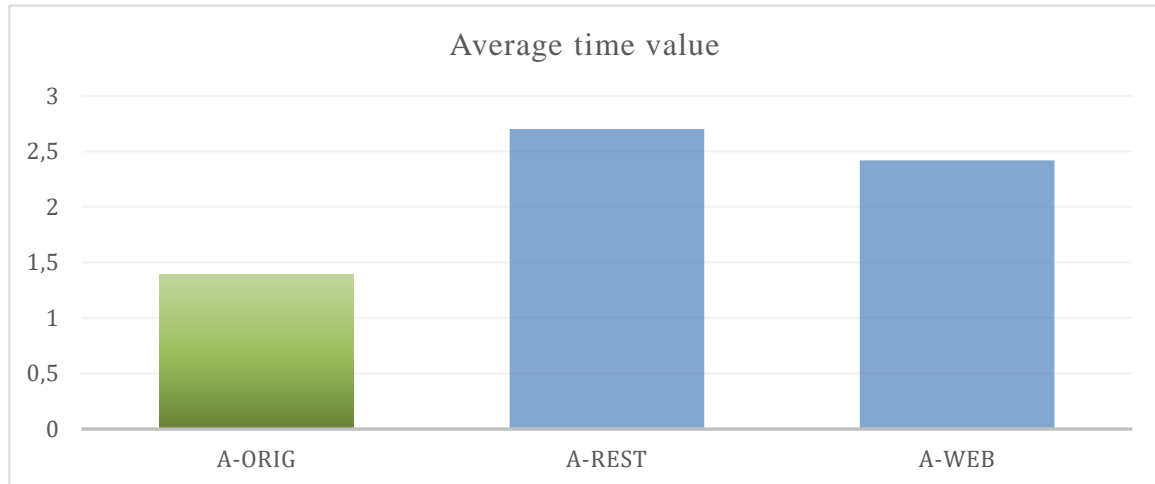


Figure 11: Average implementation time value (histogram).

5.2 Questionnaire - Qualitative questioning

The entire qualitative questionnaire is presented below, including the point value for each question in relation to the three software architectural alternatives. Table 11 contains the final results from using the qualitative questionnaire for each software architectural alternative.

Response rate: 7 responses out of 8 participants.

Questions:

1. Based on the architectural patterns and tactics being used to construct the given architectural alternative, do you think that the code is easy to understand and change in the future (e.g. since the code has been separated into layers etc.)?

A-ORIG: $1p + 0,5p - 1p - 1p - 1p - 1p + 1p = -1,5p$

A-REST: $0,5p + 1p + 0,5p + 0,5p + 0,5p + 0,5p + 1p = 4,5p$

A-WEB: $-1p + 1p + 1p + 1p + 1p + 0,5p + 0,5p = 4p$

2. Are the integrated development tools (IDEs) that are being used to maintain and evolve the given architectural alternative modern and easy to handle? For example, do they provide relevant auto completion, identification of coding errors etc.?

A-ORIG: $-1p + 1p - 1p + 1p - 1p - 1p + 0,5p = -1,5p$

A-REST: $0,5p + 1p + 0,5p + 1p + 1p + 1p = 5p$

A-WEB: $1p + 1p + 1p + 1p + 0,5p + 1p = 5,5p$

3. When coding problems occur that are directly related to the programming languages and IDEs being used, does the given architectural alternative have a good development community? For example, is it easy to search on the Internet for solution to a coding problem when it occurs?

$$\text{A-ORIG: } -1p + 0,5p - 1p + 0,5p - 1p - 1p + 0,5p = \mathbf{-2,5p}$$

$$\text{A-REST: } 0,5p + 1p + 0,5p + 0,5p + 0,5p - 1p + 0,5p = \mathbf{2,5p}$$

$$\text{A-WEB: } 1p + 1p + 1p + 1p + 1p + 1p + 1p = \mathbf{7p}$$

4. Do you think that the given architectural alternative has good future potentials?

$$\text{A-ORIG: } -1p + 0,5p - 1p - 1p - 1p - 1p - 1p = \mathbf{-5,5p}$$

$$\text{A-REST: } -1p + 1p + 0,5p - 1p + 0,5p + 0,5p + 0,5p = \mathbf{1p}$$

$$\text{A-WEB: } 1p + 1p + 1p + 1p + 1p + 1p + 1p = \mathbf{7p}$$

5. Maintaining good user security is easy for the given software architectural alternative?

$$\text{A-ORIG: } -1p + 0,5p + 0,5p - 1p - 1p + 1p = \mathbf{-1p}$$

$$\text{A-REST: } 0,5p + 1p + 0,5p + 0,5p + 0,5p + 0,5p = \mathbf{3,5p}$$

$$\text{A-WEB: } 1p + 1p + 0,5p + 1p + 1p + 0,5p = \mathbf{5p}$$

6. Adding brand new GUI features is possible and relatively efficient for the given software architectural alternative?

$$\text{A-ORIG: } 1p + 0,5p - 1p + 0,5p - 1p + 0,5p - 1p = \mathbf{-0,5p}$$

$$\text{A-REST: } 0,5p + 0,5p + 0,5p - 1p + 0,5p + 0,5p - 1p = \mathbf{0,5p}$$

$$\text{A-WEB: } -1p + 1p + 1p + 1p + 1p + 1p + 1p = \mathbf{6p}$$

7. For the given software architectural alternative, is it efficient for multiple programmers to co-operate on evolving and maintaining the system?

$$\text{A-ORIG: } 1p + 0,5p - 1p + 0,5p + 0,5p - 1p + 1p = \mathbf{1,5p}$$

$$\text{A-REST: } 1p + 0,5p + 0,5p - 1p + 1p + 0,5p + 1p = \mathbf{3,5p}$$

$$\text{A-WEB: } 1p + 1p + 1p + 1p + 1p - 1p + 1p = \mathbf{7p}$$

8. Based on your previous answers, do you consider the given architectural alternative efficient to maintain?

$$\text{A-ORIG: } -1p - 1p - 1p - 1p - 1p - 1p + 0,5p = \mathbf{-5,5p}$$

$$\text{A-REST: } 0,5p + 0,5p + 0,5p - 1p + 0,5p + 0,5p + 0,5p = \mathbf{2p}$$

$$\text{A-WEB: } 1p + 1p + 1p + 1p + 1p + 1p + 1p = \mathbf{7p}$$

Table 11: Qualitative questioning results.

Architectural alternative	Answer rate	Total points (p)	Average points (p)
A-ORIG	55	-16,5	-0,30
A-REST	54	22,5	0,42
A-WEB	54	48,5	0,90

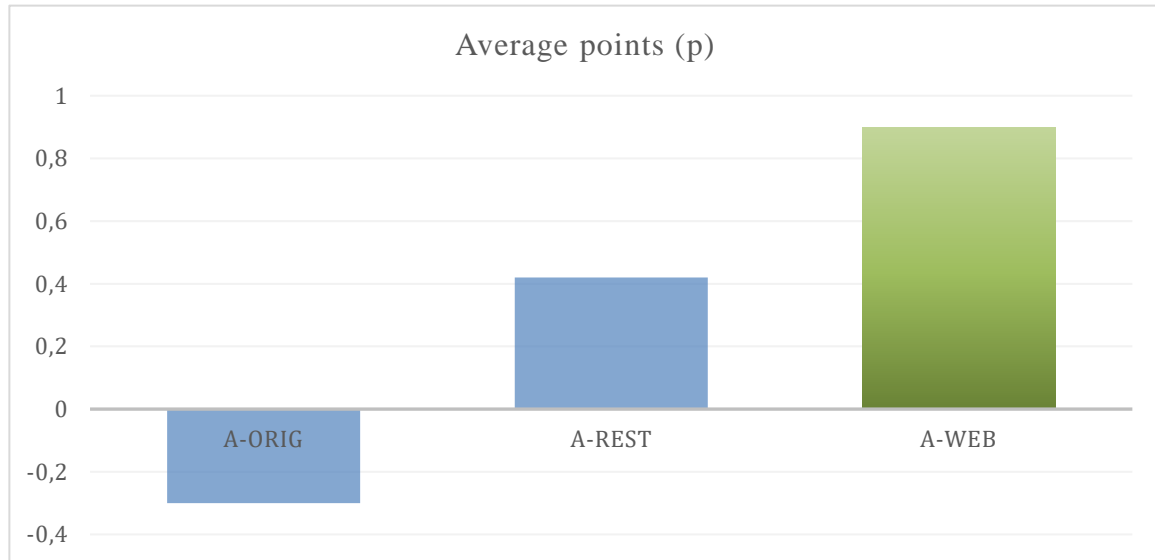


Figure 12: Qualitative questioning results (histogram).

5.3 LOC - Quantitative measuring

For each software architectural alternative, the same five application forms were selected from the corresponding software prototypes. The tables below contain the file names and lines of code needed to implement these application forms in each architectural alternative.

Table 12: LOC measurement results for A-ORIG.

Application form name	File name	LOC
RB-gildi kerfis	KerfiAh.pas	274
Þjóðskrá	Nafnaskra.pas	586
Tegundir skilagreina	SkgrTegund.pas	765
Tegundir trygginga	TegTrygg.pas	655
Iðgjaldaskrá trygginga	TrIdgj.pas	774
Grand total:		3.054

* The application form names are in Icelandic.

Table 13: LOC measurement results for A-REST.

Application form name	File name	LOC
RB-gildi kerfis	KerfiAh.pas	285
	KerfiAhBLL.cs	38
	KerfiAhDAL.cs	35
	KerfiAhService.cs	13
	KerfiAhSQL.bdy	36
	KerfiAhSQL.spc	9
	Total (KerfiAh):	416
Þjóðskrá	Nafnaskra.pas	551
	NafnaskraBLL.cs	21
	NafnaskraDAL.cs	24
	NafnaskraService.cs	6
	NafnaskraSQL.bdy	44
	NafnaskraSQL.spc	11
	Total (Nafnaskra):	657
Tegundir skilagreina	SkgrTegund.pas	768
	SkgrTegundBLL.cs	89
	SkgrTegundDAL.cs	88
	SkgrTegundService.cs	34
	SkgrTegundSQL.bdy	150
	SkgrTegundSQL.spc	55
	Total (SkgrTegund):	1.184
Tegundir trygginga	TegTrygg.pas	639
	TegTryggBLL.cs	81
	TegTryggDAL.cs	68
	TegTryggService.cs	34
	TegTryggSQL.bdy	143
	TegTryggSQL.spc	38
	Total (TegTrygg):	1.003
Iðgjaldaskrá trygginga	TrIdgj.pas	792
	TrIdgjBLL.cs	104
	TrIdgjDAL.cs	99
	TrIdgjService.cs	41
	TrIdgjSQL.bdy	185
	TrIdgjSQL.spc	51
	Total (TrIdgj):	1.272
Grand total:		4.532

* The application form names are in Icelandic.

Table 14: LOC measurement results for A-WEB.

Application form name	File name	LOC
RB-gildi kerfis	KerfiAh.js	161
	kerfiAhReducer.js	71
	KerfiAhBLL.cs	38
	KerfiAhDAL.cs	35
	KerfiAhService.cs	13
	KerfiAhSQL.bdy	36
	KerfiAhSQL.spc	9
	Total (KerfiAh):	363
Þjóðskrá	Nafnaskra.js	257
	NafnaskraReducer.js	50
	NafnaskraBLL.cs	21
	NafnaskraDAL.cs	24
	NafnaskraService.cs	6
	NafnaskraSQL.bdy	44
	NafnaskraSQL.spc	11
	Total (Nafnaskra):	413
Tegundir skilagreina	SkgrTegund.js	326
	skgrTegundReducer.js	141
	SkgrTegundBLL.cs	89
	SkgrTegundDAL.cs	88
	SkgrTegundService.cs	34
	SkgrTegundSQL.bdy	150
	SkgrTegundSQL.spc	55
	Total (SkgrTegund):	883
Tegundir trygginga	TegTrygg.js	274
	TegTryggReducer.js	125
	TegTryggBLL.cs	81
	TegTryggDAL.cs	68
	TegTryggService.cs	34
	TegTryggSQL.bdy	143
	TegTryggSQL.spc	38
	Total (TegTrygg):	763
Iðgjaldaskrá trygginga	TrIdgj.js	346
	TrIdgjReducer.js	153
	TrIdgjBLL.cs	104
	TrIdgjDAL.cs	99
	TrIdgjService.cs	41
	TrIdgjSQL.bdy	185
	TrIdgjSQL.spc	51
	Total (TrIdgj):	979
Grand total:		3.401

* The application form names are in Icelandic.

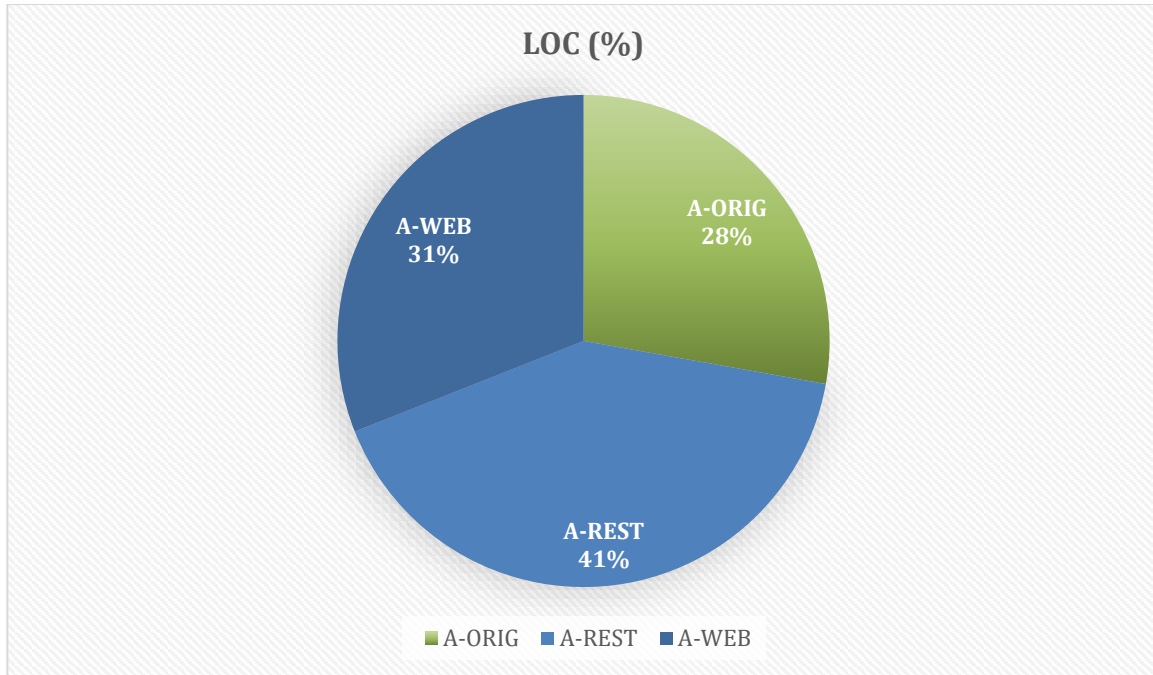


Figure 13: LOC percentage for each software architectural alternative.

5.4 Interpretation of the results

For the ground truth measurement method, a smaller time value indicates that the given architectural alternative is more maintainable. Therefore, the results obtained from conducting the ground truth measurements (see Table 10 and the histogram in Figure 11) establish that the system's original architecture, A-ORIG, is the most maintainable architectural alternative as it has the smallest average implementation time value (1,39) compared to the other two architectural alternatives. Furthermore, A-WEB has the average time value 2,42 and can be considered slightly more maintainable than A-REST which has the average time value 2,70.

For the qualitative questioning method, a larger point value (p) indicates that the given architectural alternative should be considered more maintainable. Therefore, from the results obtained from the qualitative questioning method (see Table 11 and the histogram in Figure 12), it can be concluded that the method predicts the A-WEB software architectural alternative to be the most maintainable as it has the highest average point value (0,90p) compared to the other two alternatives. Furthermore, A-REST has the average point value 0,42p and A-ORIG has the average point value -0,30p. Therefore, the qualitative questioning method predicts that the system's original architecture, A-ORIG, is the worst architectural alternative with respect to maintainability, whilst A-WEB is the most promising alternative. With regard to the proven ground truth, these results seem far from the reality and therefore, based on these results, the qualitative questioning method can be considered a bad maintainability predictor for the case.

For the quantitative measuring method, a smaller LOC value indicates that the given architectural alternative should be considered more maintainable. Therefore, based on the results obtained from the quantitative measuring method (see Table 12, Table 13, Table 14 and the histogram in Figure 13) it can be concluded that the method predicts the system's original architecture, A-ORIG, to be the most maintainable architectural alternative as it has the smallest LOC value 3.054 compared to the other two alternatives. A-WEB has the LOC value 3.401 and can be considered more maintainable than A-REST which has the LOC value 4.532.

The results from the quantitative measuring method are in line with the proven ground truth and therefore the method can be considered a good maintainability predictor for the case.

Based on these results it can be interpreted that the quantitative LOC measurement method's results are closer to the ground truth measurement results than the results obtained from the qualitative questioning method. In the next chapter a more general conclusion is given about the two evaluation methods.

5.5 Threats to validity

As for all studies, it is important to acknowledge the threats to validity that may have existed when carrying out the study. To point out these threats is especially important to establish the trustworthiness of the results and what can then possibly be improved by other researchers conducting similar research in the future [32].

External validity is important to consider as it helps to define how well the findings can be generalised for other cases. It can be considered a threat to the external validity that only a single software system, KRIA, was being used throughout the study.

Internal validity is also important as it involves properly demonstrating that there exists causal relation between variables if it has been indicated or claimed. It can be considered a threat to internal validity that both prototypes (A-REST and A-WEB) were used to compete against a fully developed system (A-ORIG) in both the ground truth measurement method and the quantitative measuring method. That is, as the original system has been fully implemented and software developers have already worked with the system, the A-ORIG alternative may have possibly have unanticipated advantage over the proposed architectural alternatives (A-REST and A-WEB). However, this threat was limited as the proposed architectural alternatives were implemented according to best practices and mimicked programming conventions from the original system.

Reliability is also important to consider as it indicates if the results would have been the same if another researcher would have conducted the same study. Many actions were undertaken throughout the study, simply to increase the reliability of the study. For example, as has been discussed before, when conducting the ground truth, tasks were implemented in a certain order (reduce learning curve), test runs conducted and time measurements redone if unrelated errors occurred etc. Therefore, if another researcher would follow the same principles the results would very likely be the same. Also, the ground truth measurement method was carried out after both the quantitative and qualitative approaches had been applied on the case, simply to eliminate risk from the researcher's perspective.

Finally, the fact the study was being carried out by a single researcher may have affected the outcome, i.e. as the study deals with qualitative data and his interpretation may have been biased by his expectations. However, mainly two types of triangulations were used to minimise that threat. Firstly, data triangulation was achieved as for each evaluation method (and the ground truth), several data sources were being used. For the ground truth measurement, several maintenance tasks and implementation times measurements were being relied on. For the quantitative approach, several source files were relied on. For the qualitative approach several developers/stakeholders participated. To some extent, methodological triangulation was also achieved as three different types of methods were used to predict maintainability quality characteristics of the software architectures. However, this methodological triangulation can be considered weak as one of the methods served as the ground truth. Observer triangulation was however achieved to some extent when applying the qualitative questioning method as several stakeholders helped to construct and review the questions from the questionnaire.

6 CONCLUSION

What can be learned?

This chapter draws conclusions based on the results presented in the previous chapter, Chapter 5 – Results. The appropriateness of the two evaluation methods, quantitative measuring and qualitative questioning when predicting maintainability, is discussed and also the future work that lies ahead.

6.1 Discussion & conclusion

The main findings from this research project are that there are great differences between the two evaluation method approaches, i.e. quantitative measuring and qualitative questioning, when predicting the maintainability of a software system and its software architecture.

Based on the results from the previous chapter the following can be summed up: The quantitative LOC measurement method gives similar results as the ground truth measurement method (the implementation time measurement). The reason being that both methods place the A-ORIG architectural alternative as the most maintainable, A-WEB as the second most maintainable and finally A-REST as the least maintainable alternative. However, the qualitative questioning method gives a very different result as it places the A-WEB architectural alternative as the most maintainable, A-REST as the second most maintainable and finally A-ORIG as the least maintainable alternative. Ironically, in relation to the proven ground truth, the qualitative questioning method therefore predicts the maintainability of the architectural alternatives in precisely the opposite order. Consequently, it can be concluded that the quantitative LOC measurement method provides more reliable evidence than the qualitative questioning method when predicting the maintainability characteristic of KRIA's software architectural alternatives.

By their nature, the two types of evaluation methods are very different in respect to what needs to be present before an assessment can take place. Obviously, for both methods a concrete documentation of each software architectural alternative (e.g. by using component and deployment diagrams) needs to be present. However, before applying the quantitative LOC measurement method on the given case, software prototypes need to be implemented that correspond to the software architectural alternatives at hand. This may require a substantial amount of development effort, which is not needed for the qualitative questioning method as it simply relies on a well formulated questionnaire, entailing subjective questions regarding maintainability characteristics of the architectural alternatives. Therefore, it is disappointing that the qualitative questioning method did not show more promising results as in general the method can be considered more straightforward to use and apply.

It is also worth pointing out that the ground truth measurement method, similar to the quantitative LOC measurement method, requires the implementation of tangible software prototypes. Therefore, one may think that conducting the implementation time measurements require similar amount of effort as with the LOC measurements and therefore; why not just apply and rely on the ground truth measurement method? This is in fact not the case as for the researcher it required significantly more effort to conduct the actual time measurements for the thesis. The main reason being that the method requires a lot of preparation for it to entail trustworthy results, i.e. the participant implementing the maintenance task has to become equally familiar with each environment and also, if a time measurement begins to involve a lot

of development steps unrelated to the given maintainability task being time measured (e.g. unexpected errors occur that need to be generally fixed), the measurement has to be redone etc. Thus, simply applying the quantitative LOC measurement method requires significantly less effort as it only involves collecting the relevant software files and then tools are used to measure the total lines of code within each file.

Surely, the results from the qualitative questioning method were unanticipated, however there may exist some explanation to why the results are so far from the ground truth's results. As the researcher noticed that even though he had clearly emphasised to the respondents that the questions should be answered strictly with respect to maintainability, the respondents were still unconsciously considering other quality attributes. As the qualitative questioning method involves subjective opinions, it can be difficult to eliminate these external factors. They can however be easily avoided when using more objective methods, such as the quantitative LOC measurement method, which can be reasonable when investigating a single quality attribute, such as maintainability. As however typically trade-offs need to be considered between quality attributes when eliciting a software architecture for a system, these external factors should in fact not be eliminated in most cases. Therefore, this does not necessarily have to be considered as being a negative thing for the company/institution doing the architectural selection, as in this way (similar to ATAM) the method opens opportunities for the company to choose efficiently according to the actual underlying business goals. Still, as this study aims to investigate the appropriateness of using these two types of evaluation methods in order to predict maintainability, the qualitative questioning method can be considered an unreliable maintainability predictor for the case but the quantitative LOC measurement method a good predictor.

For Fuglar Ltd. as a software firm having to choose the most suitable software architectural alternative for their software system KRIA, these quantitative and qualitative results can be quite misleading but at the same time useful. The results from the quantitative LOC measurement method and the ground truth confirm that Fuglar Ltd. already has a very maintainable architectural alternative (A-ORIG) for its software system KRIA. Therefore, if they were to implement the system with one of the two other architectural alternatives, they would at the same time be reducing the system's maintainability characteristic significantly. On the other hand, even though the qualitative questioning approach did not serve as a good maintainability predictor, it certainly helped the company to discover its main business goals and the importance of other quality attributes when choosing a software architectural alternative.

6.2 Future work

A lot of work can potentially be performed in the field of line relating to the thesis work. As this research study was done in collaboration with a single software firm Fuglar Ltd., it would be interesting for future researchers to investigate the two evaluation methods further, e.g. by using other case studies as inputs and see if the results can then be further generalised. The researcher and author of this study for example believes that other companies or institutions in a similar situation as Fuglar Ltd. would obtain similar results. That is, the quantitative LOC measurement method would in most cases agree with the ground truth while the qualitative questioning method would be very risky to use in order to predict maintainability as it very much depends on the respondents subjective beliefs (not systematic). As however, during this case study research there may have existed potential hidden factors and since the research has not been conducted using other cases, the general case remains uncertain.

Additionally, as has previously been discussed, numerous other potential quantitative and qualitative methods to predict maintainability exist. Investigating the reliability of another set of methods may therefore be interesting, especially since the qualitative questioning approach used throughout the thesis gave unanticipated results. Methods, such as the ATAM method (which was demonstrated on Fuglar's case) could therefore instead be investigated but as the ATAM method is quite a comprehensive method, it may be reasonable to simply compare ATAM with a ground truth measurement method, i.e. skip the quantitative investigation when conducting analysis with ATAM.

Finally, maintainability characteristics of software systems is a rising importance for the software engineering community in general. Therefore, doing further research on how to compare software architectures with respect to maintainability may be worthwhile.

References

- [1] International Organization For Standardization Iso, “ISO/IEC 9126-1,” *Software Process: Improvement and Practice*, vol. 2. pp. 1–25, 2001.
- [2] L. Dobrica and E. Niemela, “A survey on software architecture analysis methods,” *IEEE Trans. Softw. Eng.*, vol. 28, pp. 638–653, 2002.
- [3] R. Keele, “Nursing research and evidence-based practice,” *Quant. Versus Qual. Res. or Both?*, vol. 1 Sudbury, pp. 35–52, 2011.
- [4] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, and A. Zaremski, “Recommended Best Industrial Practice for Software Architecture Evaluation,” 1997.
- [5] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, vol. 2. 2012.
- [6] Joy Frechtling, S. Hood, Henry Frierson, and Gerunda Hughes, “The 2002 User-Friendly Handbook for Project Evaluation for Project Evaluation,” *Communication*, pp. 1–92, 2002.
- [7] P. Tapio, R. Paloniemi, V. Varho, and M. Vinnari, “The unholy marriage? Integrating qualitative and quantitative information in Delphi processes,” *Technol. Forecast. Soc. Change*, vol. 78, pp. 1616–1628, 2011.
- [8] Frank McKenna, “Thin-client vs Fat-client computing,” pp. 1–7, 2002.
- [9] A. H. Watson, T. J. McCabe, and D. R. Wallace, “Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric,” *NIST Spec. Publ.*, vol. 500, pp. 1–114, 1996.
- [10] D. Garlan and M. Shaw, “An Introduction to Software Architecture,” *Knowl. Creat. Diffus. Util.*, vol. 1, pp. 1–40, 1994.
- [11] N. Rozanski and E. Woods, “Applying Viewpoints and Views to Software Architecture,” *Comput. Methods Programs Biomed.*, p. 11, 2011.
- [12] J. Ivers, P. C. Clements, D. Garlan, R. Nord, B. Schmerl, and O. Silva, “Documenting Component and Connector Views with UML 2.0 (CMU/SEI-2004-TR-008),” p. 60, 2004.
- [13] Tutorialspoint, “UML - Component Diagrams.” [Online]. Available: https://www.tutorialspoint.com/uml/uml_component_diagram.htm. [Accessed: 14-Mar-2017].
- [14] Tutorialspoint, “UML - Deployment Diagrams.” [Online]. Available: https://www.tutorialspoint.com/uml/uml_deployment_diagram.htm. [Accessed: 15-Mar-2017].
- [15] S. W. Ambler, *The Object Primer: Agile Model Driven Development with UML 2*, 3rd ed. Cambridge University Press, 2004.
- [16] Sparx, “UML 2 Deployment Diagram.” [Online]. Available: http://www.sparxsystems.com.au/resources/uml2_tutorial/uml2_deploymentdiagram.html. [Accessed: 15-Mar-2017].
- [17] DevIQ, “Separation of Concerns.” [Online]. Available: <http://deviq.com/separation-of-concerns/>. [Accessed: 28-Jan-2017].

- [18] B. Kanagwa and E. K. Mugisa, “A Comparison of Service Oriented Software Architectures,” pp. 405–416, 2006.
- [19] T. Fredrich, “RESTful Service Best Practices: recommendations for creating web services,” pp. 1–34, 2013.
- [20] S. E. I.-C. M. University, “Architecture Tradeoff Analysis Method.” [Online]. Available: <http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm>. [Accessed: 03-May-2017].
- [21] R. Kazman, M. Klein, and P. Clements, “ATAM : Method for Architecture Evaluation,” *Cmusei*, vol. 4, p. 83, 2000.
- [22] D. M. Fournier, “The Program Evaluation Standards: How to Assess Evaluations of Educational Programs,” *J. Educ. Meas.*, vol. 31, pp. 363–367, 1994.
- [23] R. Mann, “Qualitative Metrics,” 2013. [Online]. Available: <http://www.credohighered.com/blog/2013/8/9/qualitative-metrics>. [Accessed: 08-Feb-2017].
- [24] S. E. Wyse, “What is the Difference between Qualitative Research and Quantitative Research?,” 2011. [Online]. Available: <https://www.snapsurveys.com/blog/what-is-the-difference-between-qualitative-research-and-quantitative-research/>. [Accessed: 08-Feb-2017].
- [25] Stevens, “Post Hoc Tests in ANOVA,” pp. 1–4, 1999.
- [26] Oracle, “Oracle Data Access Components.” [Online]. Available: <http://www.oracle.com/technetwork/database/windows/whatsnew/index.html>. [Accessed: 28-Feb-2017].
- [27] W. Buchanan, *Mastering Delphi Programming*. Palgrave Macmillan, 2003.
- [28] “Oracle Database 12c PL/SQL.” [Online]. Available: <http://www.oracle.com/technetwork/database/features/plsql/index.html>. [Accessed: 05-Mar-2017].
- [29] Microsoft, “Visual Studio.” [Online]. Available: <https://www.visualstudio.com/>. [Accessed: 05-Mar-2017].
- [30] Microsoft, “Active Directory Federation Services.” [Online]. Available: <https://msdn.microsoft.com/en-us/library/bb897402.aspx>. [Accessed: 17-Mar-2017].
- [31] JetBrains, “WebStorm.” [Online]. Available: <https://www.jetbrains.com/webstorm/>. [Accessed: 05-Mar-2017].
- [32] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empir. Softw. Eng.*, vol. 14, pp. 131–164, 2009.
- [33] C. Robson, “Real World Research: A Resource for Social Scientists and Practitioner-Researchers,” *Blackwell Publishing*. pp. 1–608, 2002.
- [34] A. Williams, “How to write and analyze a questionnaire,” *J. Orthod.*, vol. 30, pp. 245–252, 2003.
- [35] P. Oman and J. Hagemeister, “Metrics for assessing a software system’s maintainability,” *Proc. Conf. Softw. Maint. 1992*, pp. 337–344, 1992.

- [36] N. Kukreja, “Measuring Software Maintainability.” [Online]. Available: <https://quandarypeak.com/2015/02/measuring-software-maintainability/>. [Accessed: 02-Mar-2017].
- [37] A. van Deursen, “Think Twice Before Using the ‘Maintainability Index.’” [Online]. Available: <https://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index/>. [Accessed: 02-Mar-2017].
- [38] J. Schofield, “The Statistically Unreliable Nature of Lines of Code,” pp. 29–33.
- [39] R. E. Stake, “The art of case study research,” *The art of case study research*. pp. 236–245, 1995.

Appendix

Additional information

Appendix A – Software architectural alternatives

A.1 A-ORIG

Below, a code snippet from the original system is presented. Furthermore, a typical application form from the original system is shown and a screenshot from the Delphi IDE is included.


Code snippet 1: Delphi source code needed to implement a typical create feature in A-ORIG.

```
function TfmTegTrygg.StofnaTegTr : boolean;
var
  Rkodi : boolean;
  SqlStr : string;
begin
  Rkodi := True;
  SqlStr := '';
  SqlStr := SqlStr + 'insert into ' + Login.LSchemaDot + 'TegTr ';
  SqlStr := SqlStr + ' (Teg_Tr,Heiti,TegTrTab,Yf_Texti,Skrad_Dags,Skrad_Af) ';
  SqlStr := SqlStr + ' values ';
  SqlStr := SqlStr + ' (:Teg_Tr,:Heiti,:TegTrTab,:Yf_Texti,trunc(sysdate),user) ';
  SqlSkipun.SQL.Clear;
  SqlSkipun.SQL.Add(SqlStr);

  SqlSkipun.ParamByName('Teg_Tr').AsString := Trim(Teg_Tr.Text);
  SqlSkipun.ParamByName('Heiti').AsString := Trim(Heiti.Text);
  SqlSkipun.ParamByName('TegTrTab').AsString := glob.FWord(TegTrTab.Text);
  SqlSkipun.ParamByName('Yf_Texti').AsString := Trim(Yf_Texti.Text);

  Rkodi := True;

  try
    SqlSkipun.EXECsql;
  except
    on EDatabaseError do begin
      Rkodi := False;
      SqlSkipun.close;
      result := Rkodi;
      exit;
    end;
  end;
  if Rkodi then begin
    SqlSkipun.close;
    SqlSkipun.SQL.Clear;
    SqlSkipun.SQL.Add('Commit');
    SqlSkipun.EXECsql;
    SqlSkipun.close;
  end;
  SqlSkipun.close;
  result := Rkodi;
end;
```


Iðgjaldaskrá trygginga [FUGLDEV]

Takmörkun lista

Tegund tryggingar:

Aldur:

Leita/Lista

Hreinsa

Ný færsla

Breyta

Eyða

Loka

Listun á færslum í iðgjaldaskrá trygginga

Tegund tryggingar	Aldur
...	...
...	...
...	...
...	...
...	...
...	...
...	...
...	...
...	...
...	...

Skráning í iðgjaldaskrá trygginga

Tegund tryggingar:

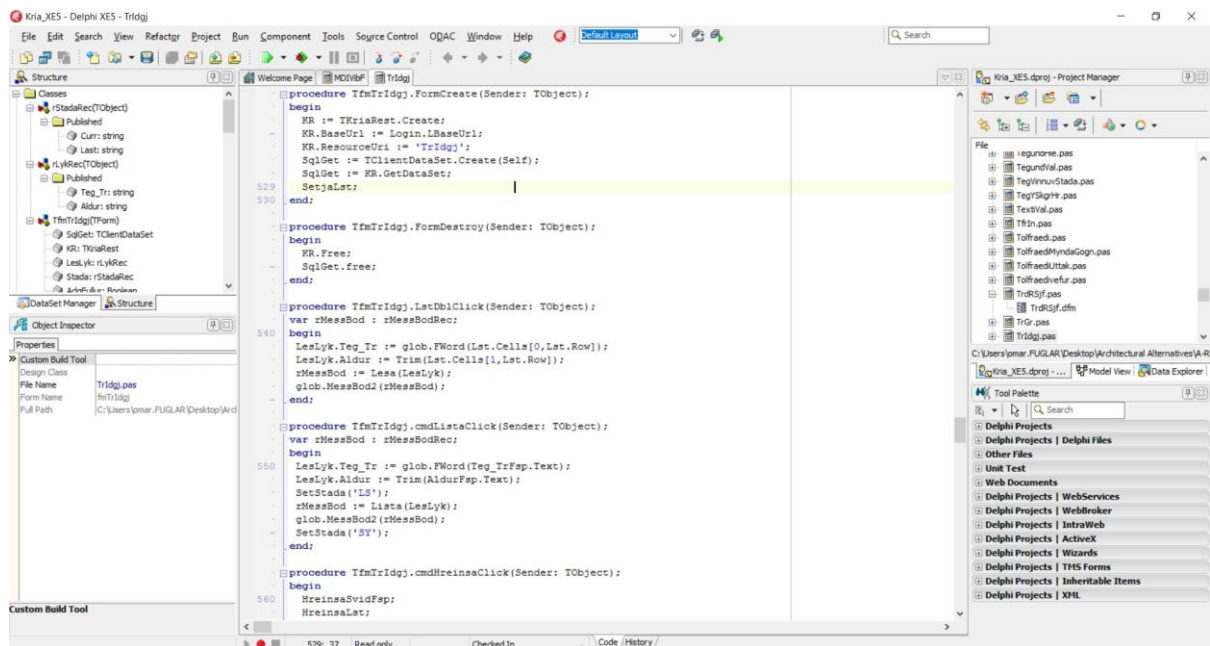
Aldur:

iðgjaldahlutf. karla:

iðgjaldahlutf. kvenna:

Fast iðgjald á ári:

kr.



A.2 A-REST

Below, code snippets from the A-REST software prototype are presented. As can be observed, the Delphi source code becomes simpler but new layers have been added to the REST service (C# code) and SQL stored procedures. The GUI remains the same as for A-ORIG, see in Figure 14 and the Delphi IDE has already been demonstrated in Figure 15. The API for the RESTful web-service is presented and the development environment (Visual Studio) is demonstrated with figures.

Code snippet 2: Delphi source code needed to implement a typical create feature in A-REST.

```
function TfmTegTrygg.StofnaTegTr : boolean;
var
  Rkodi : boolean;
  SqlStr : string;
begin
  try
    KR.Action := 'Create';
    KR.Parameters.Clear;
    KR.Parameters.Add('Teg_Tr=' + Trim(Teg_Tr.Text));
    KR.Parameters.Add('Heiti=' + Trim(Heiti.Text));
    KR.Parameters.Add('TegTrTab=' + glob.FWord(TegTrTab.Text));
    KR.Parameters.Add('Yf_Texti=' + Trim(Yf_Texti.Text));

    if KR.Execute then begin
      result := True;
    end
  else begin
    ShowMessage(KR.MessBod.RText + ' - ' + KR.MessBod.RDetail);
    glob.MessBod('Ekki gekk að stofna nýja færslu í gegnum þjónustukall !');

    result := False;
  end;
except
  on EDatabaseError do begin
    result := False;
  end;
end;
end;
```

Code snippet 3: C# code from the main layers of the RESTful web-service needed to implement a typical create feature in A-REST. Note, in the actual web-service, each function is stored in a separate programming file corresponding to the appropriate layer.

```
// Service front-end
[HttpPost, LogAction]
public ResultStatus TegTrygg([FromBody] CreateTegTryggReq req)
{
  return CreateService.TegTrygg(req.Teg_Tr, req.Heiti, req.TegTrTab, req.Yf_Texti);
}

// Service Business logic layer
public ResultStatus TegTrygg(string Teg_Tr, string Heiti, string TegTrTab, string Yf_Texti)
{
  using (DbContext db = new DbContext(User))
  {
    ResultStatus result = db.Create.TegTrygg(Teg_Tr, Heiti, TegTrTab, Yf_Texti);
    if (result.Status.Ok())
    {
      db.SaveChanges();
    }
    else
    {
      db.UndoChanges();
    }
  }
  return result;
}

// Service Data access layer
public ResultStatus TegTrygg(string Teg_Tr, string Heiti, string TegTrTab, string Yf_Texti)
{
  ResultStatus result = new ResultStatus();
  List<OracleParameter> dbParams = new List<OracleParameter>();
  dbParams.Add(new OracleParameter("v2Teg_Tr", OracleDbType.Varchar2, Teg_Tr,
    System.Data.ParameterDirection.Input));
```

```

dbParams.Add(new OracleParameter("v2Heiti", OracleDbType.Varchar2, Heiti, System.Data.ParameterDirection.Input));
dbParams.Add(new OracleParameter("v2TegTrTab", OracleDbType.Varchar2, TegTrTab,
System.Data.ParameterDirection.Input));
dbParams.Add(new OracleParameter("v2Yf_Texti", OracleDbType.Varchar2, Yf_Texti,
System.Data.ParameterDirection.Input));
Status status;
_oraHelper.RunVoidFunction("TegTrygg", _packagename, dbParams, out status);
result.Status = status;
return result;
}

```

Code snippet 4: SQL stored procedure source code needed for a typical create feature in A-REST.

```

Procedure TegTrygg (
  v2Teg_Tr in VARCHAR2, v2Heiti in VARCHAR2,
  v2TegTrTab in VARCHAR2, v2Yf_Texti in VARCHAR2,
  nOutReturnKodi Out number, v2OutMessBod Out varchar2, v2OutMessBodDetail Out varchar2
) as Begin
  nOutReturnKodi := -1;
  v2OutMessBod := Null;
  v2OutMessBodDetail := Null;

  insert into TegTr (Teg_Tr,Heiti,TegTrTab,Yf_Texti,Skrad_Dags,Skrad_Af)
  values (v2Teg_Tr,v2Heiti,v2TegTrTab,v2Yf_Texti,sysdate,Kria.GlbClientUser);

  nOutReturnKodi := 0;
  v2OutMessBod := Null;
  v2OutMessBodDetail := Null;
  Exception When Others Then
    nOutReturnKodi := -1;
    v2OutMessBod := 'Ekki gekk að stofna !';
    v2OutMessBodDetail := sqlerrm;
End;

```

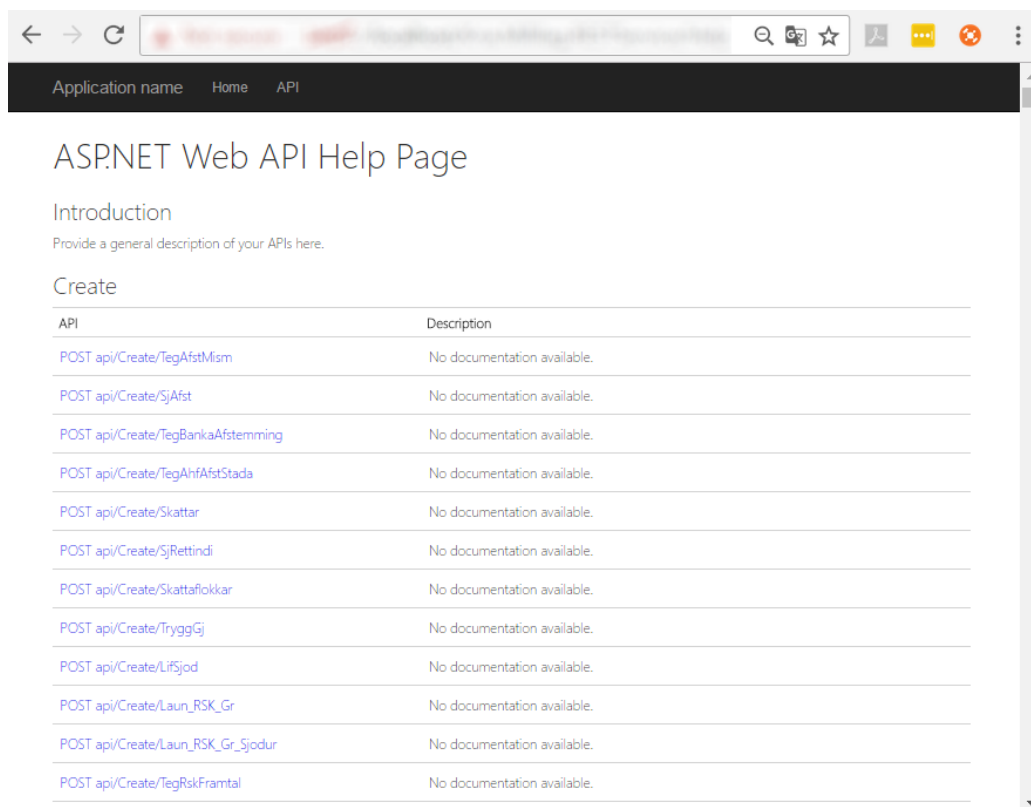


Figure 16: The web-API for the RESTful web-service.

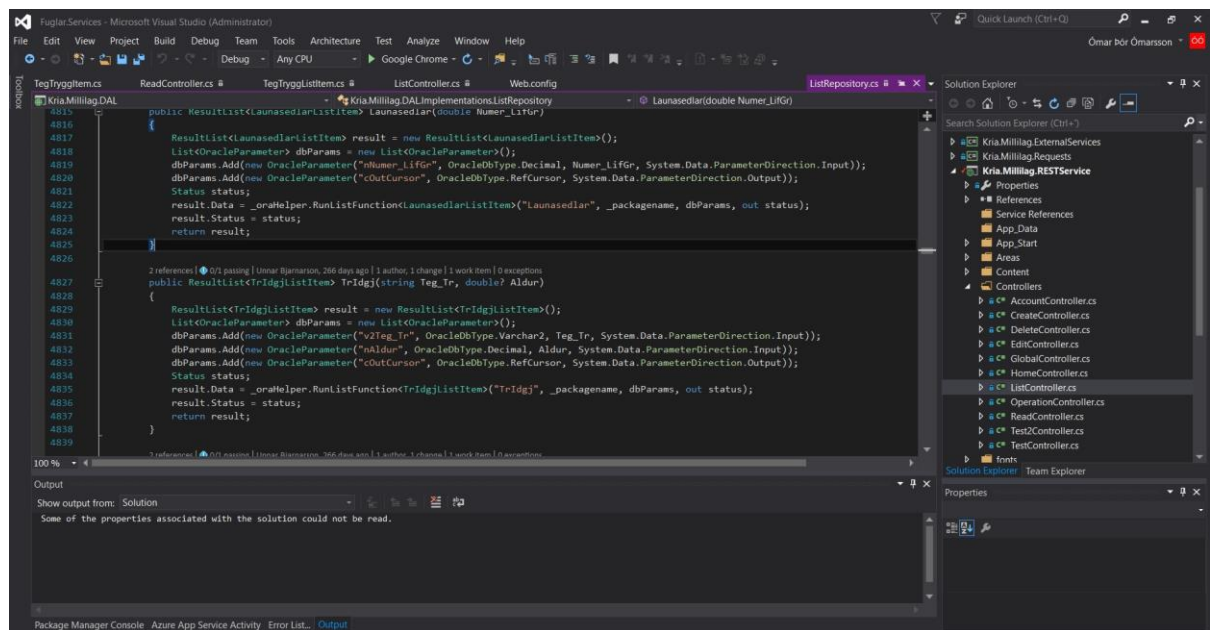


Figure 17: Microsoft Visual Studio IDE – used to develop the RESTful web-service.

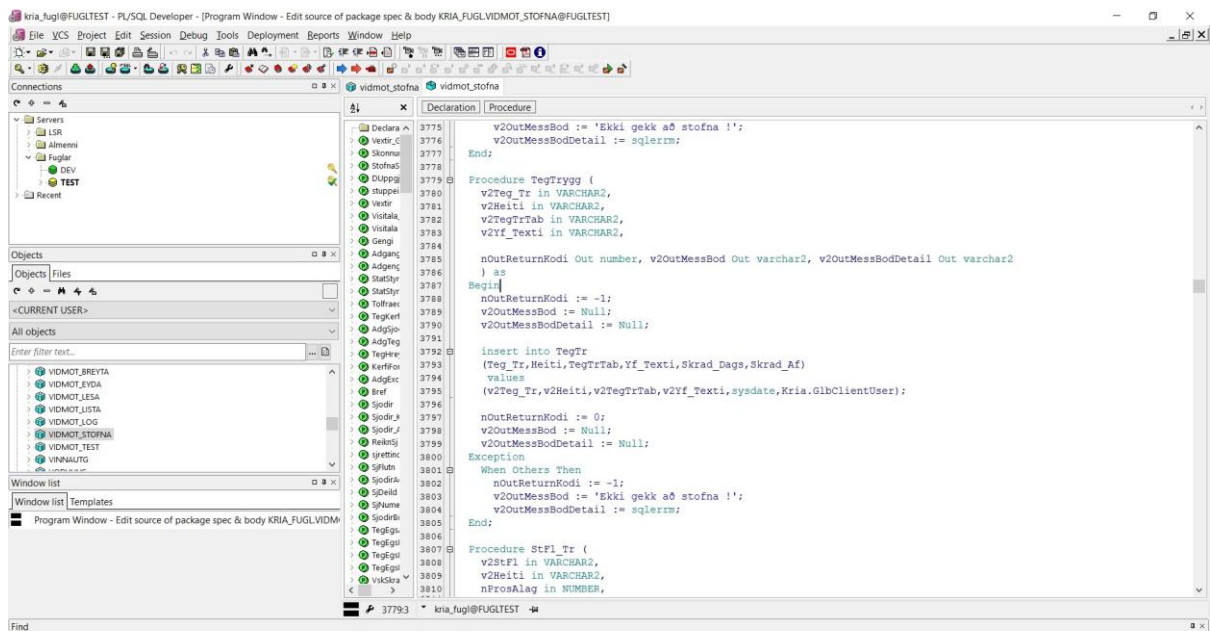


Figure 18: PL/SQL IDE – used for the database management (e.g. stored procedures).

A.3 A-WEB

Below, a code snippet from the website application needed for the A-WEB architectural alternative's prototype is presented. It was programmed with ReactJS/Redux and the WebStorm IDE entails good support for these JavaScript libraries. Screenshot from the IDE can be observed in Figure 22. Figure 20 and Figure 21 demonstrate some of the GUI. This architectural alternative, similarly to A-REST, relies on the intermediary web-service.

Code snippet 5: JavaScript source code needed to implement a typical create feature in A-WEB.

```
createEditClick_button() {  
  var parameters = {  
    Teg_Tr: this.props.Teg_Tr_modal,  
    Heiti: this.props.Heiti_modal,  
    TegTrTab: this.props.TegTrTab_modal,  
    Yf_Texti: this.props.Yf_Texti_modal  
  };  
  var payload_data = {  
    Teg_Tr: this.props.Teg_Tr_modal,  
    Heiti: this.props.Heiti_modal,  
  };  
  api_request(form,"Create",parameters,payload_data);  
  this.setState({modal_createEditIsOpen: false});  
}  
  
// Create reducer  
case form + "Create": {  
  return {...state, created: false, fetching: true, error: false}  
}  
case form + "Create_FAILED": {  
  return {...state, fetching: false, error: true, errorMessage: action.payload}  
}  
case form + "Create_FULFILLED": {  
  return {  
    ...state,  
    fetching: false,  
    list: [...state.list, action.payload],  
  }  
}
```

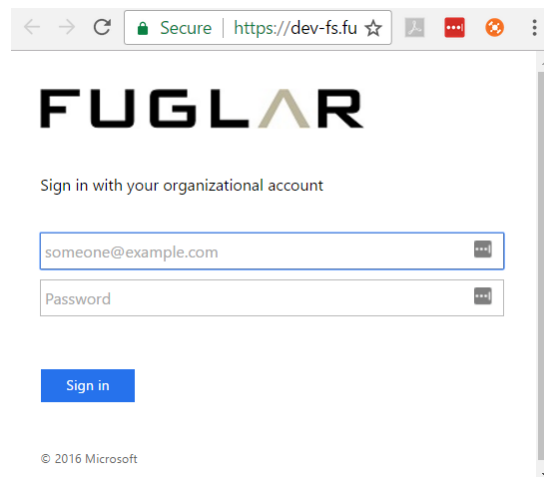


Figure 19: User's ADFS sign-on page.

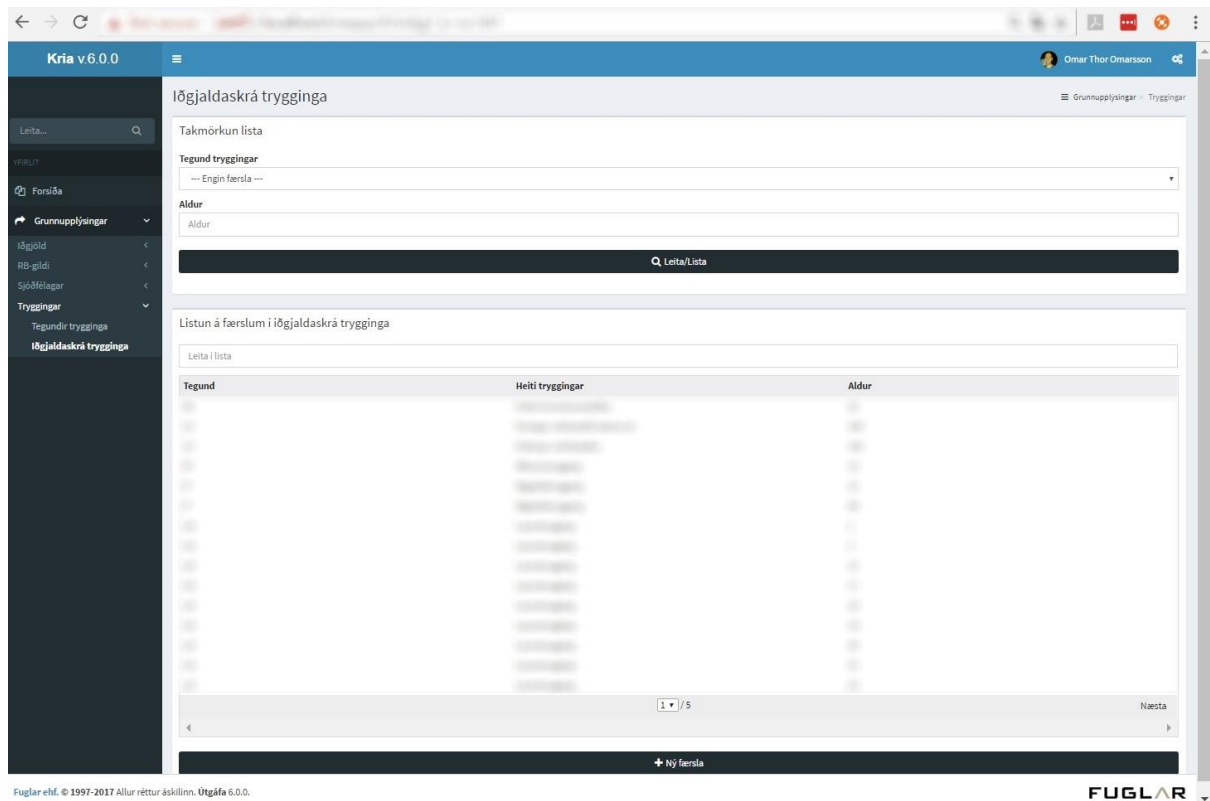


Figure 20: Part of an application form from the A-WEB prototype – presents a list.

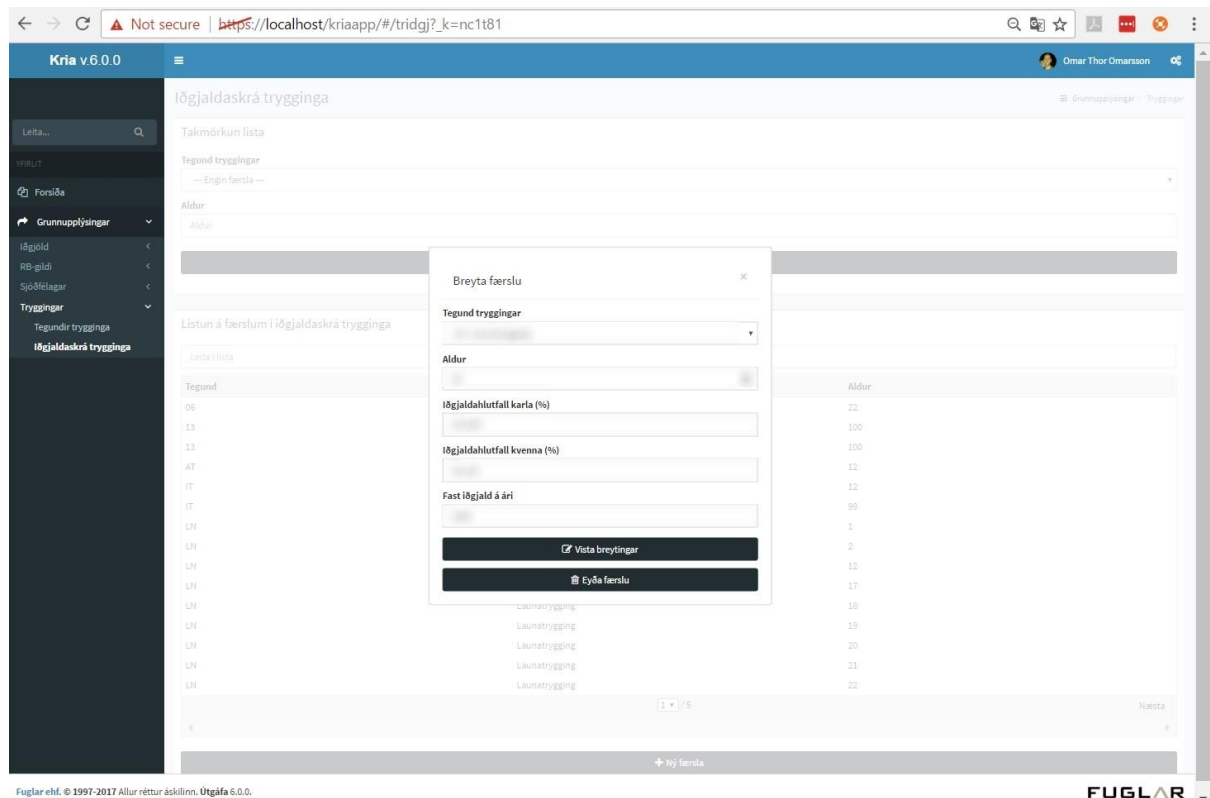


Figure 21: Part of an application form from the A-WEB prototype – allows the user to delete, modify or add a new entry to a list.

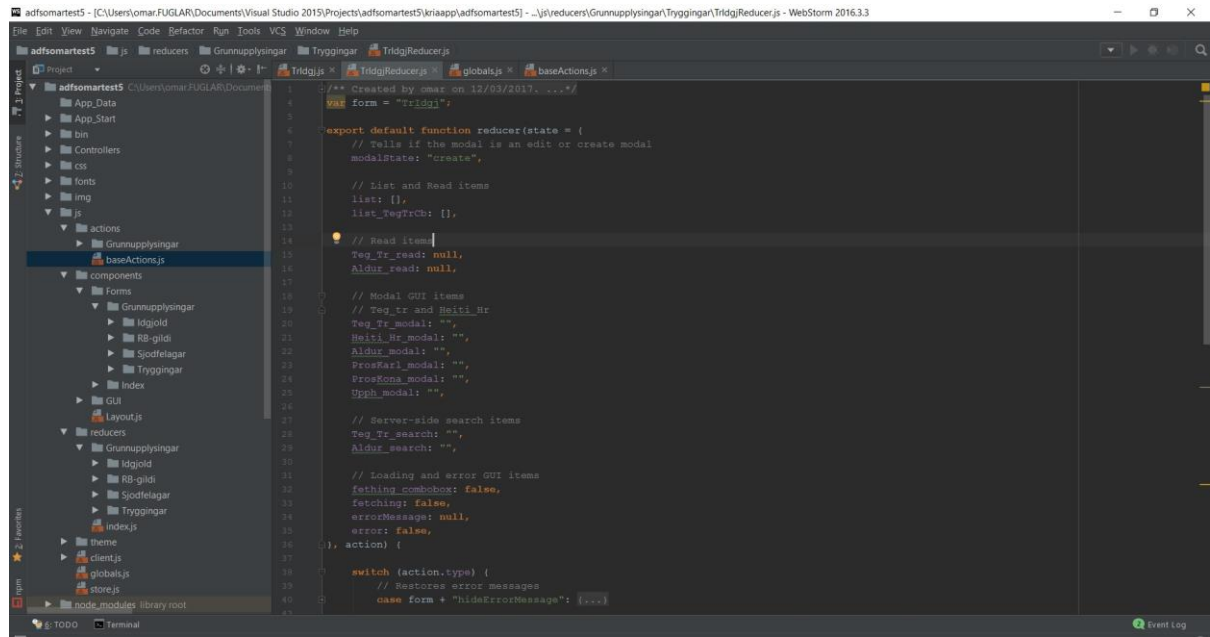


Figure 22: WebStorm IDE – used to develop the ReactJS/Redux website in A-WEB.

Appendix B – The ground truth measurement method

Table 15 below contains the maintainability tasks used for the ground truth measurement method in the study.

Table 15: Maintenance tasks for KRIA and their corresponding weight values.

ID	Maintenance task/scenario description	Weight
T1	A new column needs to be added to an application form list (see in relation to Figure 3). <i>Note</i> , it is important that the user can interact with this new column field, i.e. can still create, edit and delete list entries normally.	5
T2	Create a brand-new application form with all necessary functionalities (see in relation to Figure 3). <i>Note</i> , this involves defining the CRUDL functions and implementing the GUI.	3
T3	Remove a column from a list in an application form (see in relation to Figure 3). <i>Note</i> , it is important that the column is completely removed, i.e. side-effects in CRUDL functions are also removed.	1
T4	A new search field needs to be added to a form for the purpose of server-side searching in a list.	2
Total weight:		11

Table 16: Detailed maintenance task description.

ID	Detailed description (step-by-step)
T1	<ol style="list-style-type: none"> 1. Use the original application form: “Tegundir skilagreina”. 2. Add a column with the name “Test” into the database table “Skgr_Tegund”. 3. The type of the column should be “VARCHAR2(10)” in the database. 4. Add the same column to the table in the GUI. The name should be displayed as “Test column”. 5. Add a new input field to the form to allow the user to interact with this column. 6. Make sure that the CRUDL functions still work accordingly.
T2	<ol style="list-style-type: none"> 1. Create a new application form called “Test”. 2. Add a new database table with the name “Test”. 3. Add two new columns to the database table, i.e. “TestC1” and “TestC2”. 4. The columns should be of type “VARCHAR2(10)”. 5. Display the columns in a table in the GUI. 6. Add new input fields to the GUI form to allow the user to interact with the table’s columns. 7. Make sure that the CRUDL functions work accordingly.
T3	<ol style="list-style-type: none"> 1. Use the original application form: “Tegundir skilagreina”. 2. Remove the column “Test” from the database table “Skgr_Tegund”. 3. Remove the same column from the table in the GUI. 4. Remove the corresponding input field from the GUI. 5. Make sure that the CRUDL functions still work accordingly.
T4	<ol style="list-style-type: none"> 1. Use the original application form: “Iðgjaldaskrá trygginga”. 2. Add a new search field “Heiti” to the GUI. 3. Edit the main list function so that the user can search in the table according to “Heiti”.

Appendix C – The questionnaire for the qualitative questioning method

Software architectural alternatives: A-ORIG, A-REST, A-WEB.

Response options:

- Good architectural alternative (+1p): _____
- Reasonably good architectural alternative (+0,5p): _____
- Bad architectural alternative (-1p): _____

Questions:

1. Based on the architectural patterns and tactics being used to construct the given architectural alternative, do you think that the code is easy to understand and change in the future (e.g. since the code has been separated into layers etc.)?
2. Are the integrated development tools (IDEs) that are being used to maintain and evolve the given architectural alternative modern and easy to handle? For example, do they provide relevant auto completion, identification of coding errors etc.?
3. When coding problems occur that are directly related to the programming languages and IDEs being used, does the given architectural alternative have a good development community? For example, is it easy to search on the Internet for solution to a coding problem when it occurs?
4. Do you think that the given architectural alternative has good future potentials?
5. Maintaining good user security is easy for the given software architectural alternative?
6. Adding brand new GUI features is possible and relatively efficient for the given software architectural alternative?
7. For the given software architectural alternative, is it efficient for multiple programmers to co-operate on evolving and maintaining the system?
8. Based on your previous answers, do you consider the given architectural alternative efficient to maintain?